CS195V Week 10

Introduction to CUDA

Plan

- There's about a month before reading period
- We plan to spend one or two lectures on CUDA introductions, then...
  - Anyone want to present?
- Intro CUDA project will go out soon
  - Video filtering
  - Should be pretty simple

CUDA

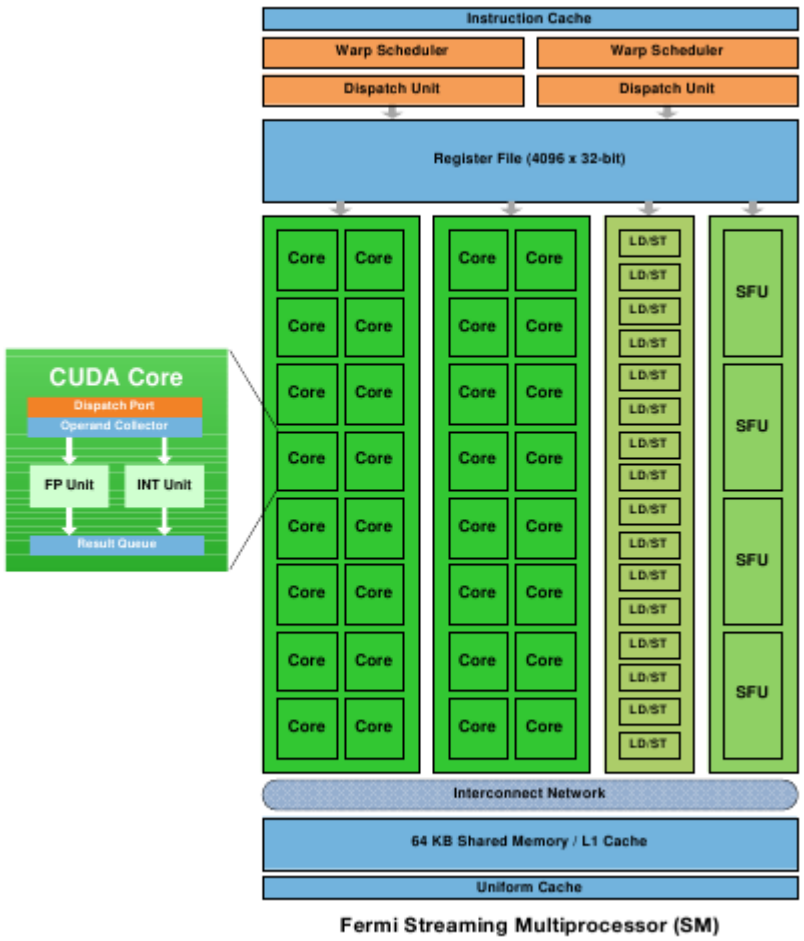
General Purpose GPU Programming

- With the introduction of the unified device architecture (UDA) GPU hardware was made generic
  - No longer specific hardware for each shader stage
  - Same hardware is responsible for all shader stages
  - Much more generic purpose hardware
- OpenCL and CUDA are the main GPGPU languages today (DirectCompute was recently introduced by MS)
  - Very similar languages - people tend to use CUDA more even though it is NVIDIA specific (more support / slightly better performance)

CUDA

- Compute Unified Device Architecture
- Released by NVIDIA first in 2007
  - Only supported on 8 series cards and later
- Framework for general purpose computation on the GPU
- Written mostly like C, so easy to convert over
- But lots of little caveats
  - The C programming guide at http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf tells you pretty much all you need to know, but it's long and boring

The Fermi Device Architecture



Fermi Streaming Multiprocessor (SM)

- The warp schedulers dispatch warps (groups of 32 threads) to the shader cores
- GTX 460 has 336 shader cores or stream processors @ 1350 MHz
  -

Getting it to work

- You need the CUDA toolkit from NVIDIA
  - /contrib/projects/cuda-toolkit/ (Toolkit)
  - /contrib/projects/cuda-sdk/ (SDK)
- Your kernel and kernel launching code goes in a .cu file (headers can go in .cuh)
  - Compiled separately by nvcc
- Tell your makefile what is CUDA code and what is C/C++ code
  - We do it for you, but the NVIDIA code samples have example makefiles, don't worry, they're short and simple

Some vocabulary that you might see

- Host: The regular computer
- Device: The GPU
- Kernel: A function made to be executed many times in parallel on the GPU
  - Origins in stream processor programming, where a kernel function is applied to each element in the stream (SIMD)
  - Shaders are a specific type of kernel
- nvcc: The NVIDIA compiler for CUDA code
- ptx: GPU assembly language
  - nvcc compiles your kernel to ptx first, and then ptx to binary code

