CUDA Memory Model 3

Prof. Seokin Hong

Agenda

- **Matrix Multiplication**
  - **Basic Version**
  - **Tiled Version**

- Review: Memory Hierarchy
- Importance of Memory Access Efficiency

- **GPU Memory Hierarchy**
- **Improving Tiled Matrix Multiplication**
- **Impact of Memory on Parallelism**

Improving Tiled Matrix Multiplication

Review: Matrix Multiplication

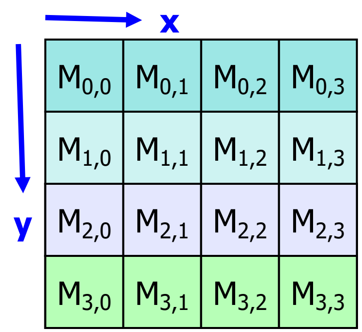
- C_{ij} = dot product of $A_{i\_}$ and $B_{\_j}$

$$\mathbf{C} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} & c_{04} \\ c_{10} & c_{11} & c_{12} & c_{13} & c_{14} \\ c_{20} & c_{21} & c_{22} & c_{23} & c_{24} \\ c_{30} & \boxed{c_{31}} & c_{32} & c_{33} & c_{34} \\ c_{40} & c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \begin{bmatrix} b_{00} & b_{01} & b_{02} & b_{03} & b_{04} \\ b_{10} & b_{11} & b_{12} & b_{13} & b_{14} \\ b_{20} & b_{21} & b_{22} & b_{23} & b_{24} \\ b_{30} & b_{31} & b_{32} & b_{33} & b_{34} \\ b_{40} & b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

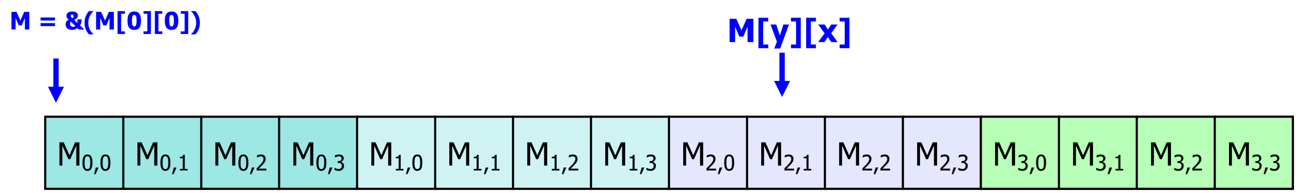
$$c_{31} = \begin{bmatrix} a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix} \cdot \begin{bmatrix} b_{01} \\ b_{11} \\ b_{21} \\ b_{31} \\ b_{41} \end{bmatrix}$$

Review: Row-major Matrix Layout in C/C++

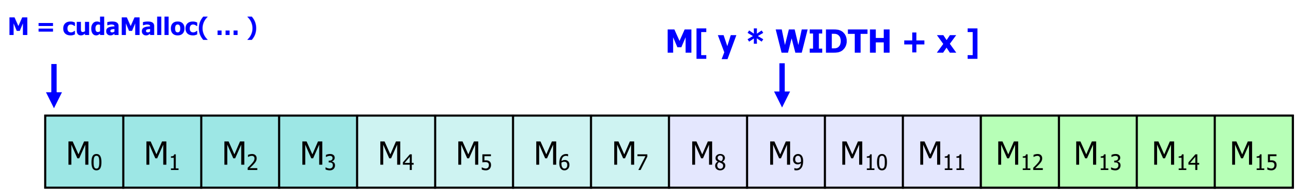
- logical layout:



- physical layout: 1D array



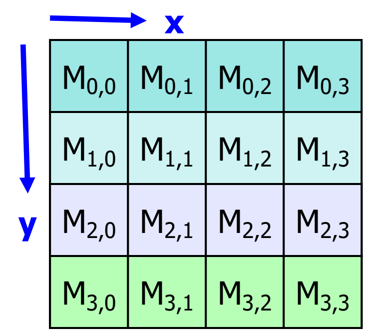
- re-interpret:



Review: CPU version

- calculate matrix multiplication on CPU

```
//calculation code
for (int y = 0; y < WIDTH; ++y) {
        for (int x = 0; x < WIDTH; ++x) {
                int sum = 0;
                for (int k = 0; k < WIDTH; ++k) {
                        sum += a[y][k] * b[k][x];
                }
                c[y][x] = sum;
        }
}
```



$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 7 \\ 4 & 5 & 6 & 7 & 8 \end{bmatrix}$$

Review: GPU version

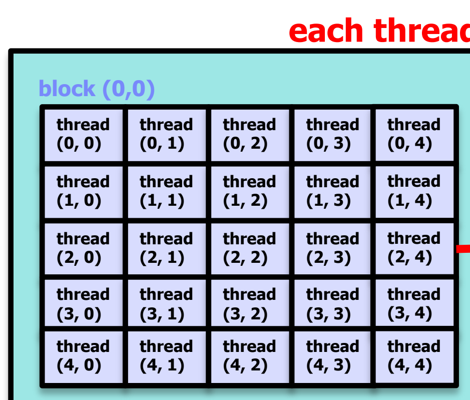
- **use WIDTH * WIDTH threads**
- **Kernel code**

```
__global__ void mulKernel(int* c, const int* a, const int* b, const int WIDTH) {
        int x = threadIdx.x;
        int y = threadIdx.y;

        int i = y * WIDTH + x;                          // [y][x] = y * WIDTH + x;

        int sum = 0;
        for (int k = 0; k < WIDTH; ++k) {
                sum += a[y * WIDTH + k] * b[k * WIDTH + x];
        }
        c[i] = sum;
}
```

$$\mathbf{C} = \begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} & c_{04} \\ c_{10} & c_{11} & c_{12} & c_{13} & c_{14} \\ c_{20} & c_{21} & c_{22} & c_{23} & c_{24} \\ c_{30} & c_{31} & c_{32} & c_{33} & c_{34} \\ c_{40} & c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

