

# **CS 380 - GPU and GPGPU Programming**

## **Lecture 14+15: Shading and Compute APIs 5**

Markus Hadwiger, KAUST

# Reading Assignment #8 (until April 9)



Read (required):

- Programming Massively Parallel Processors book, Chapter 5 (*CUDA Memories*)

# Reading Assignment #9 (until April 16)



## Read (required):

- Interpolation for Polygon Texture Mapping and Shading, Paul Heckbert and Henry Moreton

<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.7886>

- MIP-Map Level Selection for Texture Mapping

<http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=765326>

## Read (optional):

- Frame buffer objects extension specification

[http://www.opengl.org/registry/specs/ARB/framebuffer\\_object.txt](http://www.opengl.org/registry/specs/ARB/framebuffer_object.txt)

---

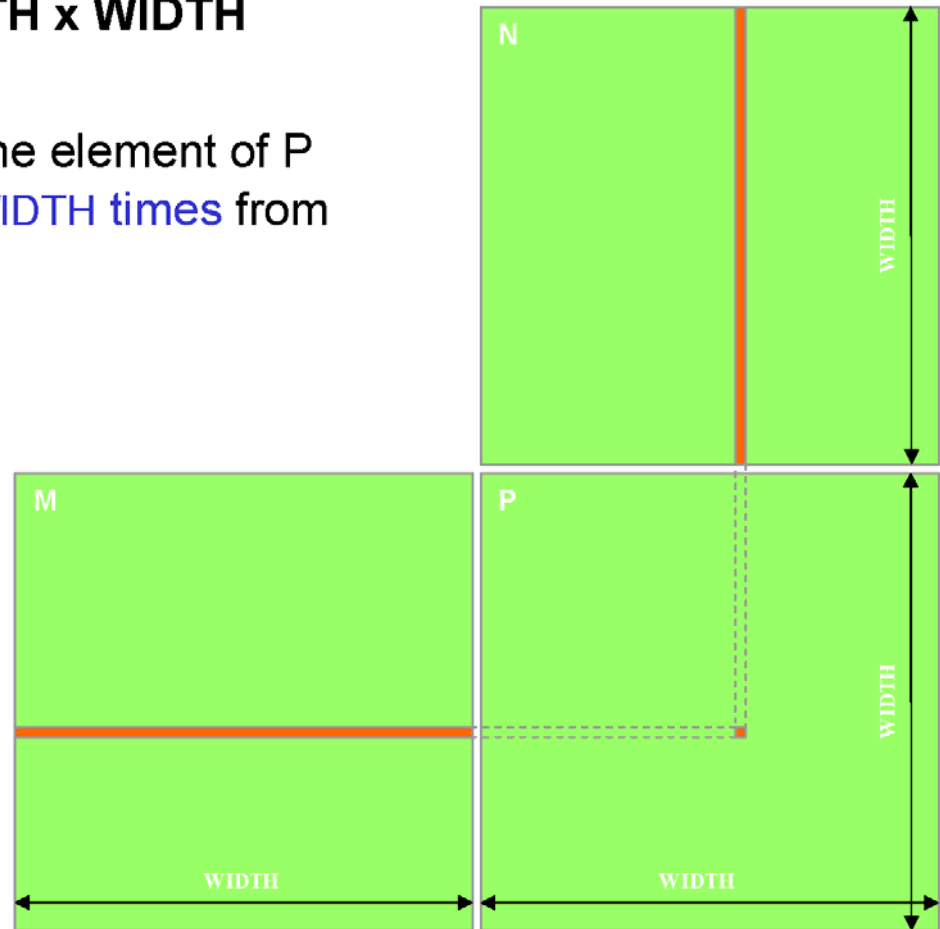
# Matrix-Matrix Multiplication

$$P=MN$$

# Programming Model: Square Matrix Multiplication

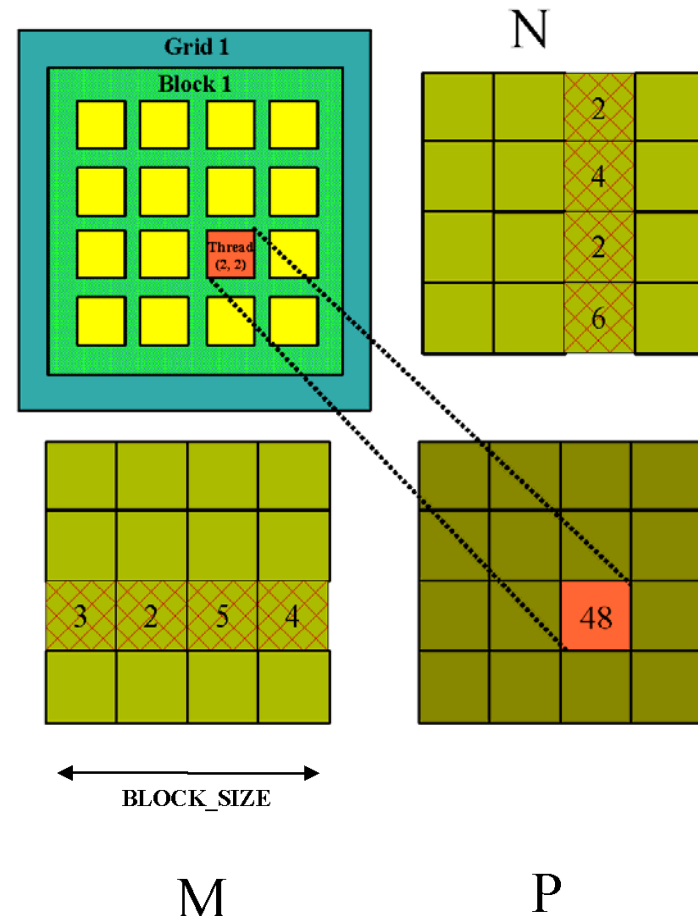
---

- $P = M * N$  of size  $WIDTH \times WIDTH$
- **Without tiling:**
  - One **thread** handles one element of  $P$
  - $M$  and  $N$  are loaded  $WIDTH$  times from global memory



# Multiply Using One Thread Block

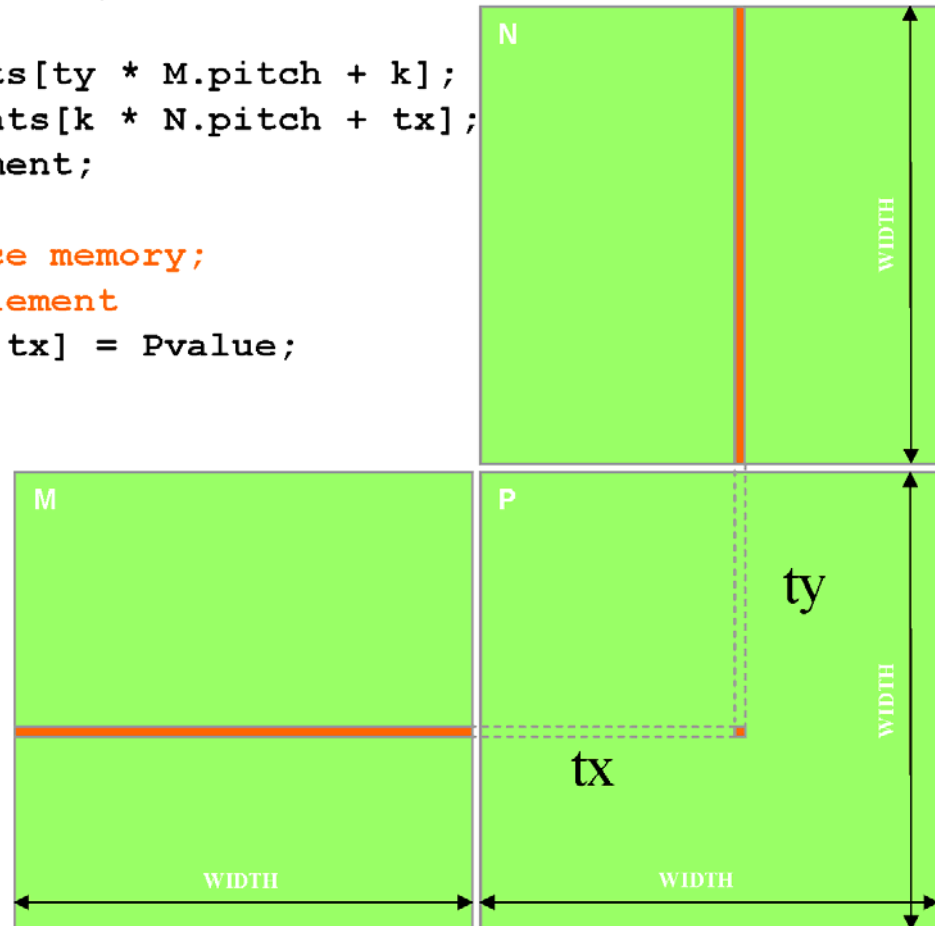
- **One block of threads computes matrix P**
  - Each thread computes one element of P
- **Each thread**
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- **Size of matrix limited by the number of threads allowed in a thread block**



# Matrix Multiplication Device-Side Kernel Function (cont.)

...