CUDA program structure

- Most CUDA programs operate like...
  - Copy some data to Device memory
  - Kernel launch (run kernel function on some data)
  - Wait for kernels to finish
  - Copy data back to Host memory
  - Keep going your merry way
- It's a simple model, but you can do a lot of things with it
  - Parallelization is implicit - your kernel function is launched thousands of times (for each piece of data)
- Also it means that much of your CUDA code will look similar (or copy pasted verbatim)

CUDA Threading Model

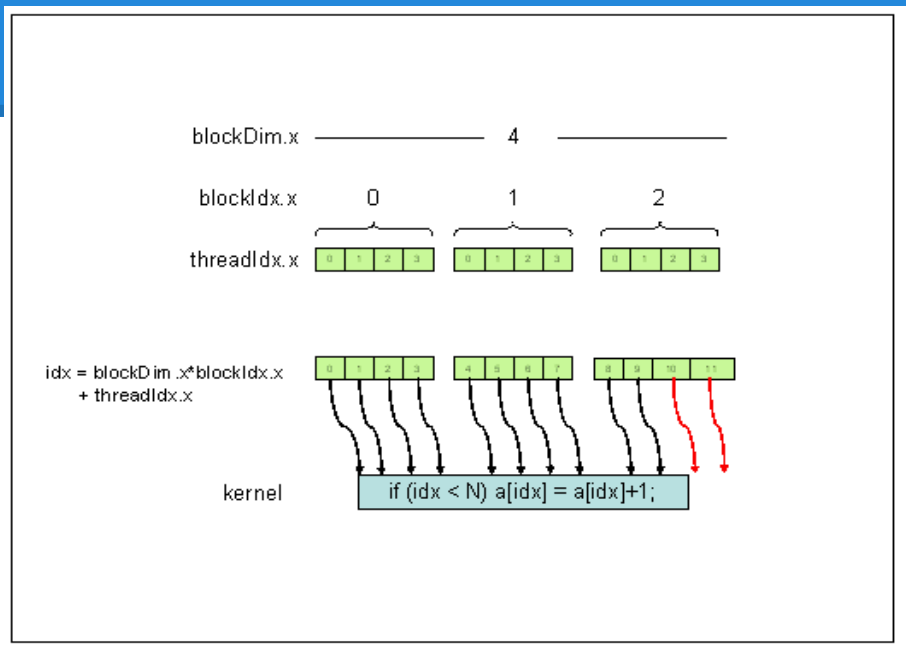
- CUDA programs are a hierarchy of concurrent threads
  - Threading is subdivided into blocks, each of which are then subdivided into threads
  - Choice of subdivision is up to you!
  - Note that (num of blocks) x (threads per block) = total number of threads
- Choosing the number of threads per block and number of blocks can be tricky
  - Largely depends on your data, but different numbers can have different performance (more on this later)
  - Note that the hardware limits number of blocks in a launch to 65,535 and the number of threads per block depends on the GPU (about 512)

Some more vocabulary

- Thread: A single execution thread, as you would expect, runs on a single GPU microprocessor
- Thread Block: A group of threads that will be issued the same instructions at the same time, they can also share some memory
  - Note that the order of block execution is random
- Grid: A group of all of the thread blocks that you are running
- Usually you know how many total threads you want to launch, so the question is, how do you partition them into blocks?

A Simple Kernel

```
#include <cuda.h>
__global__ void kernelfunction(float *a, int N)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if (idx<N) {
     a[idx] = a[idx] + 1;
  }
}
```



- Given an array of floats and the array length, this kernel adds one to each value
- blockIdx, blockDim, and threadIdx are built in variables
  - blockIdx.x/y/z is the current block index
  - blockDim.x/y/z is the block dimension (size of block)
  - threadIdx.x/y/z is the current thread in the block
- So each thread in each block doubles a value in a
- We need to check if idx < N to prevent writing past the array
  - If we start the kernel with more total threads than array elements (ie. the number elements may not be evenly divisible into blocks / threads per block)

A Simple Kernel

- __global__ keyword defines a function as being a kernel run on the device that can be launched from the host
  - __device__ can be used for subroutines launched from the device run on the device (gets compiled by nvcc and gcc)
- Note that recursion is supported only on newer devices (compute capability 2+)
  - But you should avoid using it (may see up to 30% performance hit)
- Now to run the kernel...

