KOÇ UNIVERSITY

Matrix Multiplication in CUDA

Didem Unat

dunat@ku.edu.tr

http://parcorelab.ku.edu.tr

Case Study

- Matrix – Matrix Multiplication in CUDA
- Optimizing for Memory Hierarchy
  - 2D Thread Blocks
  - Shared memory utilization
  - Register usage optimization

Matrix Multiply

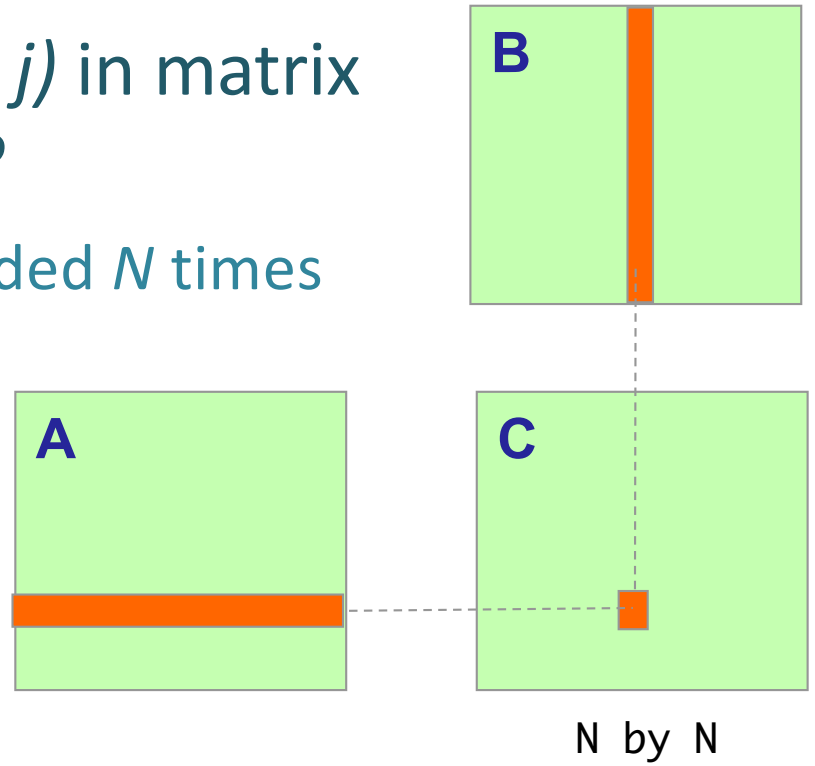
C = A * B

- Naïve Host Code for N by N square matrices

```
for (int i = 0; i<N; i++){
    for (int j = 0; j<N; j++) {
        double sum = 0.0;

        for (int k = 0; k< N; k++)
            sum += A[i * N + k] * B[k * N + j];

        C[i * N + j] = sum;
    }
}
```