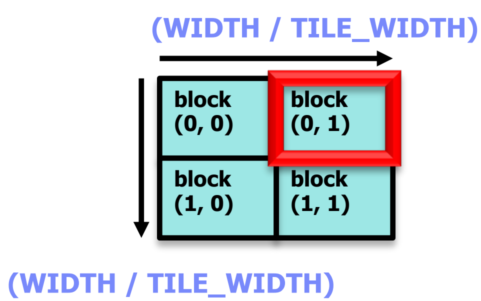


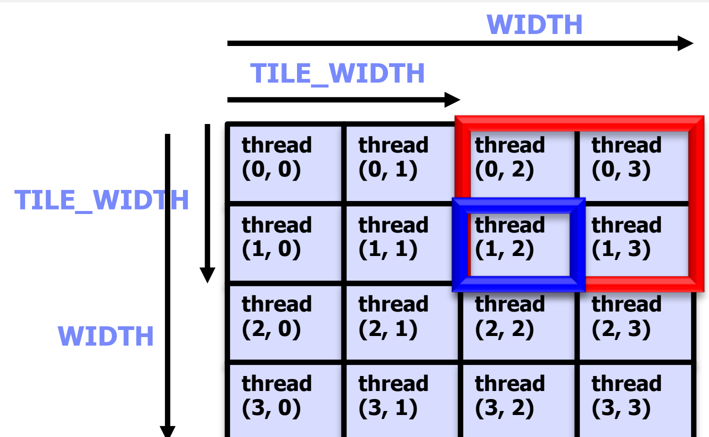
$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Review: GPU version with Tiling

- **Divide a matrix into multiple tiles and assign each tile to each block**
- **Kernel code**

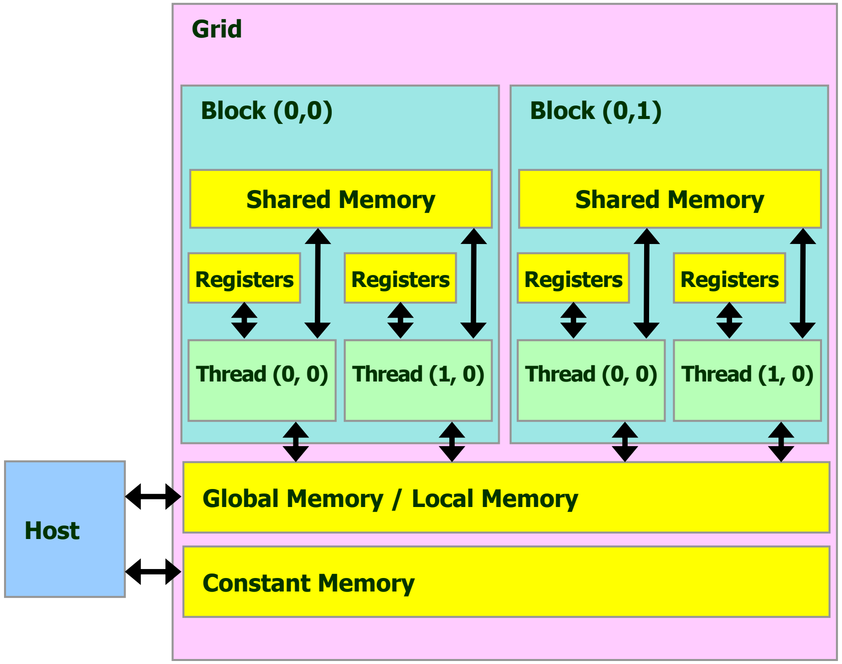
```
__global__ void matmul(float* c, const float* a, const float* b, const int width) {
    int  y = blockIdx.y * blockDim.y + threadIdx.y;
    int  x = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0F;
    for (register int k = 0; k < width; ++k) {
        float lhs = a[y * width + k];
        float rhs = b[k * width + x];
        sum += lhs * rhs;
    }
    c[y * width + x] = sum;
}
```