Launching the Kernel

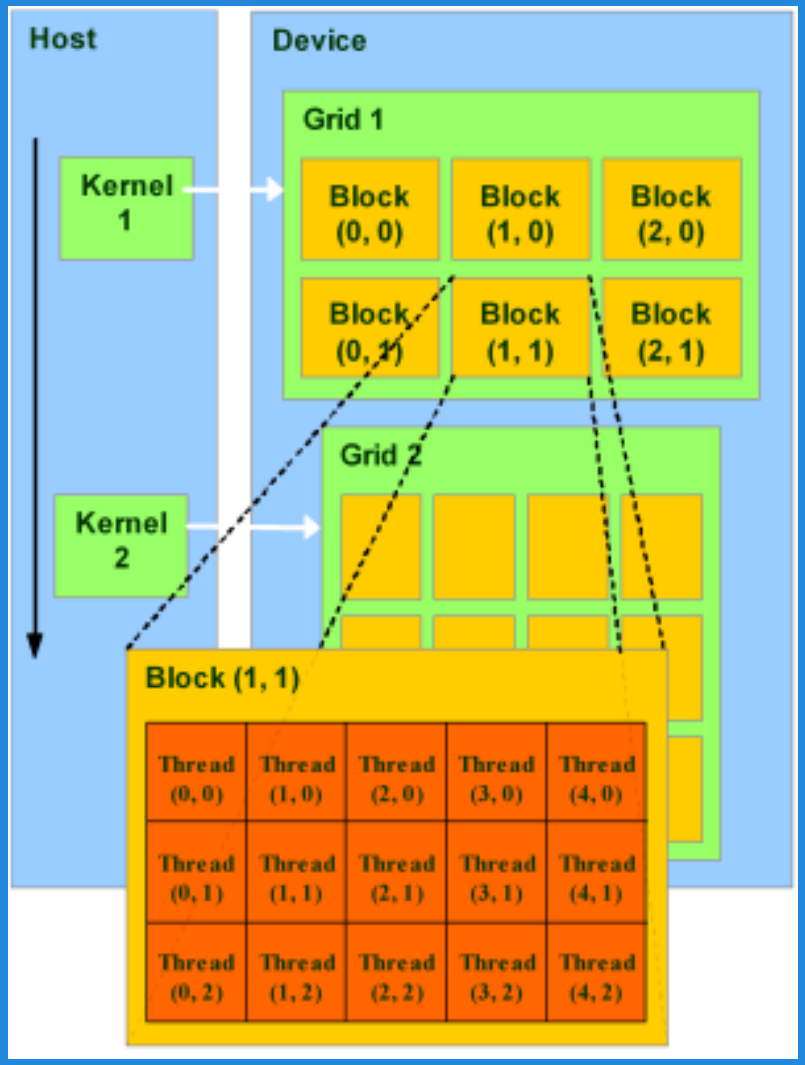
```
int c_kernel_launch(float *a_h, int N)
{
  float *a_d;                  // pointer to device memory
  int i;
  size_t size = N*sizeof(float);
  // allocate array on device
  cudaMalloc((void **) &a_d, size);
  // copy data from host to device
  cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);
  // do calculation on device:
  // Compute execution configuration
  int blockSize = 4;
  int nBlocks = N/blockSize + (N%blockSize == 0?0:1);

  // kernel launch
  kernelfunction <<< nBlocks, blockSize >>> (a_d, N);

  // Retrieve result from device
  cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
  // cleanup
  cudaFree(a_d);
}
```