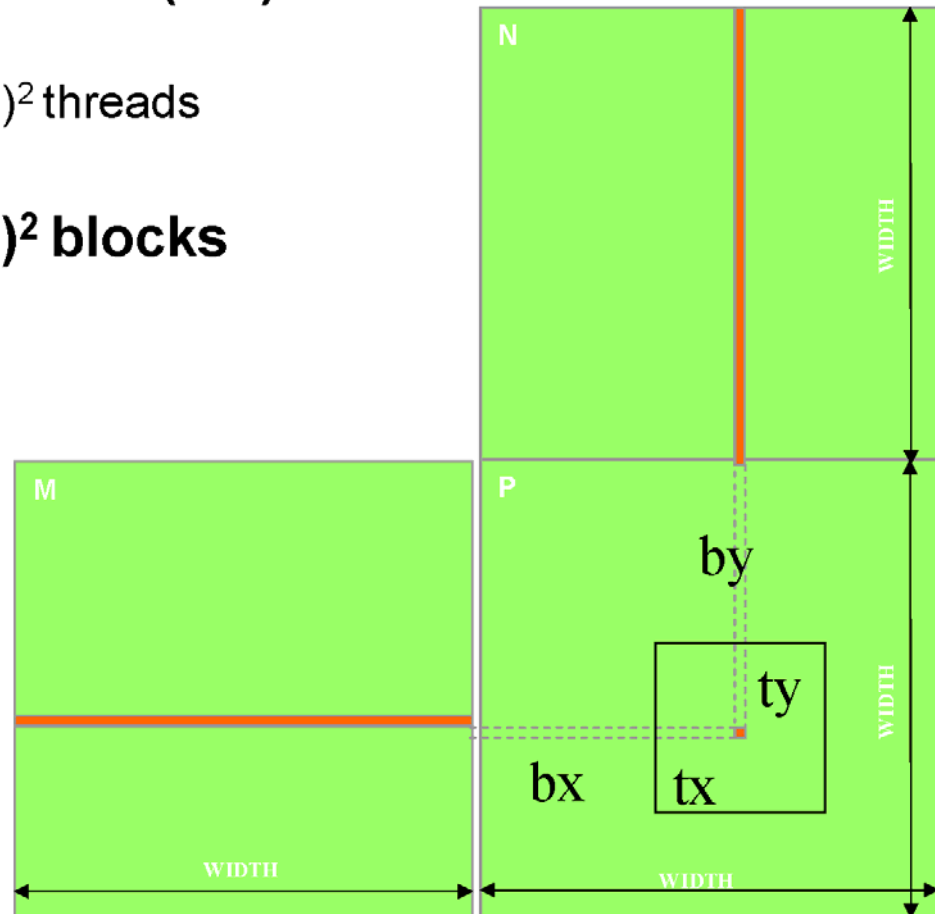
```
for (int k = 0; k < M.width; ++k)
{
    float Melement = M.elements[ty * M.pitch + k];
    float Nelement = Nd.elements[k * N.pitch + tx];
    Pvalue += Melement * Nelement;
}
// Write the matrix to device memory;
// each thread writes one element
P.elements[ty * blockDim.x + tx] = Pvalue;
}
```



# Handling Arbitrary Sized Square Matrices

- Have each 2D thread block to compute a  $(\text{BLOCK\_WIDTH})^2$  sub-matrix (tile) of the result matrix
  - Each has  $(\text{BLOCK\_WIDTH})^2$  threads
- Generate a 2D Grid of  $(\text{WIDTH}/\text{BLOCK\_WIDTH})^2$  blocks