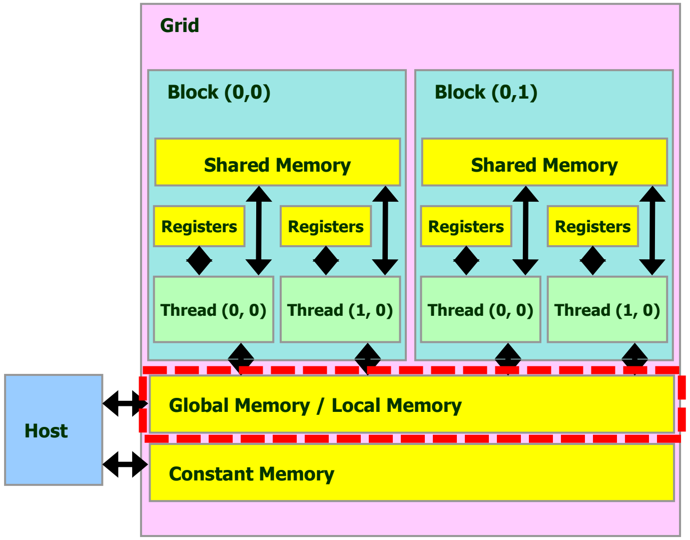
Review: CUDA Memory Hierarchy

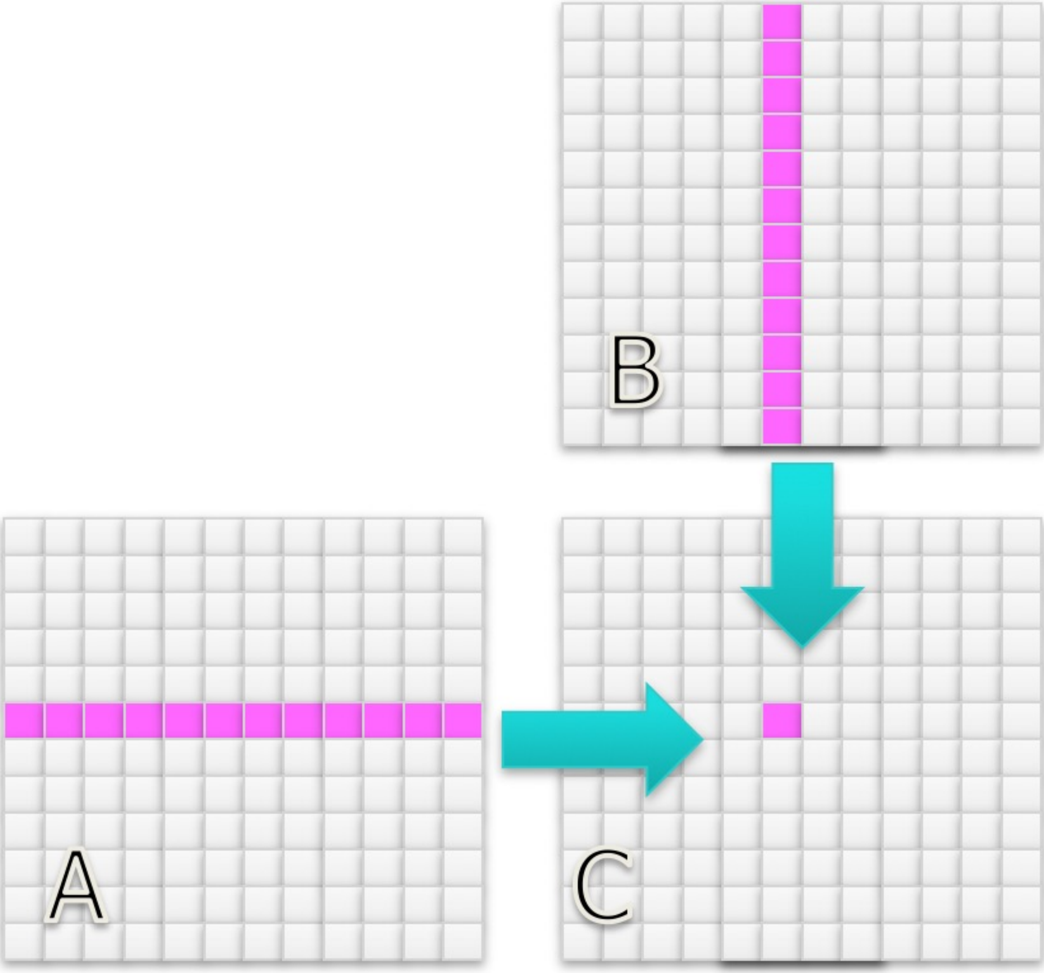
- Each thread can:
  - per-thread **registers**
    - (~1 cycle)
  - per-block **shared memory**
    - (~5 cycles)
  - per-grid **global memory**
    - (~500 cycles)
  - per-thread **local memory**
    - (~500 cycles)
    - actually, located on the global memory
  - per-grid **constant memory**
    - (~5 cycles with caching)
    - Read-only, allocated by host on device, cached on on-chip memory (fast)



Matrix Multiplication

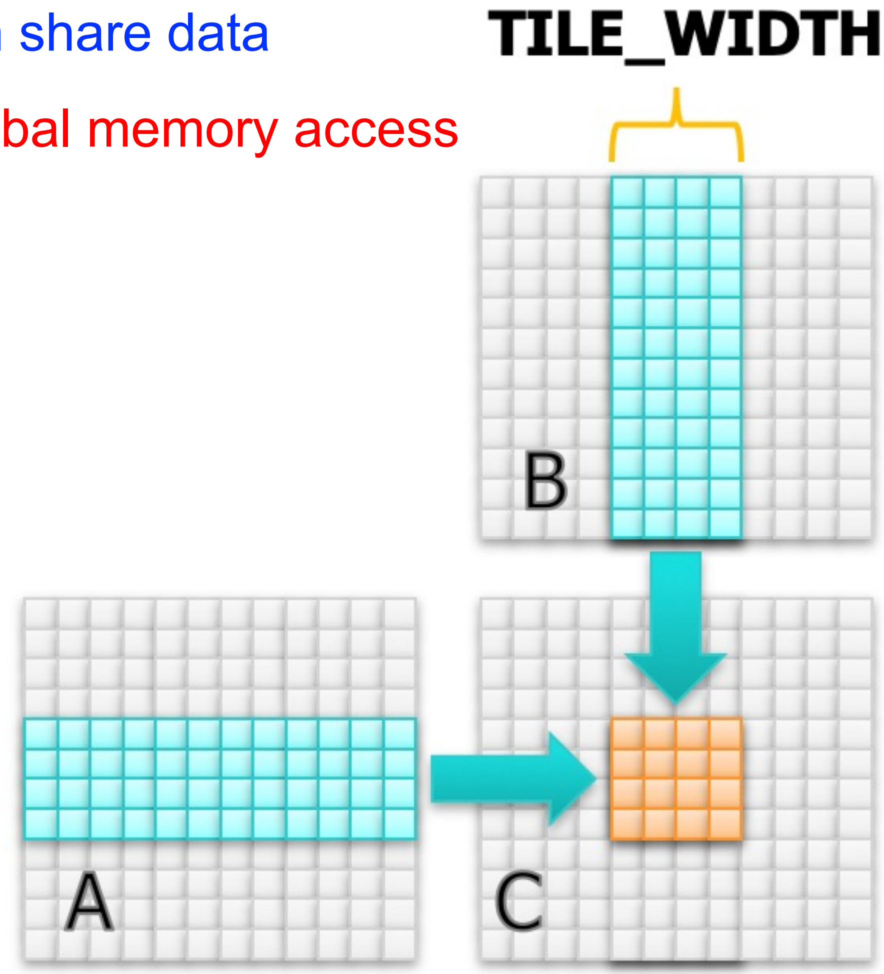
- Each thread performs global memory access