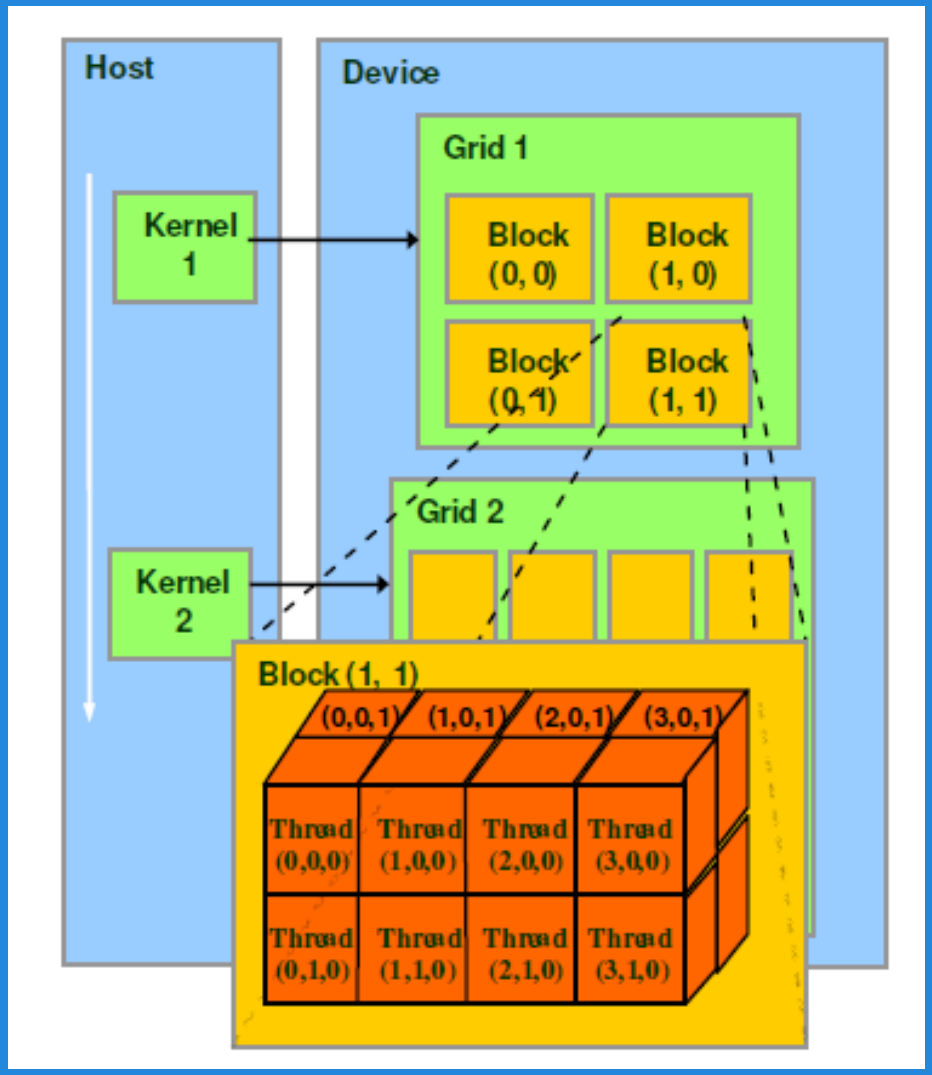
><><><><><><><><><><><><><><><><><><><><

- The cudaMalloc/Memcpy/Free work like you'd expect, except they work on device memory, more on this later
- You see the C-style function launch with arguments as usual, but sandwiched in between are angle brackets?!
- These determine the structure of your threads
- This structure can be important from a performance perspective
- Also some more arguments (optional)
  - Shared memory allocations, more on this later

More on Grid and Block Dimensions

- Grids and Blocks can be 1D, 2D, or 3D
  - You may want different dimensions depending on the format of your data
  - In this case we've used a 1D grid
- Specify the dimensions of your grid and blocks using int or dim3
- The kernel call is then kernel<<<gridDim, BlockDim>>>(args)
- You can access the block index within the grid with blockIdx, the block dimensions with blockDim, and the thread index in the block with threadIdx





Example 2D and 3D grids

A Sample 2D Kernel

```
__global__ void invert(float *image) {
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int idx = y * blockDim.x * gridDim.x + x;
    image[idx] = 1.0 - image[idx];
}

void launch_kernel(float *image)
{
    // ... move host memory to device ... //
    int threadsPerBlock = 16;
    dim3 blocks(512 / threadsPerBlock, 512 / threadsPerBlock);
    dim3 threads(threadsPerBlock, threadsPerBlock);
    invert<<<blocks, threads>>>(d_image);

}
```

