The CUDA Memory Model
Advanced Computer Architecture
COMP 140
Tuesday June 17, 2014

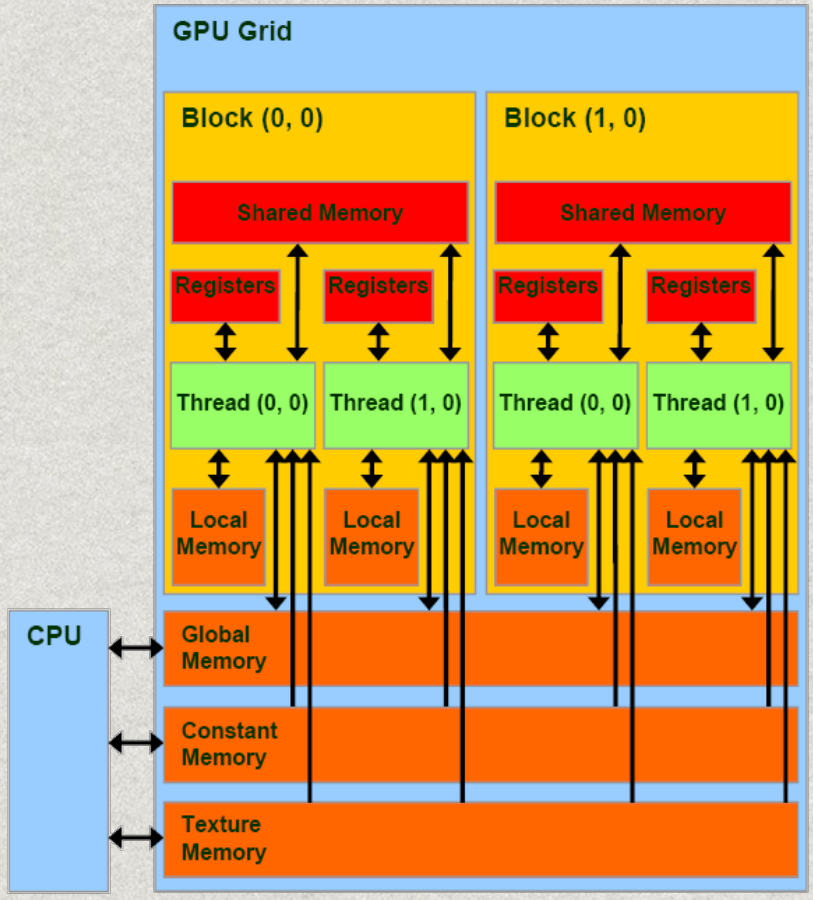
CUDA processors have multiple types of memory available to the programmer, and to each thread.

Global: large address space, high latency (100x slower than cache)

Shared: small, low latency

Texture/Constant: read-only

Registers/Local:only available to one thread



Register memory

Scalar variables (e.g., `int i`, `float f`, etc.) are stored in fast registers. There are a limited number of registers per thread (although each SIMD processor has 32,768 32-bit registers!)

Kernel-declared arrays can also be in registers, but only if the indexing is known at compile-time.

Registers are private to the thread: no sharing

Read/Write, no synchronization necessary

Local Memory

Registers can spill into “local memory,” which is just (slow) global memory set aside for each thread. New GPUs do have caches for local memory.

The only way to know if your registers have been spilled is to look at the PTX code for the “.local” mnemonic.

Read/Write, no synchronization necessary

Shared Memory

Variables declared with __shared__ are stored in shared memory, which is very fast.

It is, however, limited (48KB per multiprocessor on our GPU).

Has the lifetime of a block — can be shared between threads in a block. Cannot be shared between blocks.

Read/Write, must be synchronized with __syncthreads().

Good practice: copy global to shared for best use (but only if you can reduce global memory usage)

Global Memory

Variables declared with __device__ are stored in global memory, which is very slow.

We have lots of global memory, though (up to 2GB on our GPU).

`cudaMemcpy` reads and writes from the CPU to GPU memory.

Lifetime of the program, but sometimes tricky because you can't synchronize across blocks. If you need synchronization, you must have multiple kernel invocations.

Global Memory (continued)

Global memory is declared on the host process using **`cudaMalloc`** and freed in the host process using **`cudaFree`**. Pointers are passed from the CPU to the GPU.

Reducing global accesses is a goal, but an art form. Judicious use of shared memory is helpful.

Constant Memory

Variables that are declared with the **__constant__** attribute are declared in constant memory (which is part of global memory).

There is only a limited amount of constant memory (64KB per kernel), but it is much faster than regular global memory, because it is always cached.

Constant memory can be written to by the host process using the **`cudaMemcpyToSymbol`** function and read-from using the cudaMemcpyFromSymbol function.