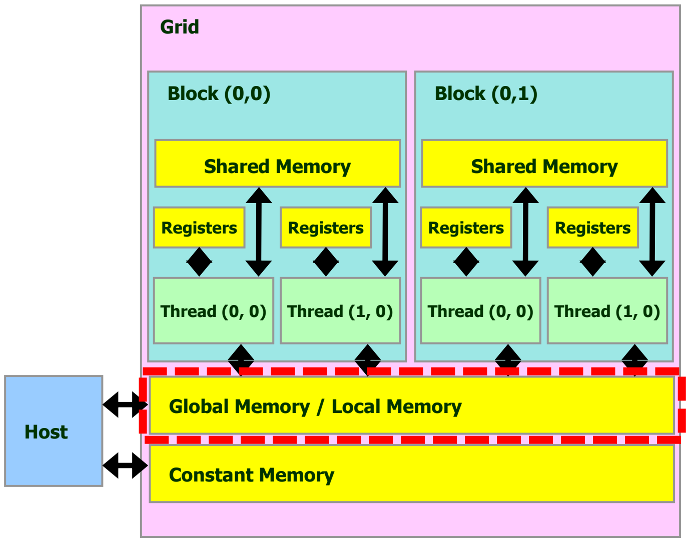




Tiled Matrix Multiplication

- Divide a result matrix into multiple tiles
- Assign each tile to each thread block
- Threads within a thread block can share data
- But, each thread still performs global memory access

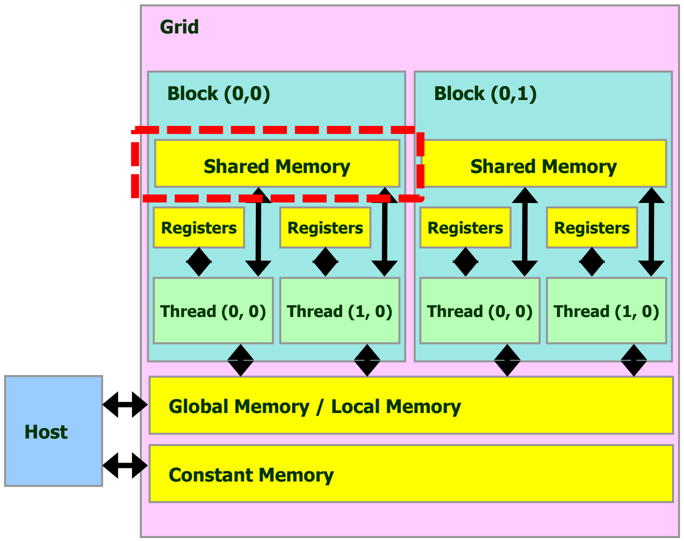


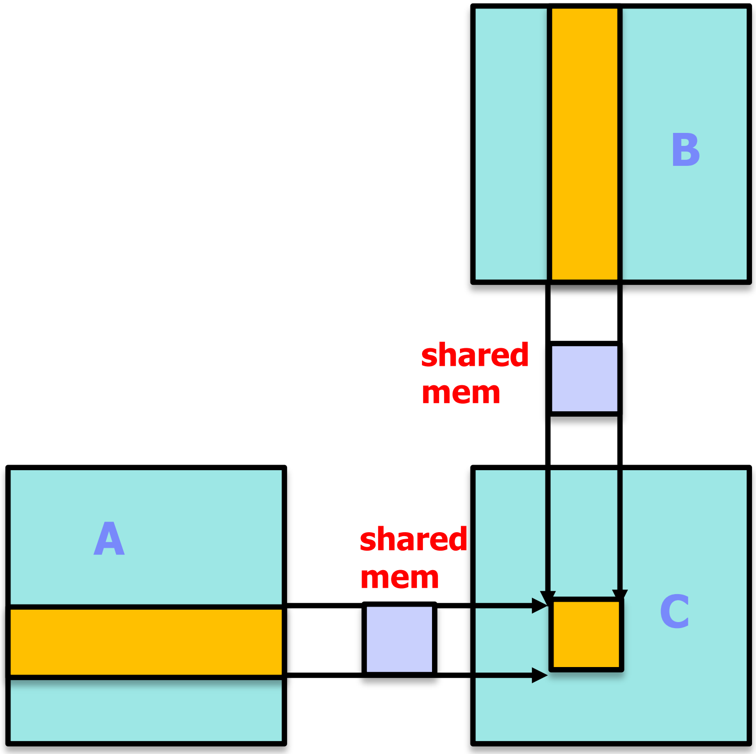


Use shared memory !

- Load each element of input matrices into **Shared Memory**

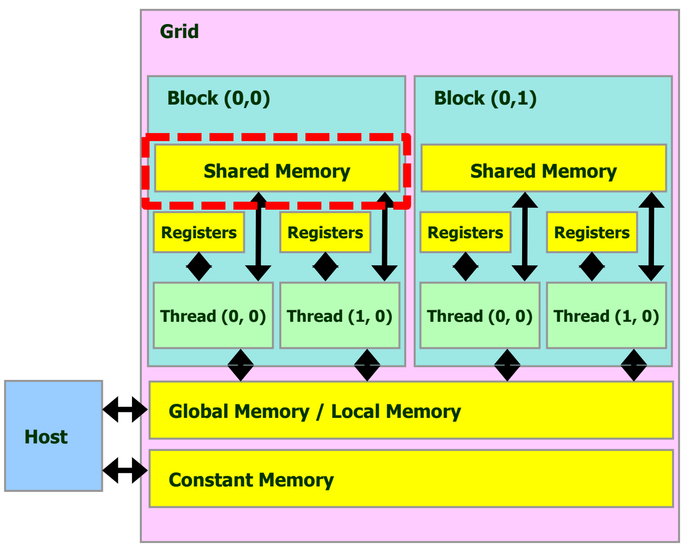
  so that several threads use the elements stored in the shared memory to reduce global memory accesses
- Use tiled approach

