CMSC 691
High Performance Distributed Systems

Introduction to CUDA II

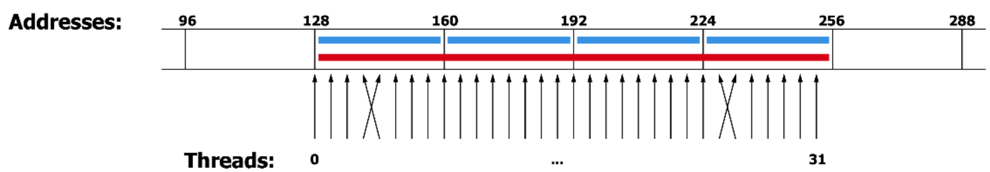
Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu

The importance of the memory hierarchy and access pattern

- Register memory latency: 0 cycles if no RAW dependency
- GPU main memory latency: 300-800 cycles or ~300 ns
- GPU arithmetic instruction latency: ~10 ns
- Having a large number of threads allows hiding the latency of memory accesses
- Access patterns that play nicely with GPU hardware are called coalesced memory accesses
- Memory accesses are done per warp in large groups setup as memory transactions
- Coalesced memory accesses minimize the number of cache lines read in through these memory transactions
- GPU cache lines are 128 bytes and are aligned

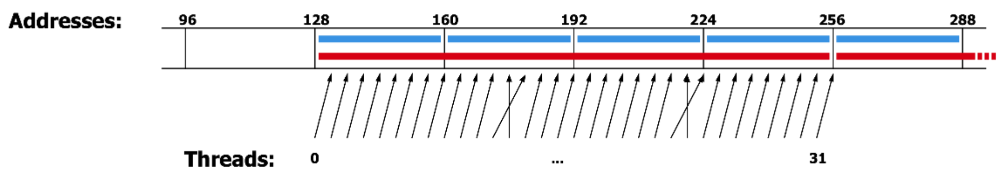
Memory coalescing

Aligned accesses (sequential/non-sequential)



Compute capability:	2.x, 3.x, 5.x	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

Mis-aligned accesses (sequential/non-sequential)

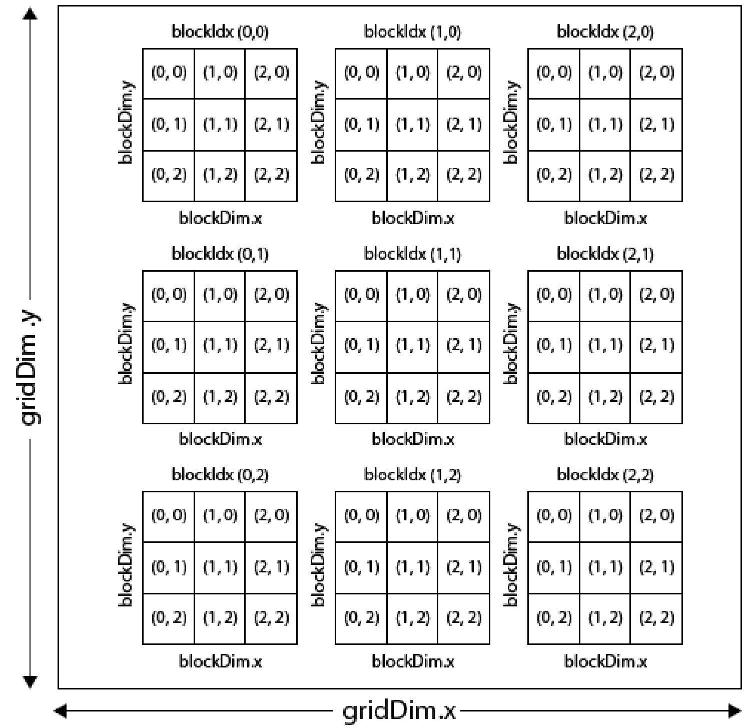


Compute capability:	2.x, 3.x, 5.x	
Memory transactions:	Uncached	Cached
	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

Advanced indexing using the 3D grid of 3D blocks capability

- Example: sum two 2D matrixes

*int column = (blockDim.x * blockIdx.x) + threadIdx.x;*
*int row = (blockDim.y * blockIdx.y) + threadIdx.y;*
*int tid = (blockDim.x * gridDim.x * row) + column;*