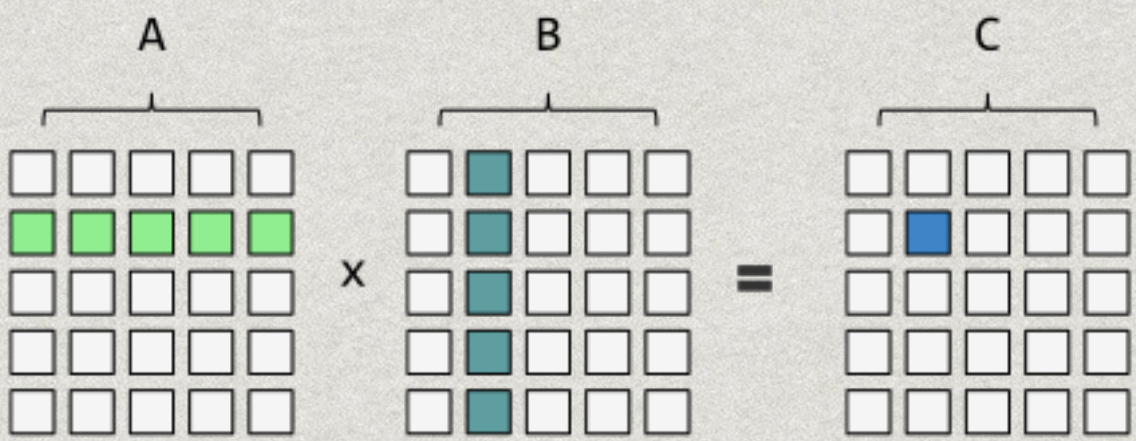
Lifetime of program, but can only be changed by the CPU.

Properties of Memory Types Overview

Memory	Located	Cached	Access	Scope	Lifetime
Register	cache	n/a	Host: None Kernel: R/W	thread	thread
Local	device	Yes	Host: None Kernel: R/W	thread	thread
Shared	cache	n/a	Host: None Kernel: R/W	block	block
Global	device	Yes	Host: R/W Kernel: R/W	application	application
Constant	device	Yes	Host: R/W Kernel: R	application	application

Looking at variations of Matrix Multiply

We will look at a few different ways to perform a matrix multiply using CUDA. We want to show how minimizing global memory accesses can greatly speed up the algorithm.

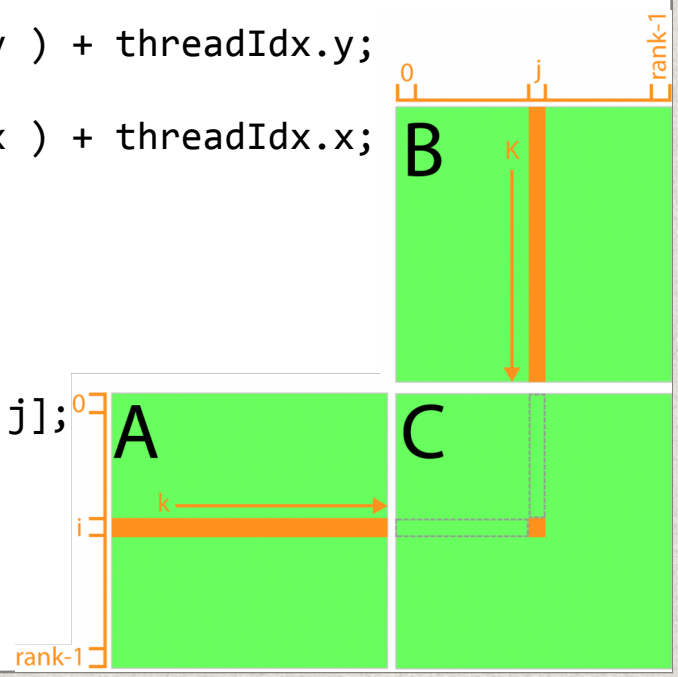



```
C[i][j] = sum(A[i][k] * B[k][j]) for k = 0 ... n
```

Matrix Multiply: naïve approach

```
// MatrixMultiply.cu (just kernel)
__global__ void MatrixMultiplyKernel_GlobalMem( float* C, const float* A,
const float* B, unsigned int rank )
{
    // Compute the row index
    unsigned int i = ( blockDim.y * blockIdx.y ) + threadIdx.y;
    // Compute the column index
    unsigned int j = ( blockDim.x * blockIdx.x ) + threadIdx.x;

    unsigned int index = ( i * rank ) + j;
    float sum = 0.0f;
    for ( unsigned int k = 0; k < rank; ++k )
    {
        sum += A[i * rank + k] * B[k * rank + j];
    }
    C[index] = sum;
}
```



Matrix Multiply: naïve approach

We would like to reduce the number of times the elements of matrix **A** and **B** are accessed to just 1.

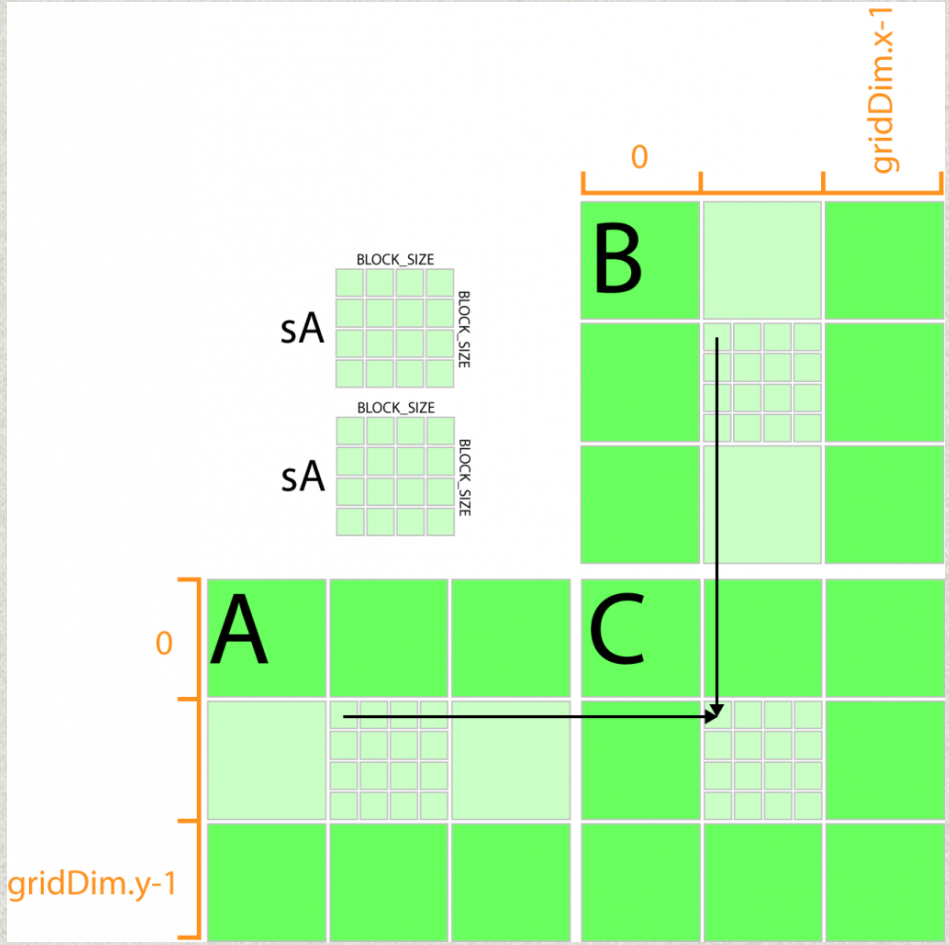
Unfortunately, we can't just store **A** and **B** into shared memory, but there isn't enough shared memory for that.

We have to figure out a way to break the problem down — we will use a method called "tiling" to do this.

Tiling is the same as partitioning grids!

We will split our grids into 16x16 threads.

Matrix Multiply: breaking into grids



Each thread block defines a pair of shared memory buffers that are used to “cache” a “tile” of data from matrix **A** and matrix **B**. Since the “tile” is the same size as the thread block, we can just let each thread in the thread block load a single element from matrix **A** into one of the shared memory buffers and a single element from matrix **B** into the other.

Matrix Multiply: Tiled MatrixMultiply.cu

```
#define BLOCK_SIZE 16

__global__ void MatrixMultiplyKernel_SharedMem( float* C, const float* A, const float* B, unsigned int rank )
{
    unsigned int tx = threadIdx.x;
    unsigned int ty = threadIdx.y;
    unsigned int bx = blockIdx.x;
    unsigned int by = blockIdx.y;

    // Allocate share memory to store the matrix data in tiles
    __shared__ float sA[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float sB[BLOCK_SIZE][BLOCK_SIZE];

    // Compute the column index
    unsigned int j = ( blockDim.x * bx ) + tx;
    // Compute the row index
    unsigned int i = ( blockDim.y * by ) + ty;

    unsigned int index = ( i * rank ) + j;
    float sum = 0.0f;

    // Loop through the tiles of the input matrices
    // in separate phases of size BLOCK_SIZE
    for( unsigned int phase = 0; phase < rank/BLOCK_SIZE; ++phase )
    {
        // Allow each thread in the block to populate the shared memory
        sA[ty][tx] = A[i * rank + (phase * BLOCK_SIZE + tx)];
        sB[ty][tx] = B[(phase * BLOCK_SIZE + ty) * rank + j];
        __syncthreads();

        for( unsigned int k = 0; k < BLOCK_SIZE; ++k )
        {
            sum += sA[ty][k] * sB[k][tx];
        }
        __syncthreads();
    }

    C[index] = sum;
}
```

See handout

References

- 3D Game Engine Programming, CUDA Tutorial: http://3dgep.com/?p=2012, accessed June 17, 2014
- NVIDIA Corporation (2011, May). NVIDIA CUDA C Programming Guide. (Version 4.0). Santa Clara, CA 95050, USA Available from: http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf. Accessed: 15 November 2011.
- NVIDIA Corporation (2009, January). NVIDIA's Next Generation CUDA Compute Architecture: Fermi. (V1.1). Santa Clara, CA 95050, USA Available from: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Accessed: 15 November 2011.