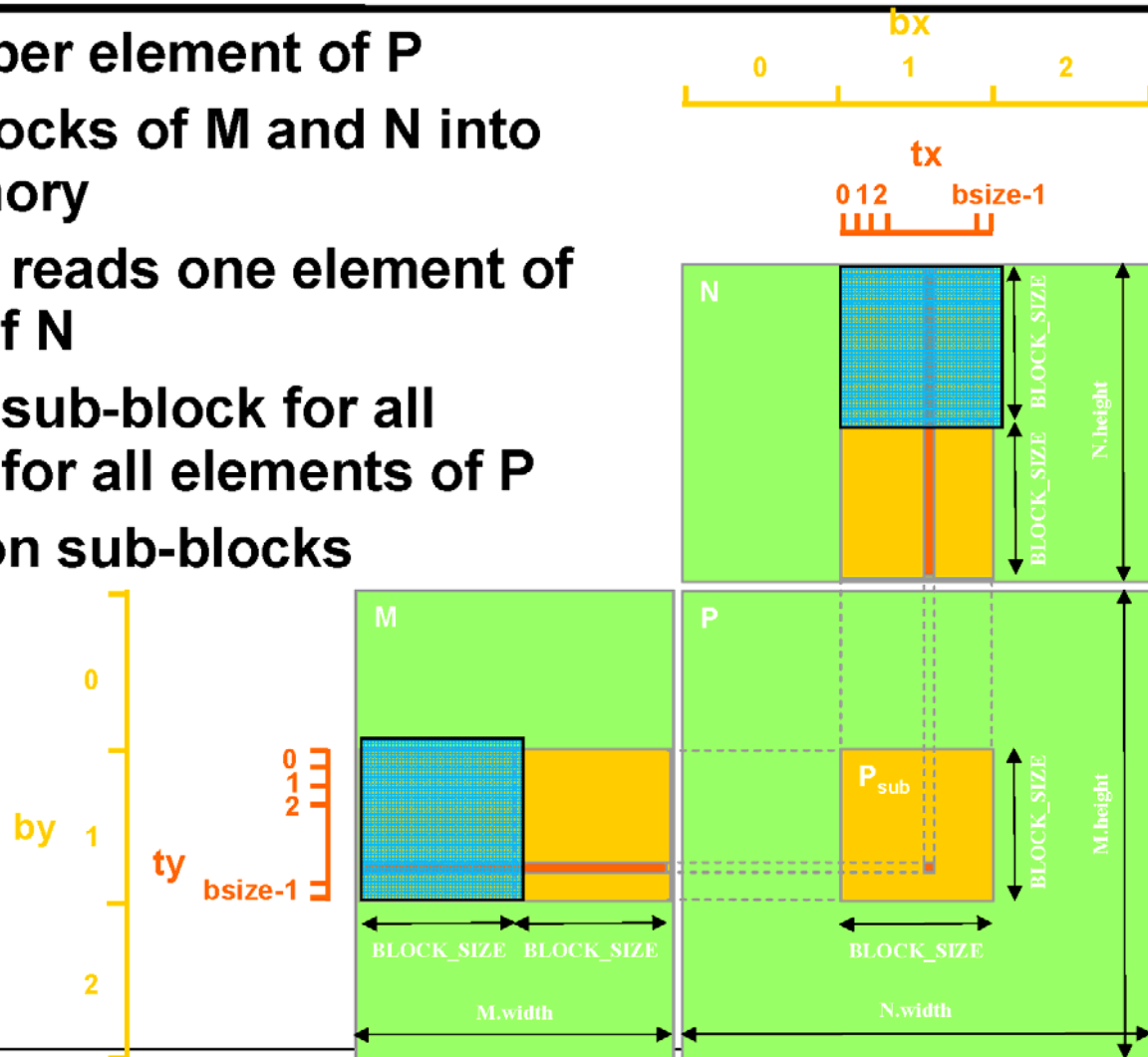
You still need to put a loop around the kernel call for cases where WIDTH is greater than Max grid size!