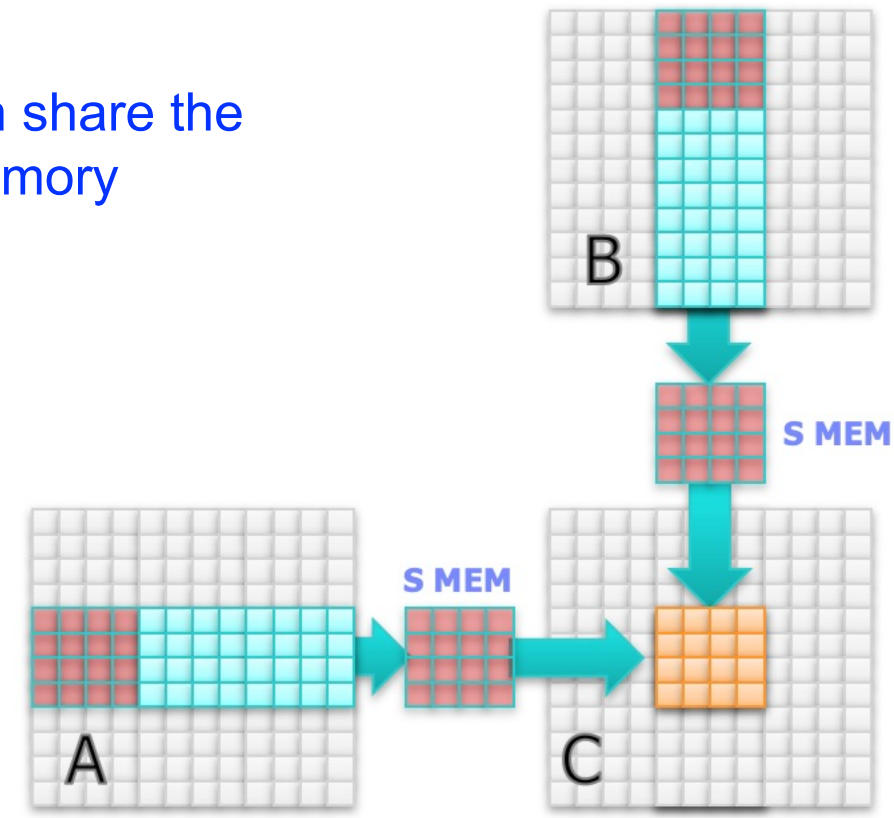




Matrix Multiplication with Shared Memory

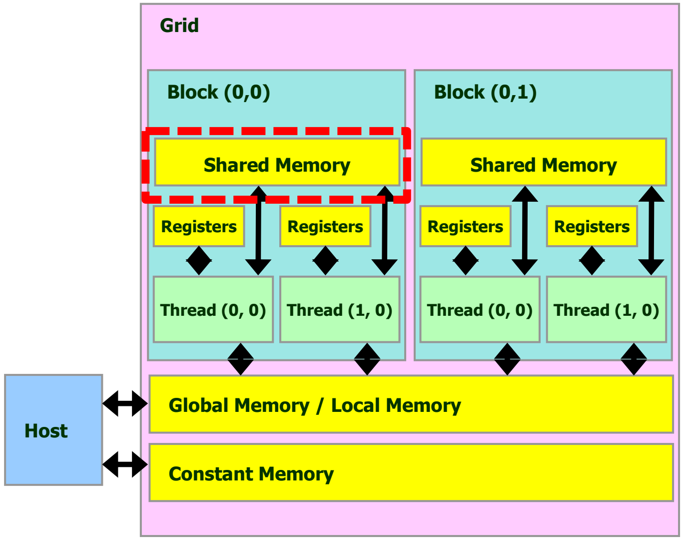
- Divide a result matrix into multiple tiles
- Assign each tile to each thread block
- Divide input matrices into multiple tiles
- **Load the tiles of input matrices to the shared memory**
- Threads within a thread block can share the elements stored in the shared memory

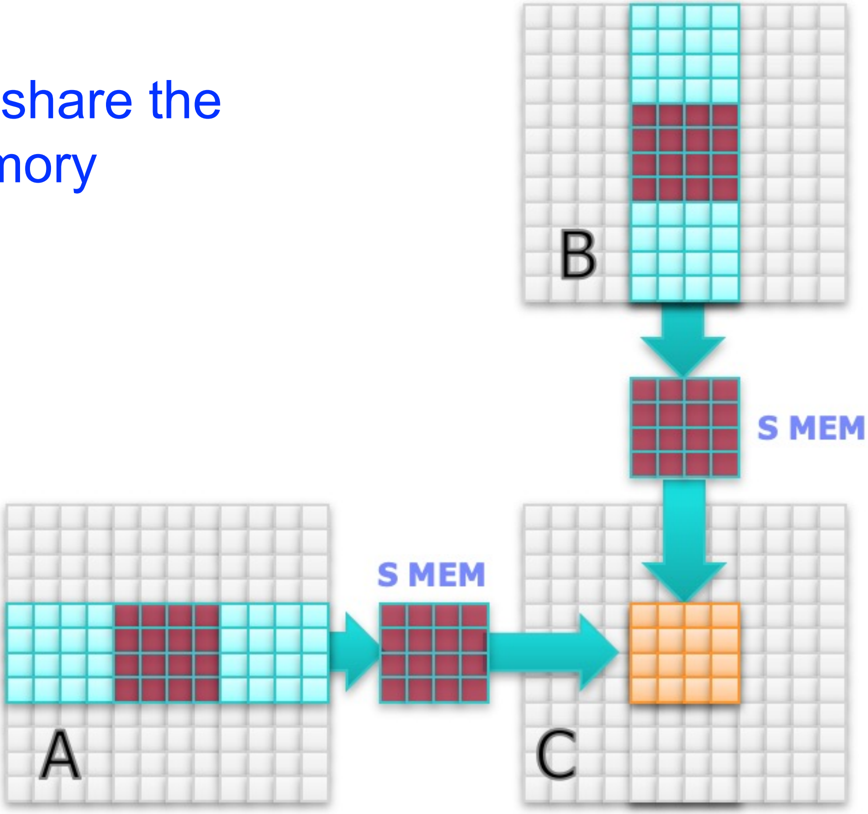




Matrix Multiplication with Shared Memory(Cont'd)

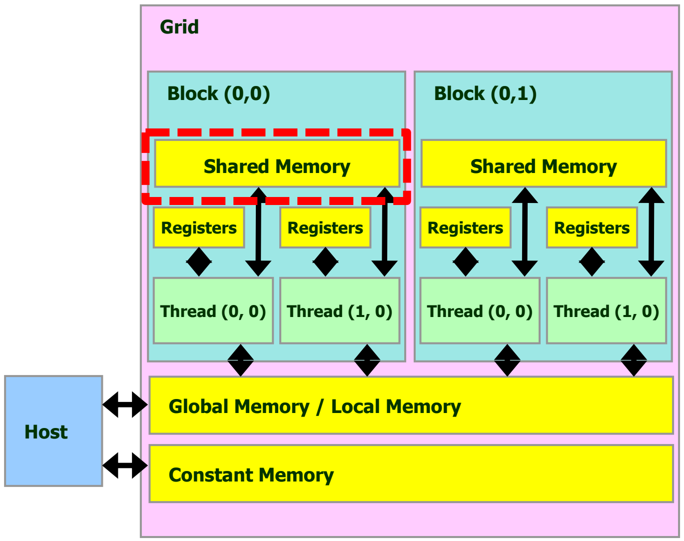
- Divide a result matrix into multiple tiles
- Assign each tile to each thread block
- Divide input matrices into multiple tiles
- **Load the tiles of input matrices to the shared memory**
- Threads within a thread block can share the elements stored in the shared memory

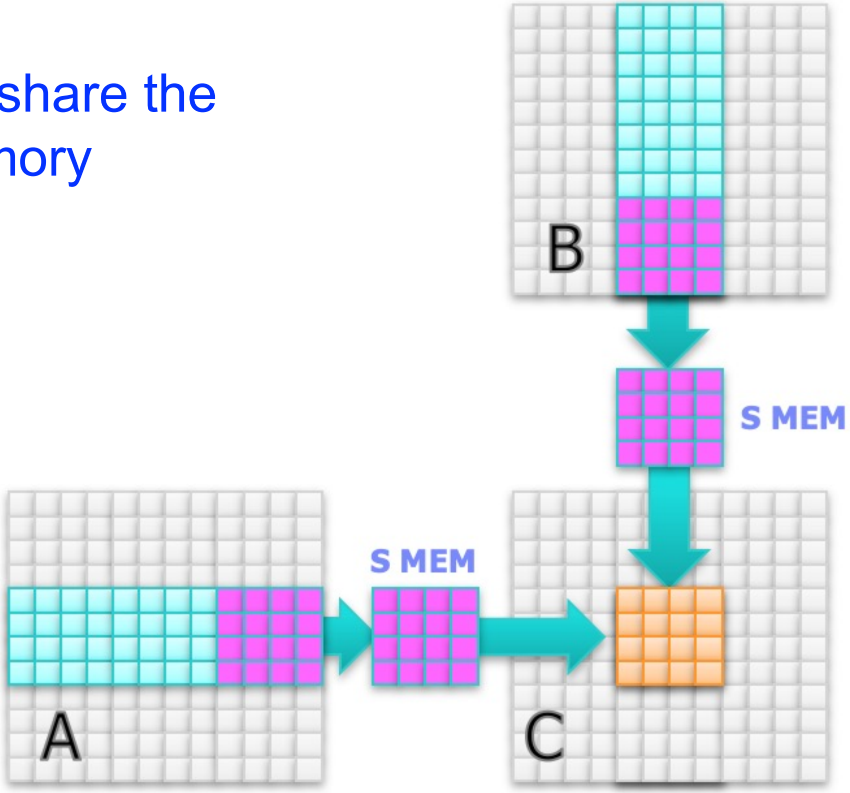




Matrix Multiplication with Shared Memory(Cont'd)

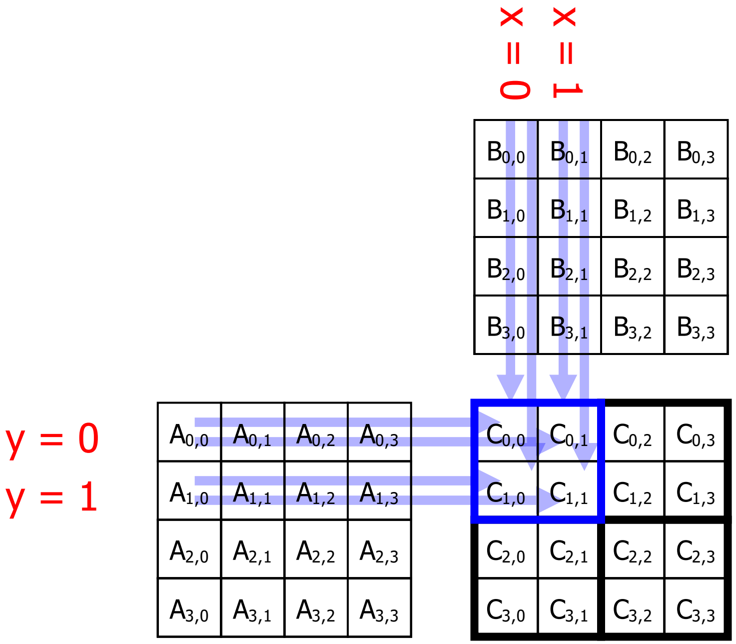
- Divide a result matrix into multiple tiles
- Assign each tile to each thread block
- Divide input matrices into multiple tiles
- **Load the tiles of input matrices to the shared memory**
- Threads within a thread block can share the elements stored in the shared memory