Naïve Matrix Multiply

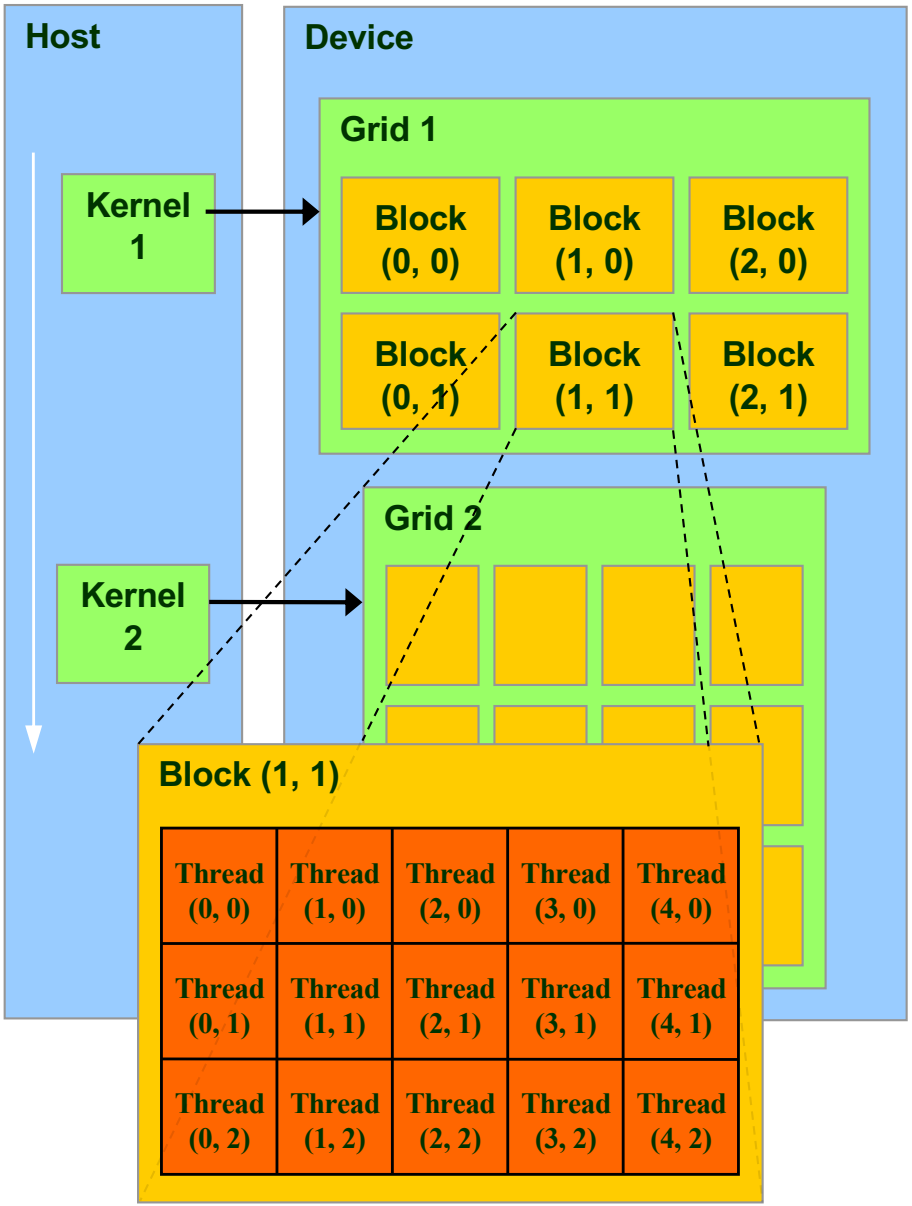
- Algorithm 1:
  - Each thread computes one element of *C*
    - Loads a row of matrix *A*
    - Loads a column of matrix *B*
    - Computes **a dot product**
- How many times a point *(i, j)* in matrix *A* is loaded from memory *?*
  - Every value of A and B is loaded *N* times from memory



Lab-7

- The lab is at
  - **/kuacc/users/dunat/COMP429/GPU/Hands-on-Labs/labs/7_MatMul**
- Implement the naïve version of matrix multiply
- Study the host code first
- Use 2D kernels to implement the naïve version
- If your code is incorrect, you get a FAILED message
- Wait for me to discuss the optimized version
  - For now, the optimized version has FIXMEs and parts of it are commented out, you will get a very high GFLOPs rates and speedups