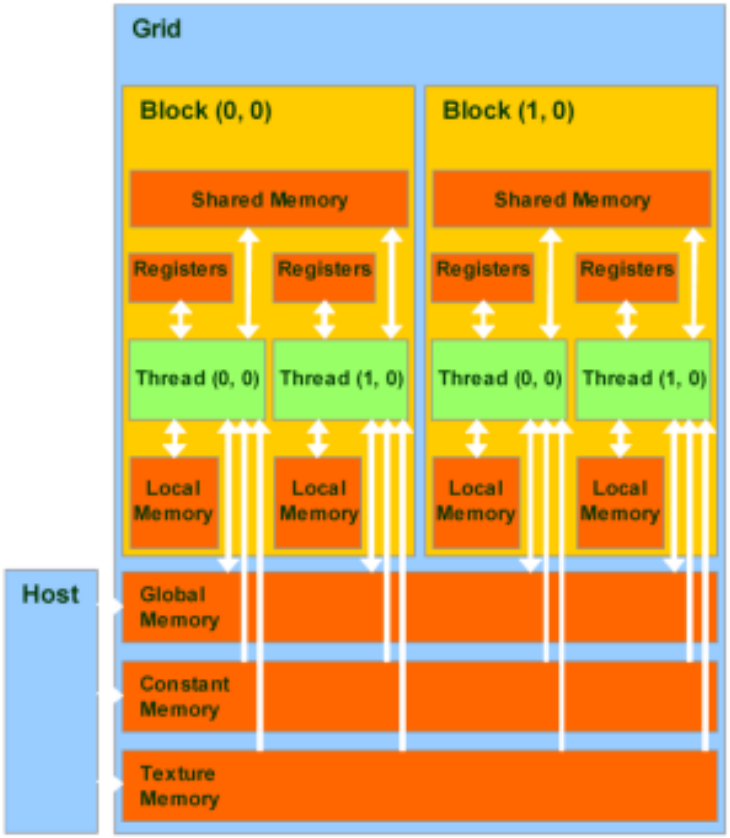
Occupancy

- So how many threads/blocks do you want?
- Number of threads per block should be a multiple of 32 because of the architecture
  - How many depends on how much sharing needs to happen between threads in a block
- Total number of blocks depends on size of block and the size of your data
- You can use cudaGetDeviceCount() and cudaGetDeviceProperties() to learn more about the specifics of your system
  - If you really want to optimize, check out the occupancy calculator at http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

Memory

Memory Types

- Global memory (read and write)
  - Slow, but has cache
- Texture memory (read only)
  - Cache optimized for 2D access pattern
- Constant memory
  - Slow but with cache
- Shared memory (~48kb per MP)
  - Fast, but kind of special
- Local Memory



Device / Global Memory

- CUDA kernels can only operate on device memory, not host memory
  - So we need to copy over relevant data from the host to device
- 2 general types: linear memory and CUDA arrays
  - We will focus on linear memory for now
  - Linear memory works like normal C memory that you're used to

Linear Memory

- Use cudaMalloc and cudaFree to allocate and free linear device memory
- Use cudaMemcpy to move data between host and device memory
- Device memory is in a 32 bit address space for compute level 1.x cards and 40 bit space for 2.x cards (we are 2.1)
  - Thus you can refer to memory via pointers as you usually would

Global Memory

- You can also allocate 2D and 3D memory using cudaMallocPitch and cudaMalloc3D
  - You will want to use these if you can because they are properly optimized and padded for performance
  - It might make sense to also use 2D and 3D thread blocks to operate on such a memory arrangement
- You can also malloc and free inside your kernels, and such allocations will persist across different kernels

CUDA Atomics

- Note that thread blocks are not independent, and their scheduling order is not guaranteed
  - Clearly, threads can access and write to any location in global memory
- CUDA provides atomic functions to safely update the same data across multiple threads
  - ie. atomicAdd, atomicMin, ...

Constant Memory

- Constant memory has some optimizations
  - Use the __constant__ keyword, optionally with __device__
  - However, with computer 2.0 and later, if you malloc as usual and pass data to your kernel as a const pointer, it will do the same thing
- Local memory is local to threads like registers, but it's actually as slow as global memory

Shared Memory

- So far, we have only talked about global memory, accessible across all threads
  - This is fine, but it is the slowest memory available on the GPU
- For speed, you'll want to make use of shared memory
  - Shared memory is private to a single thread block, but can be accessed by all threads in the block
  - Many times faster than global memory
  - The amount of shared memory must be determinable at kernel launch time
  - Shared memory has a lifetime of the kernel block

Shared Memory

- Since shared memory is private per thread block it's useful for communicating data between threads in a block
- To synchronize threads across a block, use __syncthreads();
  - Note that this does not synchronize all threads globally, but only threads within that block
  - Useful for reading / writing shared memory

