CMSC 691
High Performance Distributed Systems

Introduction to CUDA III

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu

Using shared memory to avoid non-coalesced accesses:

- Non-coalesced global memory accesses cause larger number of smaller memory transactions
- Shared memory provides low latency accesses

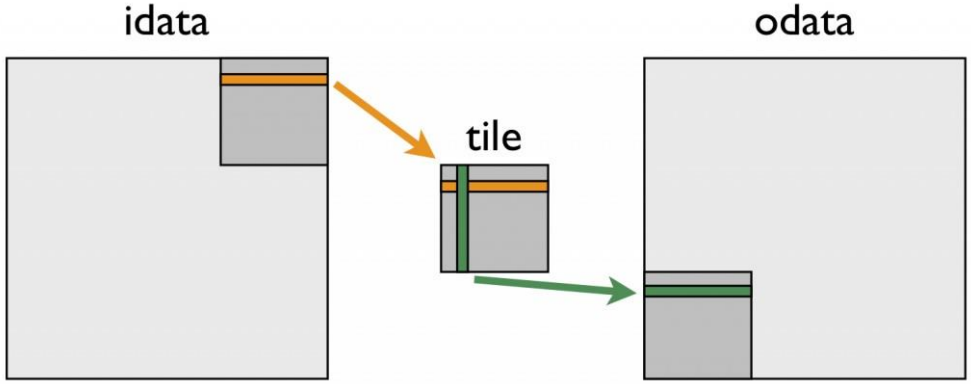
- Idea to avoid non-coalesced pattern:
  1. Load from global memory with stride 1
  2. Store into shared memory with stride x (no problem!)
  3. __syncthreads()
  4. Load from shared memory with stride y (no problem!)
  5. Store to global memory with stride 1

The naïve matrix transpose

- As many threads as the number of elements in the matrix
- Not best performance due to non-coalesced memory accesses
- Stride = matrix width!

Tiled matrix transpose

- Uses shared memory to load and store tiles
- Breaking into tiles size TILE_WIDTH, e.g. 16x16



The naïve matrix multiplication

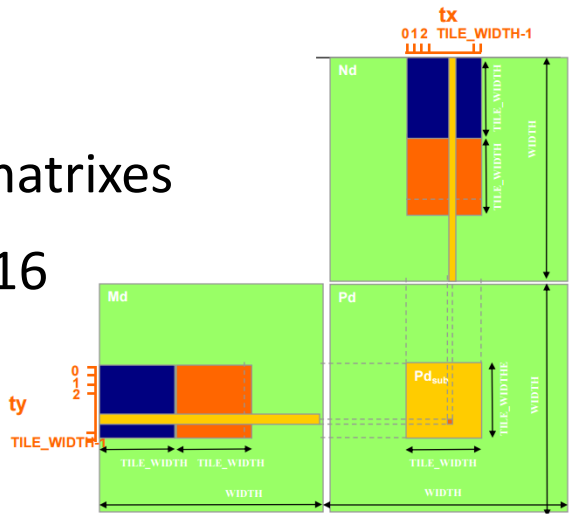
- As many threads as the number of elements in the matrix
- 8192 x 8192 multiplication:

  5.5 s in GPU (67 million threads!)

  1h 40 mins in CPU, this is 1000x speedup!!
- Not best performance due to non-coalesced memory accesses

Tiled matrix multiplication

- Uses shared memory to load and reuse submatrixes
- Breaking into tiles size TILE_WIDTH, e.g. 16x16



Tiled matrix multiplication

```
__global__ void matrixMulTiled(float *A, float *B, float *C, int width)
{
    int column = ( blockDim.x * blockIdx.x ) + threadIdx.x;
    int row    = ( blockDim.y * blockIdx.y ) + threadIdx.y;

    float sum = 0;

    // Loop over the A and B tiles required to compute the submatrix
    for (int t = 0; t < width/TILE_WIDTH; t++)
    {
        __shared__ float sub_A[TILE_WIDTH][TILE_WIDTH];
        __shared__ float sub_B[TILE_WIDTH][TILE_WIDTH];

        // Coolaborative loading of A and B tiles into shared memory
        sub_A[threadIdx.y][threadIdx.x] = A[row*width + (t*TILE_WIDTH + threadIdx.x)];
        sub_B[threadIdx.y][threadIdx.x] = B[column + (t*TILE_WIDTH + threadIdx.y)*width];

        __syncthreads();

        // Loop within shared memory
        for (int k = 0; k < TILE_WIDTH; k++)
          sum += sub_A[threadIdx.y][k] * sub_B[k][threadIdx.x];

        __syncthreads();
    }

    C[row*width + column] = sum;
}
```

Maximizing the occupancy

- Warp schedulers find a warp that is ready to execute its next instruction and available execution cores and then start execution
- GK110: 4 warp schedulers, 2 dispatchers in each SM
- Starts instructions in up to 4 warps each clock, and starts up to 2 instructions in each warp.
- max threads / SM = 2048 (64 warps)
- max threads / block = 1024 (32 warps)
- 32 bit registers / SM = 64k
- max shared memory / SM = 48KB
- The number of blocks that run concurrently on a SM depends on the resource requirements of the block! Minimize register and shared memory usage

Maximizing the occupancy

- occupancy = warps per SM / max warps per SM
- max warps / SM depends only on GPU
- warps / SM depends on warps / block, registers / block, shared memory / block.

100% occupancy
2 blocks of 1024 threads
32 registers/thread
24KB of shared memory / block

50% occupancy
1 block of 1024 threads
64 registers/thread
48KB of shared memory / block

- Use the CUDA occupancy calculator to maximize the occupancy!

Synchronization

- On a CPU, you can solve synchronization issues using Locks, Semaphores, Condition Variables, etc.
- On a GPU, these solutions would introduce too much memory and process overhead
- All the threads within the warp (32) are implicitly synchronous
- Use the __syncthreads() function to sync threads **within** a block, only works at the block level
- We cannot synchronize all the threads in the grid within the kernel code
- Divide the kernel code into multiple functions, which are called sequentially, kernels in a stream are not overlapped

Atomic operations

- CUDA provides built in atomic operations, but **DO NOT** use them unless strictly necessary. Rewrite the code to avoid them.
- Introduce long overheads due to mechanisms to guarantee the mutual exclusion.
- Use the functions: atomic<opertaor>(float *address, float val);
- Operators: Add, Sub, Exch, Min, Max, Inc, Dec, And, Or, Xor

atomicAdd(float *address, float val) for atomic addition of a given value to a memory address in global memory

Exercise

- Parallel reduction of an array using CUDA

CPU code:

```
float sum = 0.0;
for (int i = 0; i < size; i++)
    sum += array[i];
```

GPU “Code” lol:

```
// assign, allocate, initialize device and host memory pointers
// create threads and assign indices for each thread
// assign each thread a specific region to get a sum over
// wait for all threads to finish running
// combine all thread sums for final solution
```

CMSC 691
High Performance Distributed Systems

Introduction to CUDA III

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu