CMSC 603 High-Performance Distributed Systems

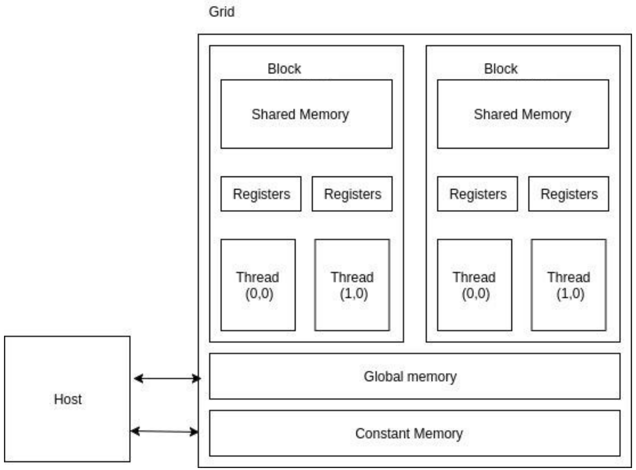
CUDA shared memory

VCU College of Engineering

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

Shared memory

- High-speed on-chip memory per Streaming Multiprocessor (SM)
- Up to 100x faster than global memory
- Small capacity < (96 - 164 KB) per SM
- Shared memory is allocated **per** thread block
- No memory access pattern penalty



- Threads in a block **can** read/write from/to shared memory of the block
- Threads in a block **cannot** read/write from/to shared memory of another block
- Use case: user-managed data cache

Shared memory

- Variables annotated with __*shared*__ are stored in shared memory
- **Static shared memory**: when the size of the memory is known at compile time

```
__global__ void staticReverseArray(int *array, int length)
{
   // Declare static shared memory (size known in compile time). Kernel: 1 block with 256 threads
   __shared__ int sharedMemory[256];

   // Load data from global memory (fully coalesced) into shared memory
   sharedMemory[threadIdx.x] = array[threadIdx.x];

   // Synchronize all threads within the block to make sure all threads loaded respective data
   __syncthreads();

   // Write data to global memory (fully coalesced) from shared memory
   array[threadIdx.x] = sharedMemory[length - threadIdx.x - 1];
}
```

Shared memory

- **Dynamic shared memory**: when the size of the memory is known at run time
- Shared memory allocation per block must be specified in **bytes** in the kernel call

```
dynamicReverseArray <<< 1, length, length*sizeof(int) >>> (d_array, length);

__global__ void dynamicReverseArray(int *array, int length)
{
    // Declare dynamic shared memory (size known in run time)
    extern __shared__ int sharedMemory[];

    // Load data from global memory (fully coalesced) into shared memory
    sharedMemory[threadIdx.x] = array[threadIdx.x];

    // Synchronize all threads within the block to make sure all threads loaded respective data
    __syncthreads();

    // Write data to global memory (fully coalesced) from shared memory
    array[threadIdx.x] = sharedMemory[length - threadIdx.x - 1];
}
```

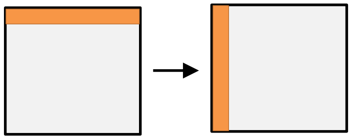
Using shared memory to avoid non-coalesced accesses

- Non-coalesced global memory accesses (read or write) cause a larger number of smaller memory transactions, decreasing the effective bandwidth
- Shared memory among threads in the block provides low latency accesses

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

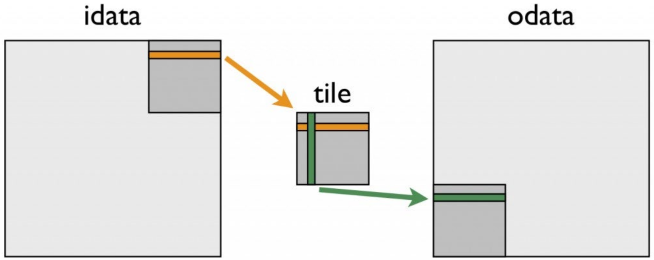
The naïve matrix transpose

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



Tiled matrix transpose

- Uses shared memory to load and store tiles
- Breaking into tiles size e.g. 16x16
- **Example**: matrixTranspose.cu



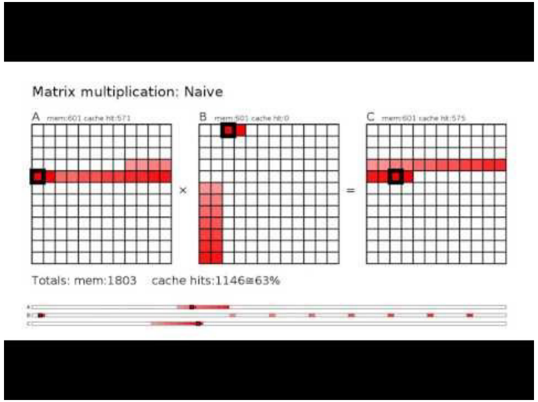
The naïve matrix multiplication

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

  5.5 s in GPU (67 million threads!)

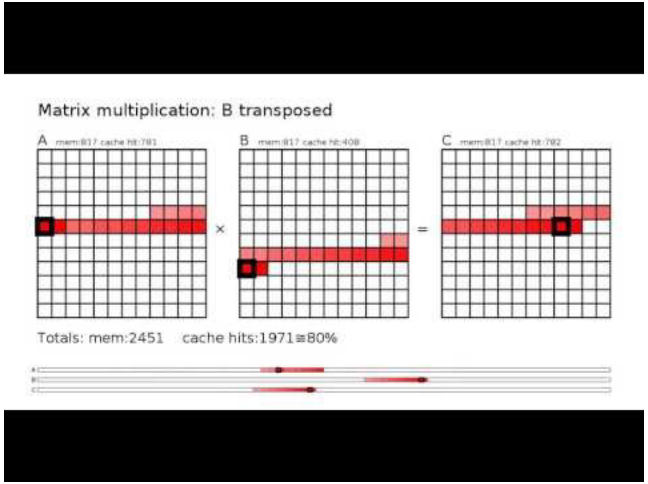
  1h 40 mins in CPU (naïve), this is 1000x speedup!!
- **See**: matrix_multiplication_6_naive.cu
- Not best performance due to non-coalesced memory accesses
- Scatter memory access in matrix B

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

    if (row < width && column < width)
    {
        float sum = 0;

        for(int k = 0; k < width; k++)
            sum += A[row * width + k] * B[k * width + column];

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



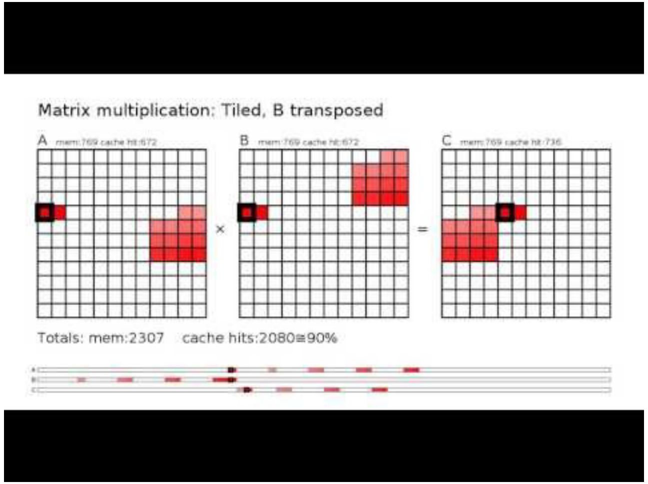
Matrix multiplication

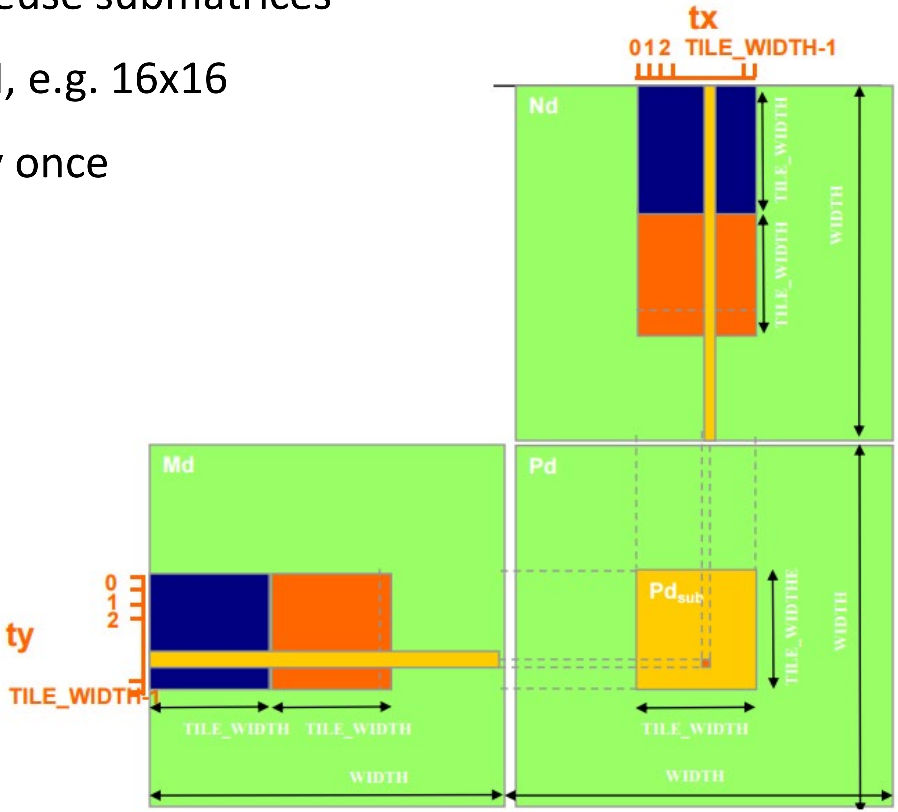
- Transpose matrix B
- Now memory reads from a row in B are fully coalesced
- Requires B to be given in a transposed way, or we must count the time to transpose B in real time



Tiled matrix multiplication

- Uses shared memory to load and reuse submatrices
- Breaking into tiles size TILE_WIDTH, e.g. 16x16
- Multiple global memory reads only once
- Maximum effective bandwidth





Tiled matrix multiplication

```
__global__ void matrixMultiplicationTilesGPU (float *A, float *B, float *C, int matrixSize) {

    int column = blockIdx.x * blockDim.x + threadIdx.x;
    int row    = blockIdx.y * blockDim.y + threadIdx.y;

    __shared__ float tile_A[THREADS_DIM][THREADS_DIM];
    __shared__ float tile_B[THREADS_DIM][THREADS_DIM];

    int tiles_iterations = (matrixSize + THREADS_DIM - 1) / THREADS_DIM;

    if(row < matrixSize && column < matrixSize) {

        float sum = 0;

        for (int tile = 0; tile < tiles_iterations; tile++) {

            // A indexing [1+2+3]
            // 1. How many total elements there are above my row = matrixSize * row;
            // 2. How many total elements there are in the titles left of my current tile = tile * THREADS_DIM
            // 3. How many elements there are in the left within my tile = threadIdx.x
            // Memory access pattern is fully coalesced (consecutive threads in X dimension read consecutive memory positions in global memory)
            tile_A[threadIdx.y][threadIdx.x] = A[matrixSize * row + tile * THREADS_DIM + threadIdx.x];

            // B indexing [1+2+3]
            // 1. How many total elements there are in the tiles above my current tile = tile * THREADS_DIM * matrixSize
            // 2. How many total elements there are above within my tile = threadIdx.y * matrixSize
            // 3. How many total elements there are left of my column = column
            tile_B[threadIdx.y][threadIdx.x] = B[tile * THREADS_DIM * matrixSize + threadIdx.y * matrixSize + column];

            __syncthreads();

            for(int k = 0; k < THREADS_DIM; k++) {
                sum += tile_A[threadIdx.y][k] * tile_B[k][threadIdx.x];
            }

            __syncthreads();
        }

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

- **Example**: matrix_multiplication_9_complete_tiles.cu

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:

// allocate and 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 603
High-Performance Distributed Systems

CUDA shared memory

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

VCU College of Engineering