2D Thread Blocks



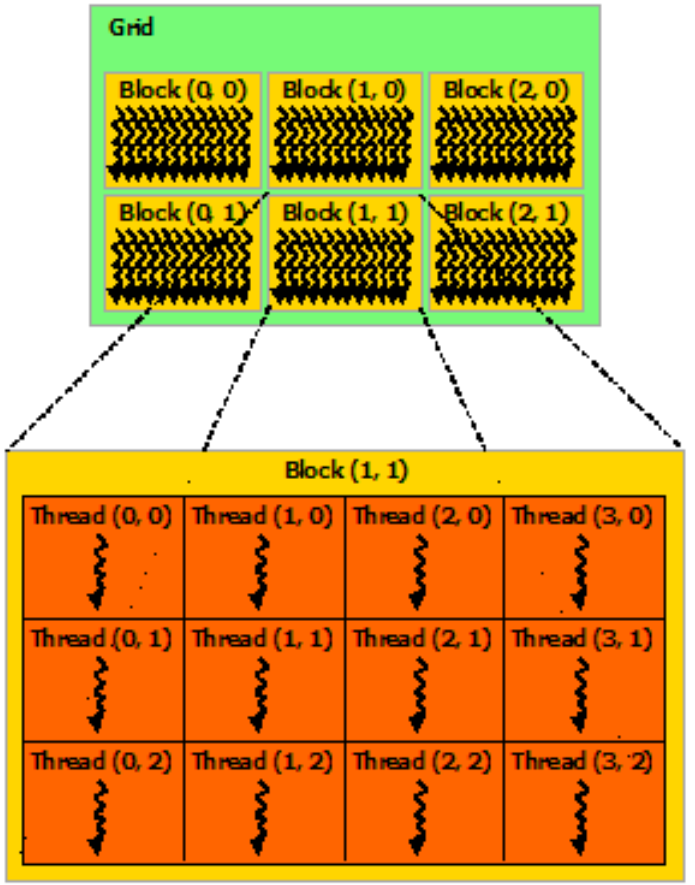
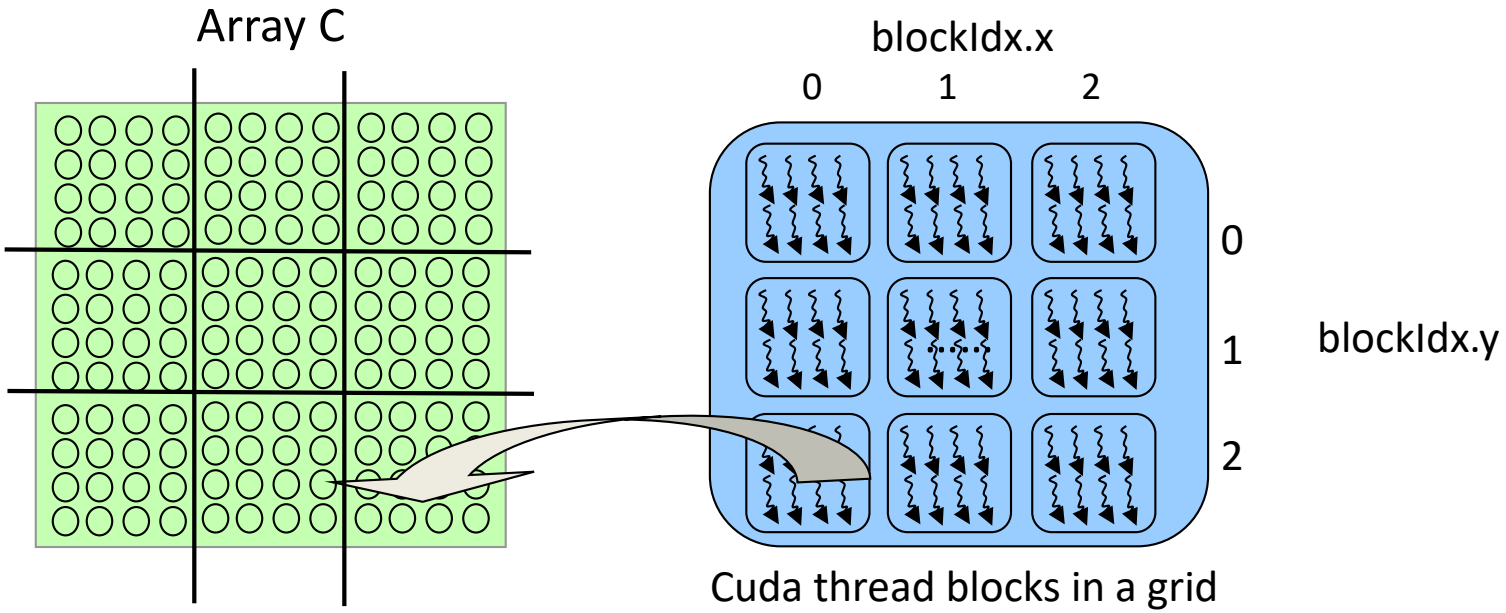


Image Courtesy: Nvidia

2D Thread Blocks

- A thread computes global indices from local indices (thread Id x and y) and block parameters
- *Array C* is divided into 2D thread blocks and a thread block consists of 2D threads.
- Each thread computes a single element of a block in Array C

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



Code stub on host side

```
unsigned int size = N*N*sizeof(float); //size in bytes

// Host arrays
float *h_A = (float*) malloc(size);
float *h_B = (float*) malloc(size);

// Initialize matrices
// Device arrays
double *d_A, *d_B, *d_C;

// cudaMalloc, do the same for d_B and d_C
cudaMalloc((void**) &d_A, size);
checkCUDAError("Error allocating device memory arrays");

// copy host memory to device
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

checkCUDAError("Error copying data to device");
```

Note that we allocated 1D arrays to represent 2D matrix

Code stub on host side (cont.)

```
// setup execution configurations, creating 2D threads
    int ntx=16, nty = 16;
    dim3 threads(ntx, nty, 1);
    dim3 grid(N / threads.x, N / threads.y);

// launch the kernel
    matMul<<< grid, threads >>>(d_C, d_A, d_B, N);


// retrieve result
     cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
     checkCUDAError("Unable to retrieve result from device");

// Free device storage and host storage
     cudaFree(d_A);
     cudaFree(d_B);
     cudaFree(d_C);
     . . .
```