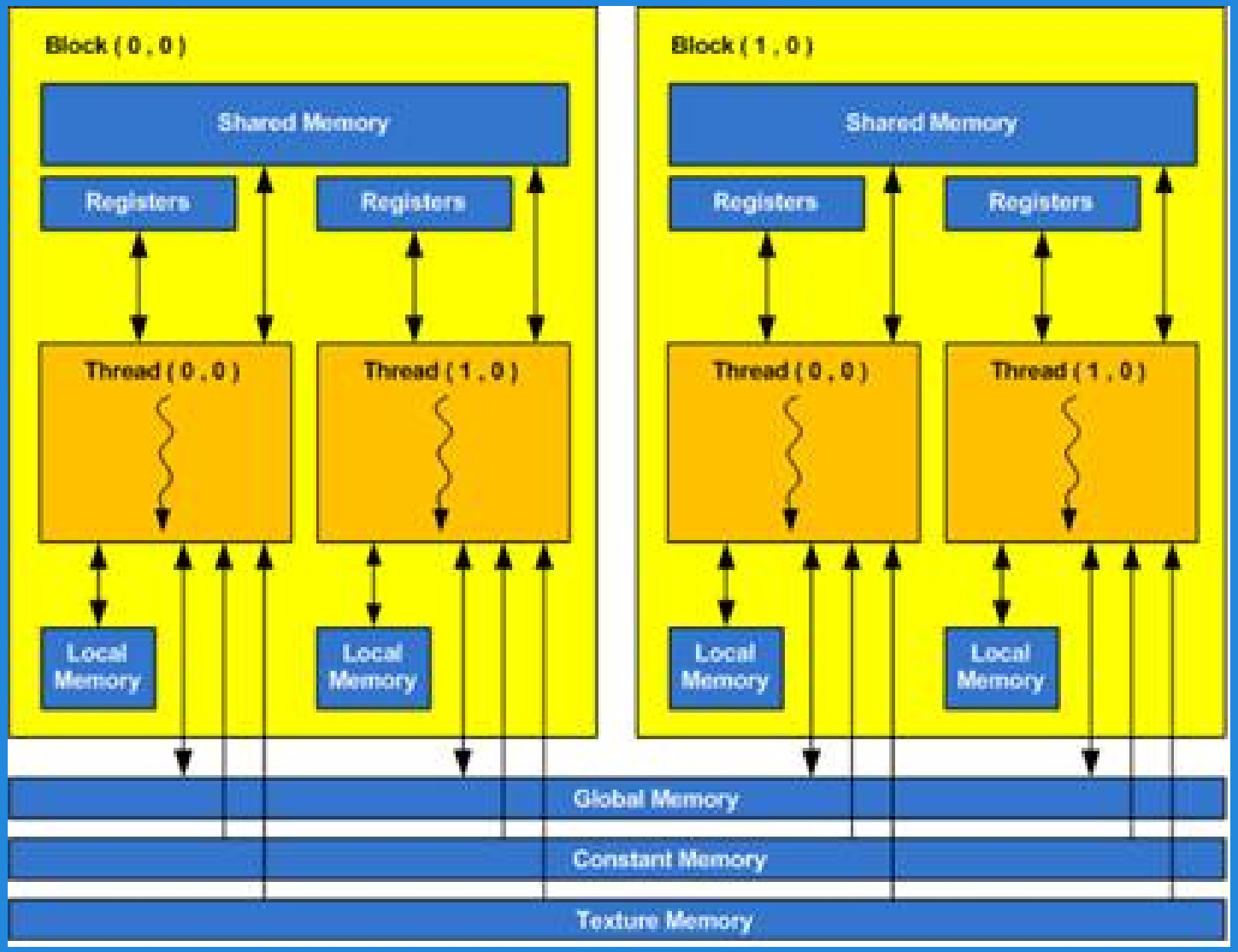
__syncthreads()

- Useful, but be careful when using __syncthreads()!
- The specification states that no thread will advance to the next instruction until every thread in the block reaches __syncthreads()
- Recall that in the case of a branch, the GPU feeds threads through one condition while the others wait - then the remaining threads complete the other branch
  - What happens if the __syncthreads() lies in a divergent branch?

Divergent Branch with Sync

```
__global__ void vec_dot(float *a, float () {
    __shared__ float cache[threadsPerBlock];

    int cacheIndex = threadIdx.x;
    cache[cacheIndex] = a[threadIdx.x + blockIdx.x * blockDim.x];
    int i = blockDim.x / 2;
    while(i != 0) {
        if (cacheIndex < i) {
            cache[cacheIndex] += cache[cacheIndex + i];
            __syncthreads();
        }
        i /=2 ;
    }
}
```