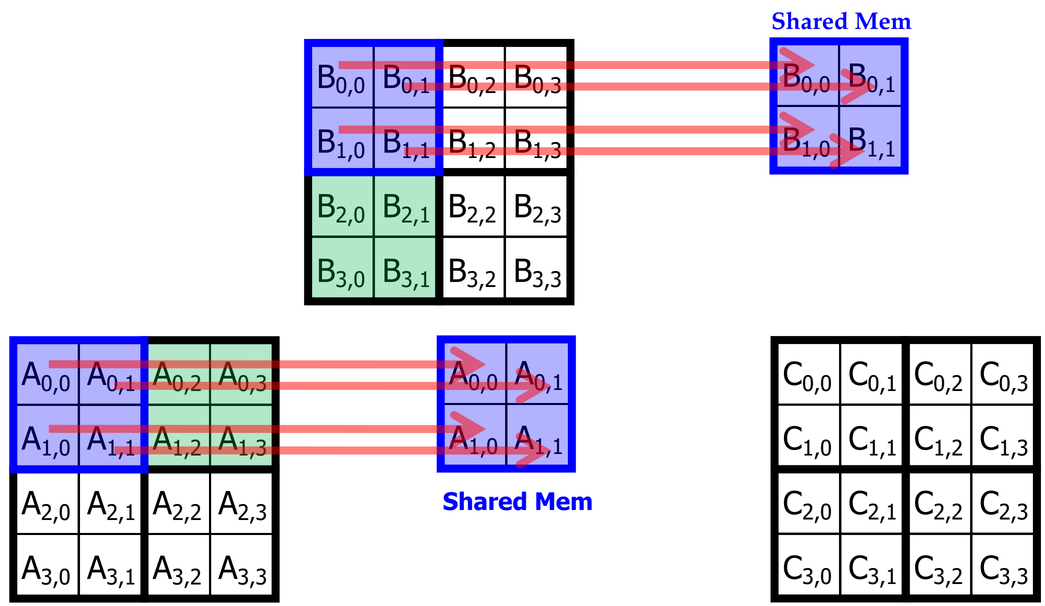
Work for Block (0,0), TILE_WIDTH = 2

- $C_{00} = A_{00} * B_{00} + A_{01} * B_{10} + A_{02} * B_{20} + A_{03} * B_{30}$
- $C_{01} = A_{00} * B_{01} + A_{01} * B_{11} + A_{02} * B_{21} + A_{03} * B_{31}$
- $C_{10} = A_{10} * B_{00} + A_{11} * B_{10} + A_{12} * B_{20} + A_{13} * B_{30}$
- $C_{11} = A_{10} * B_{01} + A_{11} * B_{11} + A_{12} * B_{21} + A_{13} * B_{31}$



Work for Block (0,0) (cont'd)

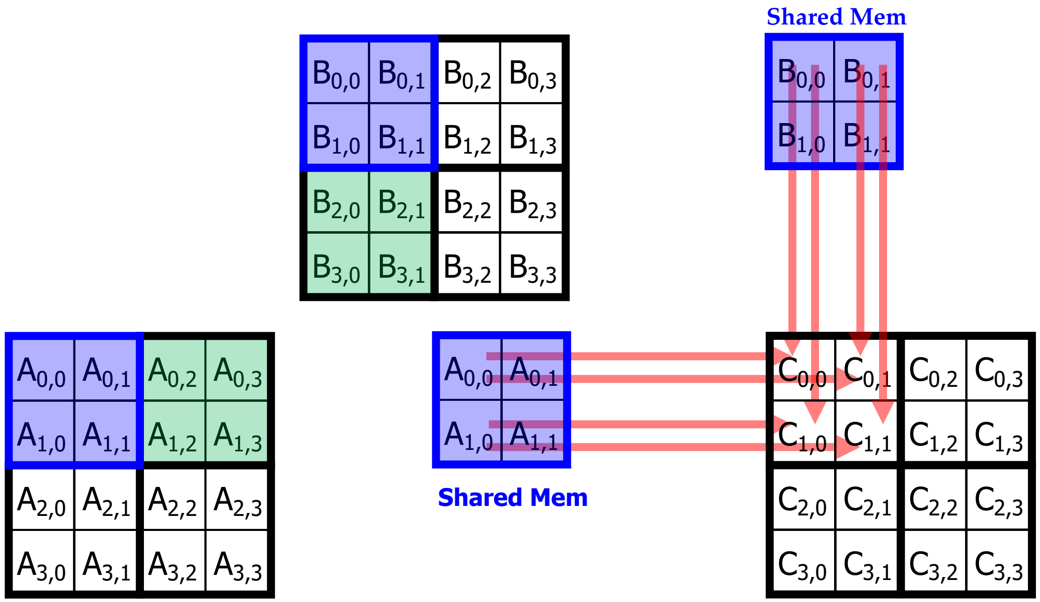
- $C_{00} = A_{00} * B_{00} + A_{01} * B_{10} + A_{02} * B_{20} + A_{03} * B_{30}$
- $C_{01} = A_{00} * B_{01} + A_{01} * B_{11} + A_{02} * B_{21} + A_{03} * B_{31}$
- $C_{10} = A_{10} * B_{00} + A_{11} * B_{10} + A_{12} * B_{20} + A_{13} * B_{30}$
- $C_{11} = A_{10} * B_{01} + A_{11} * B_{11} + A_{12} * B_{21} + A_{13} * B_{31}$



Load the elements (tile) of matrix A and B to the shared memory!

Work for Block (0,0) (cont'd)

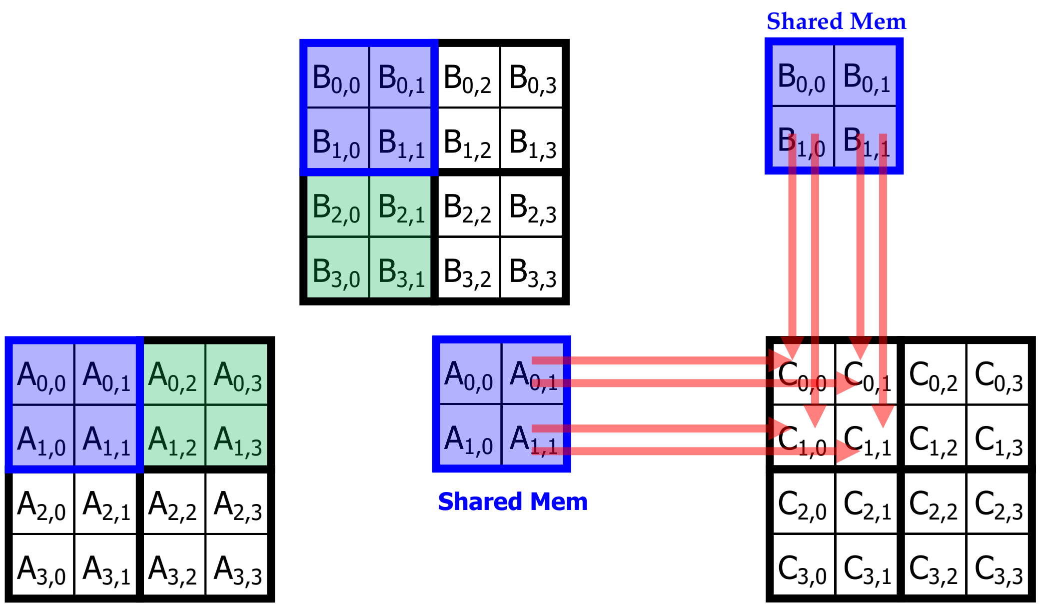
- $C_{00} = A_{00} * B_{00} + A_{01} * B_{10} + A_{02} * B_{20} + A_{03} * B_{30}$
- $C_{01} = A_{00} * B_{01} + A_{01} * B_{11} + A_{02} * B_{21} + A_{03} * B_{31}$
- $C_{10} = A_{10} * B_{00} + A_{11} * B_{10} + A_{12} * B_{20} + A_{13} * B_{30}$
- $C_{11} = A_{10} * B_{01} + A_{11} * B_{11} + A_{12} * B_{21} + A_{13} * B_{31}$