Naive Implementation

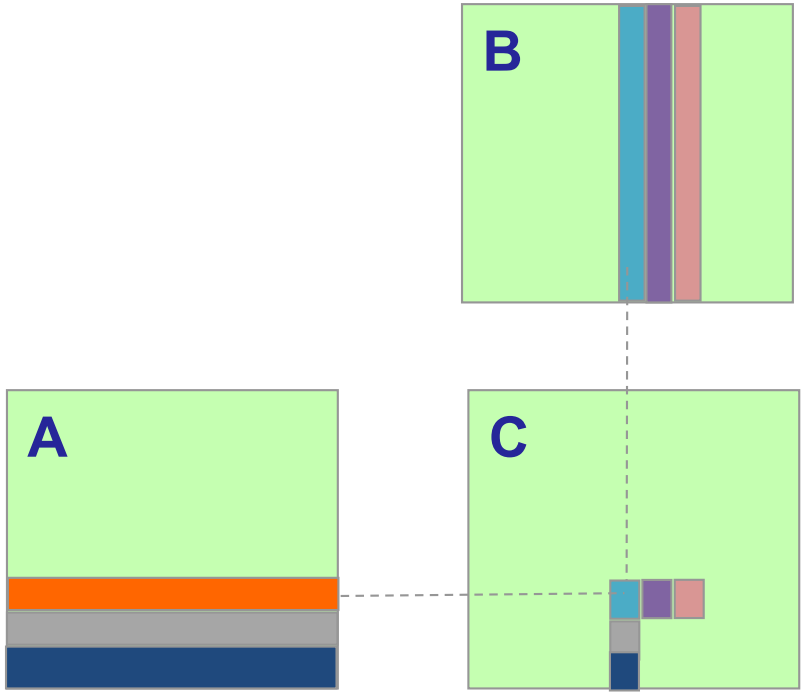
```
__global__ void
matMul( double* C, double* A, double* B, int N) {

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

if ((col < N) && (row < N)){
        double sum = 0;
        for (int k = 0; k< N; k++) {

        // perform a dot product
            double a = A[row * N + k];
            double b = B[k * N + col];
            sum += a * b;
        }
        C[row * N + col] = sum;
    }
}
```

Lab-7

- The lab is at
  - /kuacc/users/dunat/COMP429/GPU/Hands-on-Labs/labs/7_MatMul
- Improve matrix multiplication with tiling
- Compare its performance with the simple implementation
- Compare its performance with the library implementation
- Play with the input size and thread block size (TILE_DIM)

Performance Improvement

- Problems with the naïve implementation
  - Redundant global memory accesses
  - Performs dot-product for every element in C
  - Each value of A and B are read *N* times
- Can we do better?



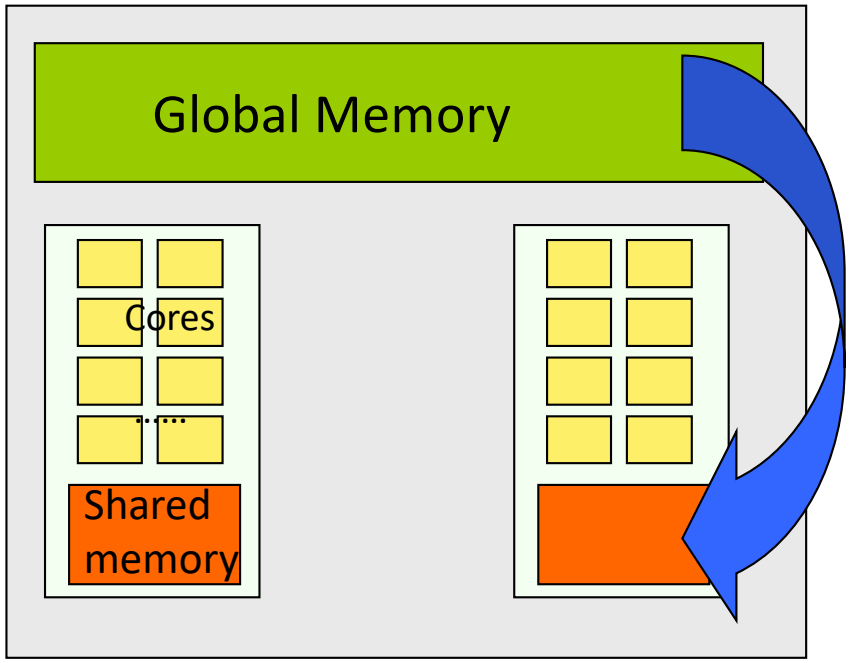
Performance Improvement

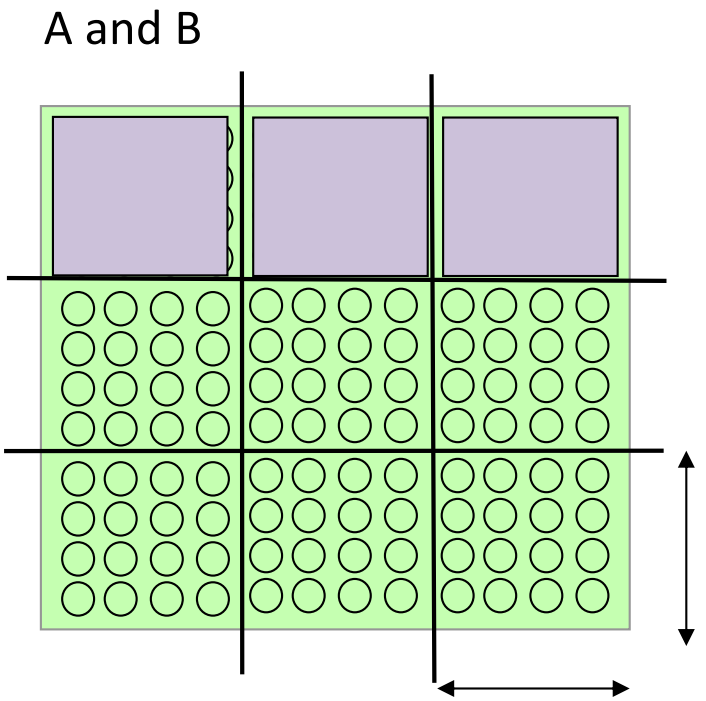
- Problems with the naïve implementation
  - Redundant global memory accesses
  - Performs dot-product for every element in C
  - Each value of A and B are read *N* times

- Improve the performance with blocking or tiling the matrix
- Use on-chip **local storage** on the GPU
  - Also called software-managed cache/memory
  - Also called shared memory
  - Threads will bring data from global memory to on-chip shared memory and reuse the data from shared memory

Shared Memory Tiles

- A and B are in global memory
- Threads cooperate to read a tile size of TILE_DIM* TILE_DIM into shared memory.
- https://www.youtube.com/watch?v=aMvCEEBIBto





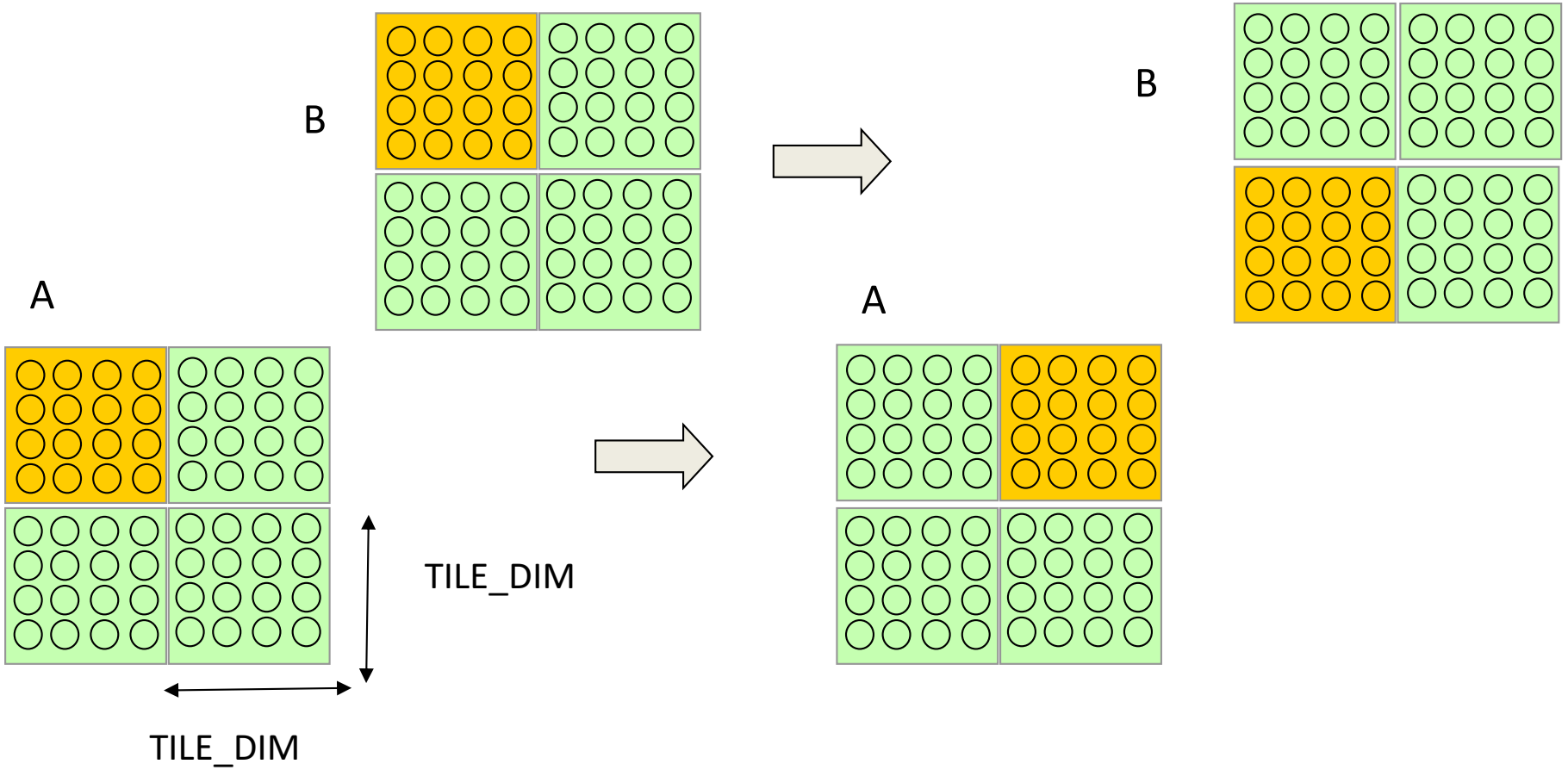
Shared Memory Tiles (cont.)

- Define memory space for A and B arrays on local storage

```
__shared__ double ATile[TILE_DIM][TILE_DIM],
__shared__ double BTile[TILE_DIM][TILE_DIM];
```

- Now, each thread block has a copy of *ATile* and *BTile*.
- Note that data in shared memory is accessible by threads only in the same block.
- Create 2D thread blocks so that each thread **loads 1 element of** A and **1 element** of B

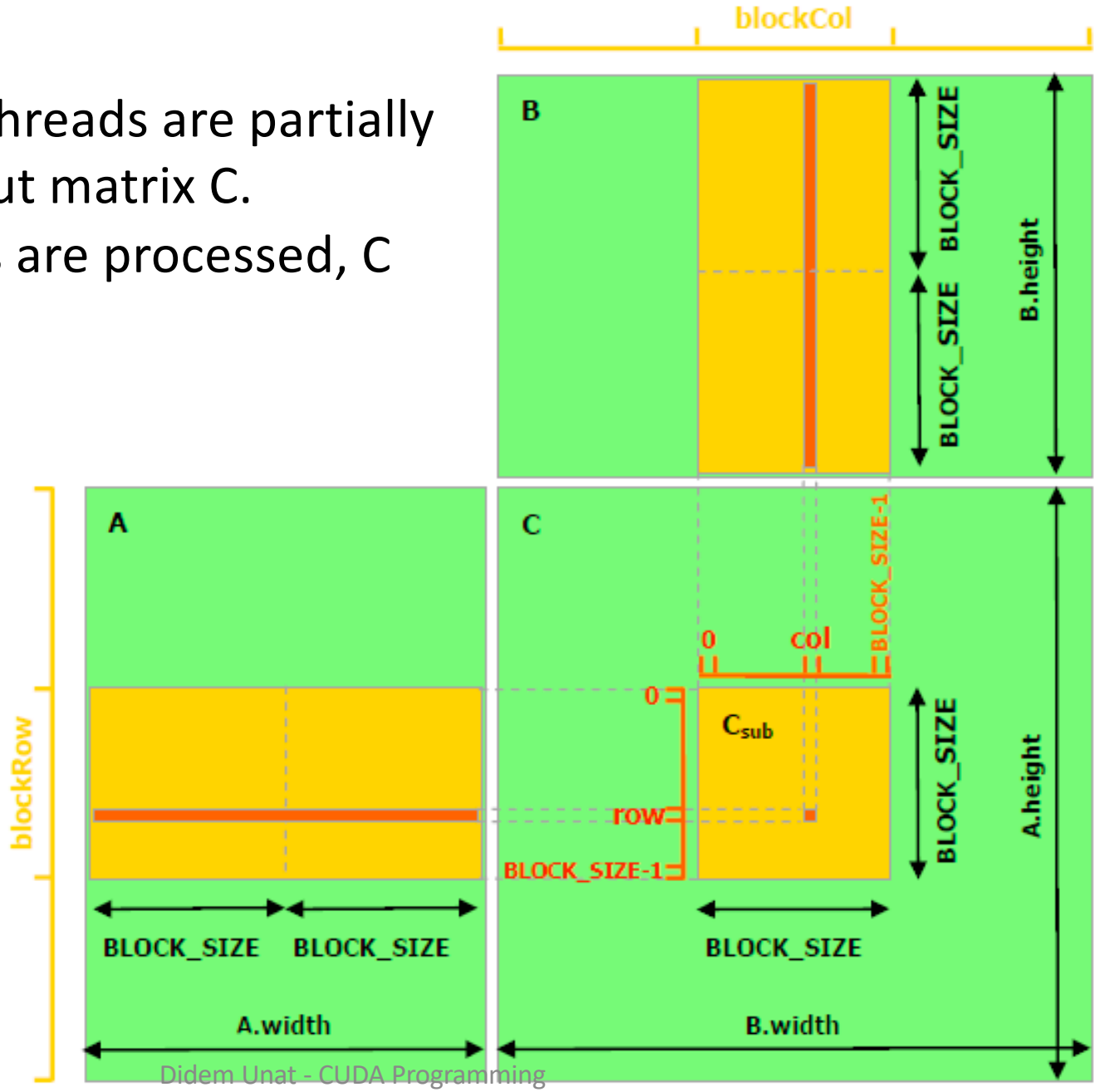
Shared Memory Blocks (cont.)




```
For each tile //(N/TILE_DIM) many tiles
   Each thread brings one element in a tile of
        A and B into shared memory
   Compute tile
```