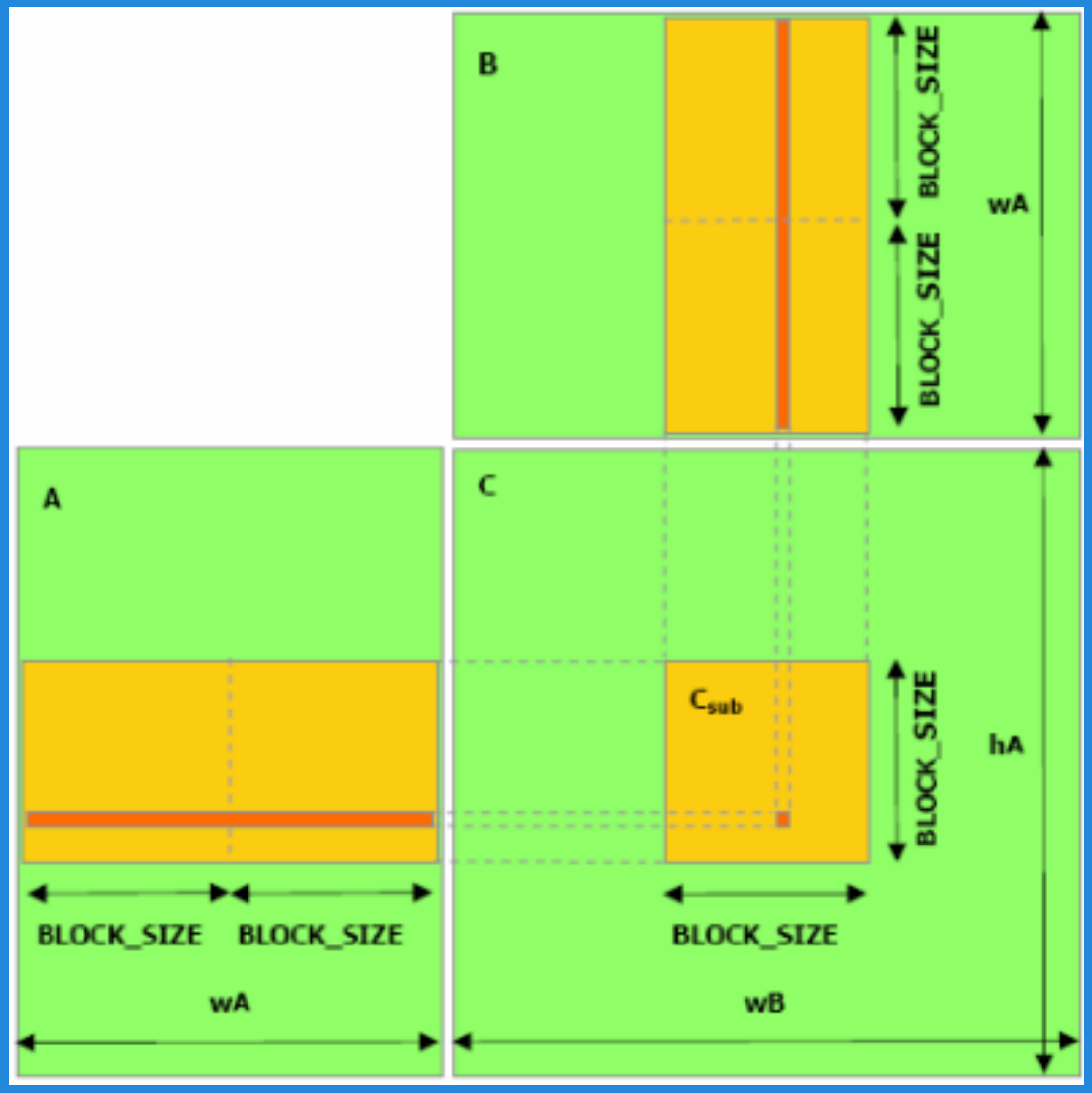
- The above code will cause the GPU to stall indefinitely
- Normally, divergent branches result in some idling threads, but in the case of __syncthreads(), the results are somewhat tragic
  - Since all threads within a block must reach __syncthreads() before continuing, the GPU ends up waiting forever



CUDA memory arrangement

Shared Memory Example

- Matrix multiplication oh boy
- Split it up into smaller matrices, and assign a block to each section
  - For each block, copy the parts of the multiplicands that you are interested in into shared memory, then use that for your computations
- Will probably have to draw this out...



Matrix multiplication

Matrix Multiply Example

```
for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
    // Get sub-matrix Asub of A
    Matrix Asub = GetSubMatrix(A, blockRow, m);
    // Get sub-matrix Bsub of B
    Matrix Bsub = GetSubMatrix(B, m, blockCol);
    // Shared memory used to store Asub and Bsub
respectively
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    // Load Asub and Bsub from device memory to
shared memory
    // Each thread loads one element of each sub-
matrix
    As[row][col] = GetElement(Asub, row, col);
    Bs[row][col] = GetElement(Bsub, row, col);
    // Synchronize to make sure the sub-matrices
are loaded
    // before starting the computation
    __syncthreads();
    // Multiply Asub and Bsub together
    for (int e = 0; e < BLOCK_SIZE; ++e)
         Cvalue += As[row][e] * Bs[e][col];
     // Synchronize to make sure that the
preceding
    // computation is done before loading two
new
    // sub-matrices of A and B in the next
iteration
            __syncthreads();
}
SetElement(Csub, row, col, Cvalue);
```