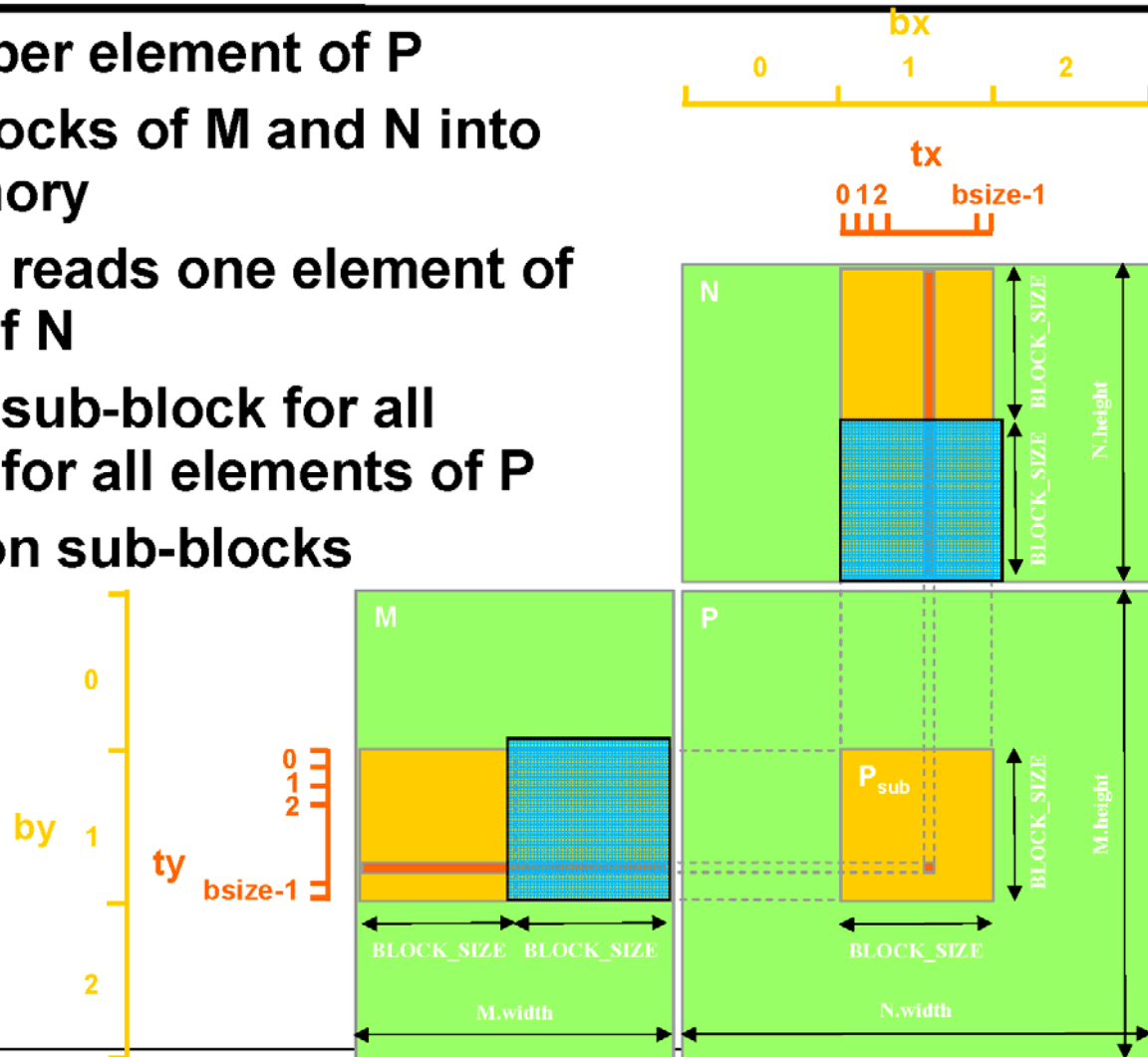
# Multiply Using Several Blocks - Idea

- One thread per element of P
- Load sub-blocks of M and N into shared memory
- Each thread reads one element of M and one of N
- Reuse each sub-block for all threads, i.e. for all elements of P
- Outer loop on sub-blocks



# Multiply Using Several Blocks - Idea

- One thread per element of P
- Load sub-blocks of M and N into shared memory
- Each thread reads one element of M and one of N
- Reuse each sub-block for all threads, i.e. for all elements of P
- Outer loop on sub-blocks



# Example: Matrix Multiplication (1)



- Copy matrices to device; invoke kernel; copy result matrix back to host

```
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc((void**)&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);

    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc((void**)&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);
}
```