Shared Memory Blocks

- From each tile, threads are partially computing output matrix C.
- Once all the tiles are processed, C will be ready.



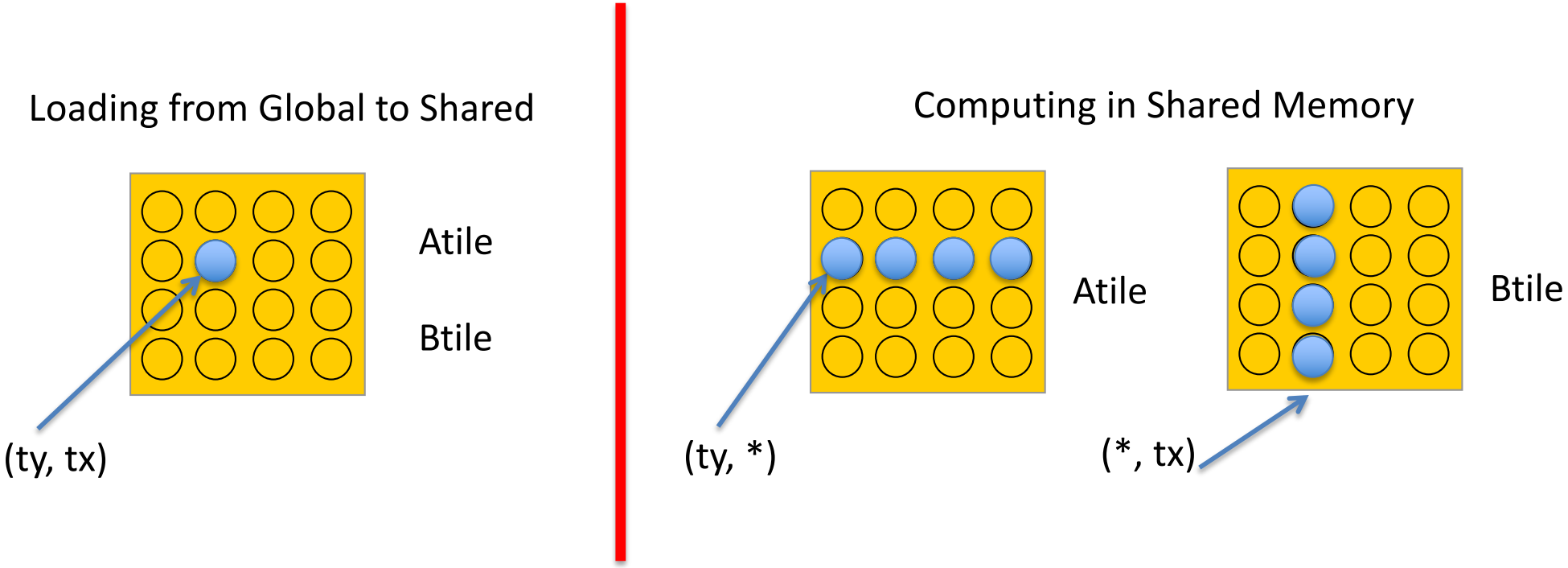
Tiled Version

```
__shared__ double ATile[TILE_DIM][TILE_DIM],
__shared__ double BTile [TILE_DIM][TILE_DIM];

//Compute global indices from thread ids and block

For each tile in X dimension
{
    //Read an element of a tile of A and B into shared memory


    //Each thread computes one element of the sub-matrix


}
//Write the sub-matrix to device memory

//Each thread writes back one element of matrix
```

Which thread brings which element of the tile?