Advanced indexing using the 3D grid of 3D blocks capability

- Option 1: define a 2D grid of 2D blocks

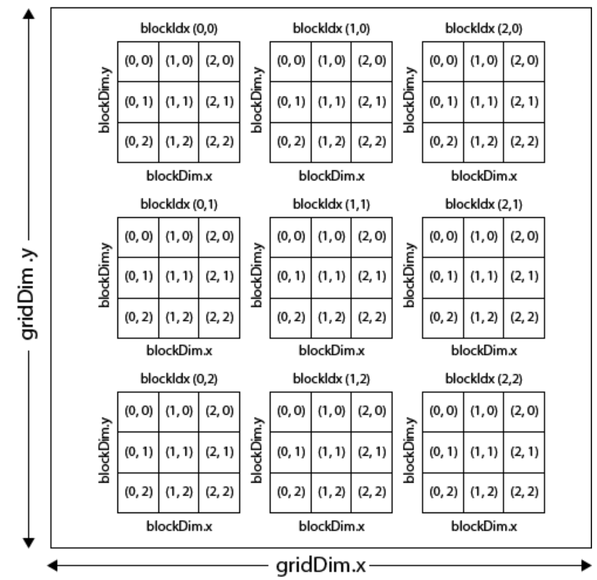
```
int threadsPerBlockDim = 16;
int gridDimSize = (matrixSize + threadsPerBlockDim - 1) / threadsPerBlockDim;

dim3 blockSize(threadsPerBlockDim, threadsPerBlockDim);
dim3 gridSize (gridDimSize, gridDimSize);

matrixAdd<<<gridSize, blockSize>>>(d_A, d_B, d_C, numElements);
```

Advantages

- Easy to understand
- Matched out human-understanding
- Divide into smaller submatrixes

Disadvantages:

- Not optimal memory transfers
- Divisibility of the blocks in X and Y



Advanced indexing using the 3D grid of 3D blocks capability

- Option 2: worst-ever approach, using a 1D grid and 1D-y block

```
int threadsPerBlock = 256;
int gridDim = (numElements + threadsPerBlock - 1) / threadsPerBlock;

dim3 blocksize(1, threadsPerBlock);

matrixAdd<<<gridDim, blocksize>>>(d_A, d_B, d_C, numElements);
```

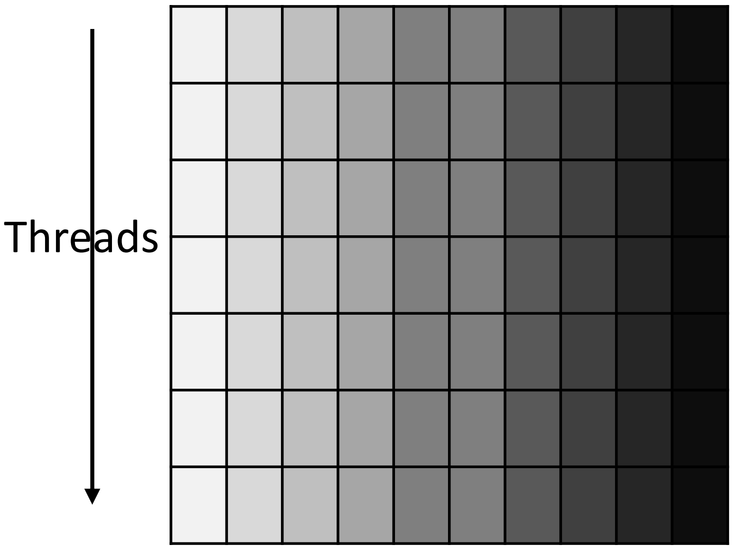
Considering an equal number of threads per block

Advantages

- None?
- Divisibility of the blocks and Y

Disadvantages:

- Terrible memory access pattern!
- Stride = matrix width
- Huge performance bottleneck!!!



Advanced indexing using the 3D grid of 3D blocks capability

- Option 3: best approach, using a 1D grid and 1D-x block

```
int threadsPerBlock = 256;
int gridDim = (numElements + threadsPerBlock - 1) / threadsPerBlock;

matrixAdd<<<gridDim, threadsPerBlock>>>(d_A, d_B, d_C, numElements);
```

Considering an equal number of threads per block

Advantages

- Minimum loss due to divisibility of the blocks 1D only, up to 1 block extra
- Best access pattern, stride = 1, fully coalesced memory
- Matrix width doesn’t affect the performance!