## Example: Matrix Multiplication (2)



```
// Allocate C in device memory
Matrix d_C;
d_C.width = d_C.stride = C.width; d_C.height = C.height;
size = C.width * C.height * sizeof(float);
cudaMalloc((void**)&d_C.elements, size);

// Invoke kernel
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

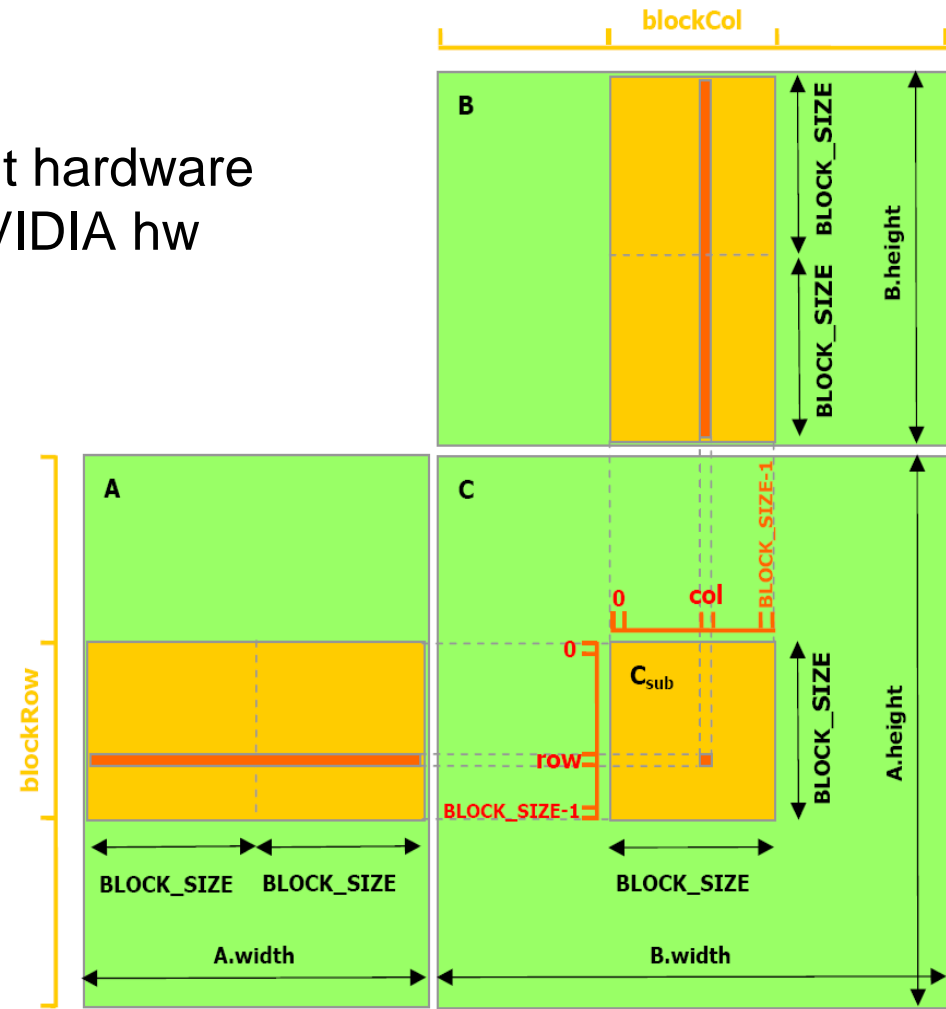
// Read C from device memory
cudaMemcpy(C.elements, d_C.elements, size,
           cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(d_A.elements);
cudaFree(d_B.elements);
cudaFree(d_C.elements);
}
```

# Example: Matrix Multiplication (3)



- Multiply matrix block-wise
- Set BLOCK\_SIZE for efficient hardware use, e.g., to 16 on current NVIDIA hw (or 32 on Fermi)
- Maximize parallelism
  - Launch as many threads per block as block elements
  - Each thread fetches one element of block
  - Perform row \* column dot products in parallel



# Example: Matrix Multiplication (4)



```
__global__ void MatrixMul( float *matA, float *matB, float *matC, int w )
{
    __shared__ float blockA[ BLOCK_SIZE ][ BLOCK_SIZE ];
    __shared__ float blockB[ BLOCK_SIZE ][ BLOCK_SIZE ];

    int bx = blockIdx.x; int tx = threadIdx.x;
    int by = blockIdx.y; int ty = threadIdx.y;

    int col = bx * BLOCK_SIZE + tx;
    int row = by * BLOCK_SIZE + ty;

    float out = 0.0f;
    for ( int m = 0; m < w / BLOCK_SIZE; m++ ) {

        blockA[ ty ][ tx ] = matA[ row * w + m * BLOCK_SIZE + tx ];
        blockB[ ty ][ tx ] = matB[ col + ( m * BLOCK_SIZE + ty ) * w ];
        __syncthreads();

        for ( int k = 0; k < BLOCK_SIZE; k++ ) {
            out += blockA[ ty ][ k ] * blockB[ k ][ tx ];
        }
        __syncthreads();
    }

    matC[ row * w + col ] = out;
}
```

Thank you.