Thread's Assignment

- A thread loads a single element from a tile A and then B from global memory into shared memory
- However, it computes a row of A or a column of B in a tile
- Other threads bring the necessary data for the tile

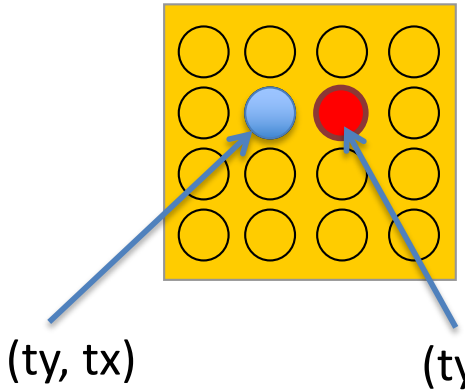


Shared Memory Tiles

```
__shared__ double ATile[TILE_DIM][TILE_DIM]
__shared__ double BTile[TILE_DIM][TILE_DIM]

for (int p =0; p < N/TILE_DIM; p++){  For each tile in X dimension
 ATile[ty][tx] = A[row*N + (p*TILE_DIM + tx)];
 BTile[ty][tx] = B[(p*TILE_DIM + ty)*N + col];
 . . .
}
```

Adjacent threads access adjacent memory locations – coalesced accesses!

Loading from Global to Shared

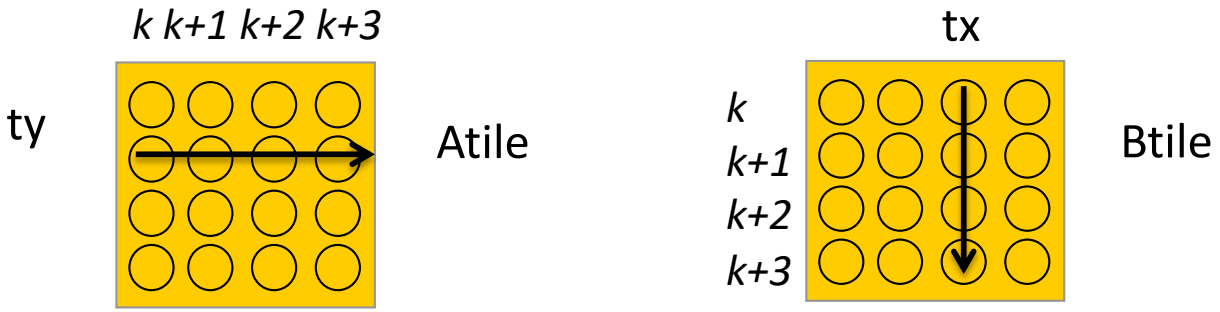


bx: BlockID.x
by: BlockID.y
tx : threadIdx.x
ty : threadIdx.y

Compute Part

```
__shared__  double  ATile[TILE_DIM][TILE_DIM],
__shared__  double  BTile[TILE_DIM][TILE_DIM];

//compute global indices from thread ids and block ids

For each tile in X dimension
{
    //read an element of a tile of A and B into shared memory

     For k in TILE_DIM
          result+= ATile[ ty][k ] * BTile[ k][tx ];    //mini dot-product


}
C[ ] = result ; // store result in global m
```

A thread accesses different columns and rows of A and B



Synchronization

```
__shared__   double  ATile[TILE_DIM][TILE_DIM],
__shared__   double  BTile[TILE_DIM][TILE_DIM];

//compute global indices from thread ids and block ids

For each tile in X dimension
{
    //read an element of a tile of A and B into shared memory
    __syncthreads();

     For k in TILE_DIM
          result+= ATile[ ty][k ] * BTile[ k][tx ];   //mini dot-product

    __syncthreads();
}
C[ ] = result ; // store result in global memory
```

Do we need to synchronize? Why twice?

- We need to synchronize before accessing shared memory because threads do not read the data they brought. They read the data that other threads brought
- No global synchronization- only threads within a thread block synchronize.

Memory Access Analysis

- In the naïve version,
  - Every thread loads every element N times
  - There are N^2 elements, as a result:
    - N^3 reads for A
    - N^3 reads for B
    - N^2 writes for C
- What is the memory traffic for shared memory implementation?
  - N^2 * N/TILE_DIM reads for A
  - N^2 * N/TILE_DIM reads for B
  - N^2 writes for C

Acknowledgments

- These slides are inspired and partly adapted from
  - Mary Hall (Univ. of Utah)
  - Programming Massively Parallel Processors: A Hands On Approach, available from *http: David Kirk and Wen-mei Hwu, February 2010, Morgan Kaufmann Publishers, ISBN 0123814723.*
  - NVidia, *CUDA Programming Guide,* available from *http://docs.nvidia.com/cuda/cuda-c-programming-guide*