Disadvantages: none! Well … makes you think

Measuring elapsed time: events

- CUDA kernels are non-blocking
- The code will continue until synchronization is forced by
  - cudaDeviceSynchronize()
  - cudaMemcpy()
  - cudaEventSynchronize()

```
float milliseconds;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);

// do something!

cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&milliseconds, start, stop);
printf("GPU time %f ms\n", milliseconds);
```

Shared memory

- Very fast memory located in each multiprocessor, low latency ~ 5ns
- User-configurable size, maximum 48 KB per multiprocessor
- Scope and lifetime of the current block, cannot share data among different blocks
- Can allocate shared memory statically (size known at compile time) or dynamically (size not known until runtime)
- Data races among threads in the block may happen in shared mem
- Programmer may use *__syncthreads()* to synchronize threads in a given block (threads within the block, not all in the grid)

Shared memory declaration:

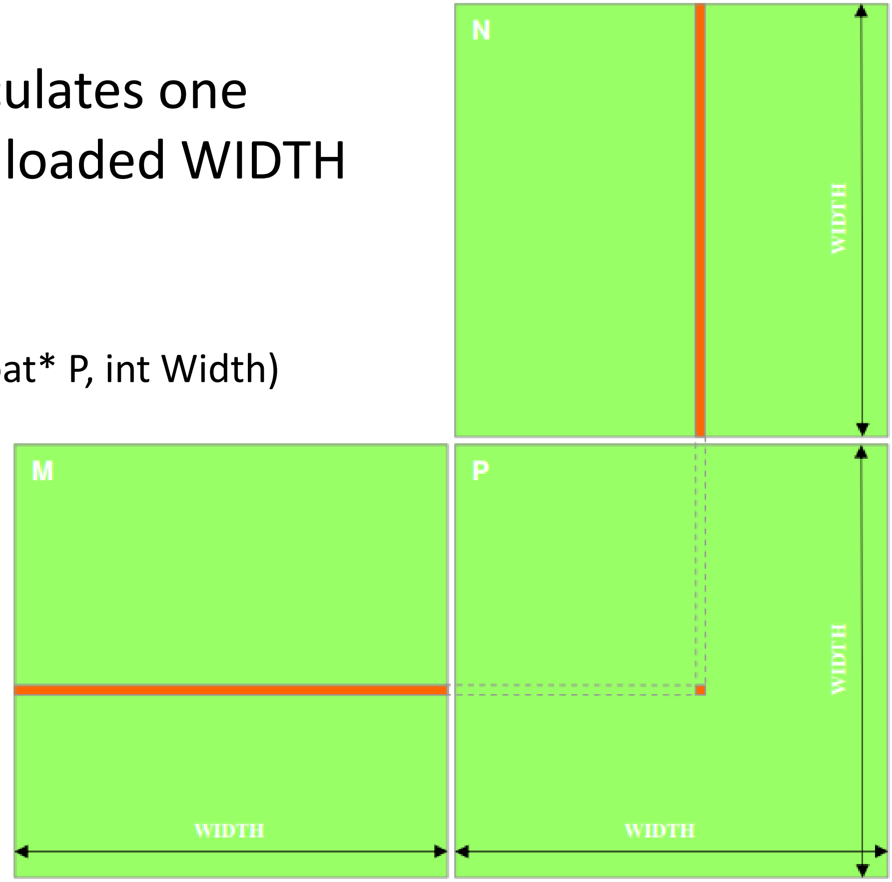
__shared__ float data[1024];

data[tid] = result;

Matrix multiplication

- Classic academic example to show the benefits of shared memory
- P = M * N of size WIDTH x WIDTH
- First approach: one thread calculates one element of P, and M and N are loaded WIDTH times from global memory

```
void MatrixMultiplicationHost(float* M, float* N, float* P, int Width)
{
 for (int i = 0; i < Width; i++)
   for (int j = 0; j < Width; j++) {
     float sum = 0;
     for (int k = 0; k < Width; k++)
       sum += M[i * width + k] + N[k * width + j];
     P[i * Width + j] = sum;
 }
}
```



First GPU approach for matrix multiplication

- One thread computes one element of the result matrix
- Each thread loads a row of M and a column of N
- Consecutive threads likely to access the same row but different columns
- Inner loop iterates for each N row, leading to a bad memory access pattern

```
__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;
    }
}
```

CMSC 691
High Performance Distributed Systems

Introduction to CUDA II

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