Allocating Shared Memory

- You can add a parameter to the kernel launch <<<gridDim, blockDim, sharedBytes>>>
  - Allocates a certain number of shared bytes that you can access with something like
    extern __shared__ float data[]
  - You can only do it once, so if you want multiple shared items dynamically allocated, you have to do
    extern __shared__ float data[]
    float *d1 = data;
    int *d2 = &data[32]; // if you want d1 to be 32 bytes
    double *d3 = &data[64]; // if d2 is 32 bytes
  - Watch out for memory alignment!

Compile time shared memory

- Alternatively, you can allocate your shared memory in the kernel
  - __shared__ float data [DATA_SIZE]
  - But data size needs to be known at compile time (I think)
  - Useful if amount of shared memory required by a block is same for all blocks, like in the matrix multiply example

CUDA Boilerplate & Utility

CUDA gdb

- CUDA gdb is installed for debugging CUDA programs
  - (check the toolkit)
- Allows for realtime debugging of a CUDA application on GPU hardware (should be very similar to GDB)
- See the user manual for instructions
  - http://developer.download.nvidia.com/compute/cuda/2_1/cudagdb/CUDA_GDB_User_Manual.pdf

CUDA Init

- Including cudart.h provides you with simple device functionality checks
- Good practice to use CUDA_INIT(argc, argv) at the beginning of your program to check the device
- Use CUDA_EXIT(argc, argv) when you're done

Error Checking

- The cutil.h header provides several utility functions for error checking
- It's good practice to wrap your cudaMalloc / cudaMemcpy / other calls with cudaSafeCall ([function])
  - Checks if an error occurs when calling that function

Error checking

- In your C/C++ code, you can use cudaGetLastError and cudaPeekLastError to get error data after any synchronous CUDA call
  - For asynchronous calls like kernel launches, call cudaThreadSynchronize and then check for errors
  - cudaThreadSynchronize blocks until device has completed all calls including kernel calls, and returns an error if something fails
- Use cudaGetErrorString to translate error into something readable

No-copy pinning of Memory

- In all the previous examples, we have used copy pinning to move memory from host to device
- In CUDA 4, you can map host memory to the device to avoid a copy with the drawback of using higher latency memory

```
float *a = (float *)malloc(sizeof(float) * 64), *d_a;
cudaHostRegister(a, sizeof(float) * 64,
cudaHostRegisterMapped);
cudaHostGetDevicePointer((void **) &d_a, (void *)a, 0);
mykernel<<<32,32>>>(d_a,  64);
cudaHostUnregister(a);
free(a);
```

Justin's notes

grids, blocks, threads

blocks must be independent but threads do not, they can synchronize and share memory

threads have local memory, blocks have shared memory, grid has global memory

also constant, texture memory

there is a host (CPU) and device (GPU), assumes that threads execute separately on these devices

CUDA code compiles down to PTX assembly (you can also write directly in PTX, but we won't)

nvcc compiler turns ptx into binary code

-arch=sm_10 means compute capability 1.0, we have 2.1 for fermi

you can use __CUDA_ARCH__ macro to change between different code paths depending on your architecture

use __global__ for your kernel, and __device__ for your subroutines

kernels can only operate on device memory aka gpu memory, so you have to allocate it for the kernels, either as linear memory or cuda arrays

cuda arrays are for texture stuff, which we won't use much

linear memory is in a 32 bit space for compute 1.x and 40 bit for computer 2.x, so separately allocated memory can still reference each other via pointers

use cudamalloc and cudafree to make/destroy linear memory, and cudamemcpy to transfer between host and device memory

you can also use cudamallocpitch or cudamalloc3d for 2d or 3d arrays, recommended for performance because it appropriately pads it

memory allocated with these is considered global, you can create __shared__ memory inside your kernels

there are more memory types like page-locked, portable, write-combined, mapped, but we won't go into these

using SLI requires some additional stuff like transferring between devices but we won't go into it

More notes

use cudagetlasterror or cudapeeklasterror to get errors, for asynchronous calls (like kernel launches) call cudaDeviceSynchronize first

there is direct interoperability with gl and dx where you can map resources directly to cuda so that it can read or modify

you can malloc and free inside your kernels, and allocated memory will persist across different kernels

in general, you want multiple of 32 threads per block, shared memory can play into this, and the number of blocks depends on the size of your problem (for scheduling)

see the occupancy spreadsheet if you want to optimize more?