Calculate the **partial sum** for each element of the matrix C (result) with the elements stored in the shared memory

Work for Block (0,0) (cont'd)

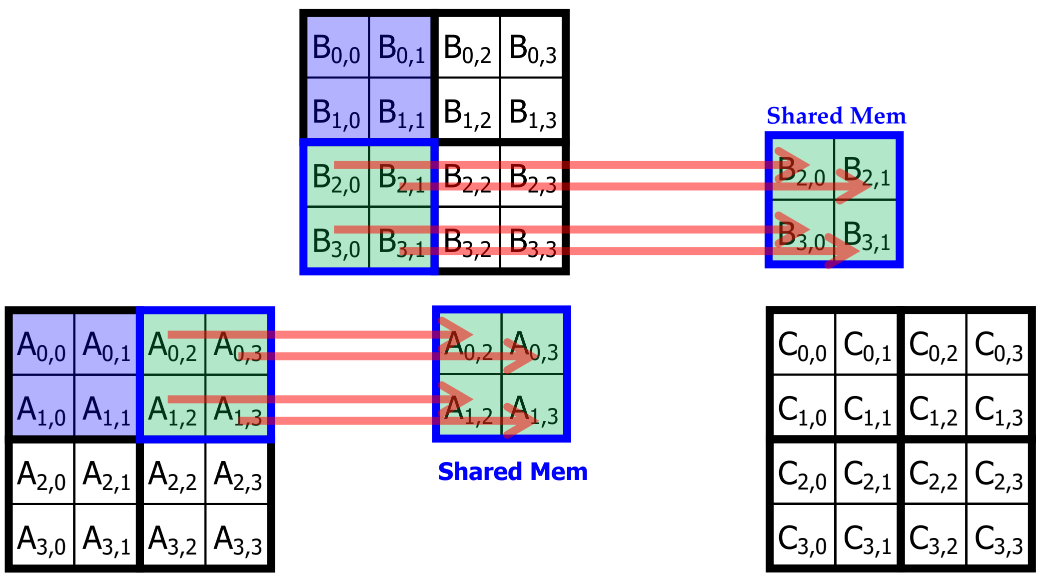
- $C_{00} = A_{00} * B_{00} + A_{01} * B_{10} + A_{02} * B_{20} + A_{03} * B_{30}$
- $C_{01} = A_{00} * B_{01} + A_{01} * B_{11} + A_{02} * B_{21} + A_{03} * B_{31}$
- $C_{10} = A_{10} * B_{00} + A_{11} * B_{10} + A_{12} * B_{20} + A_{13} * B_{30}$
- $C_{11} = A_{10} * B_{01} + A_{11} * B_{11} + A_{12} * B_{21} + A_{13} * B_{31}$



Calculate the **partial sum** for each element of the matrix C (result) with the elements stored in the shared memory

Work for Block (0,0) (cont'd)

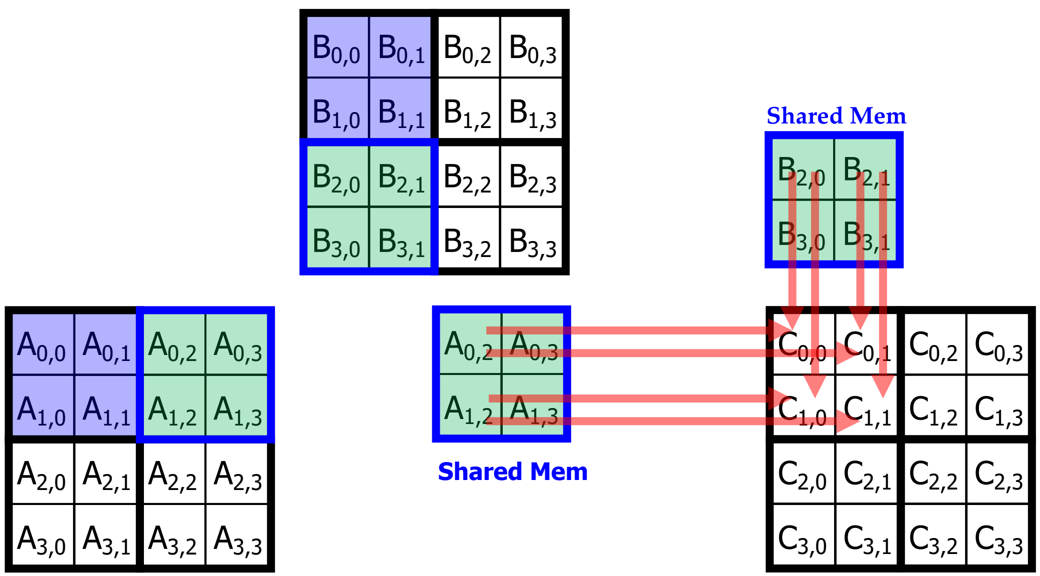
- $C_{00} = A_{00} * B_{00} + A_{01} * B_{10} + A_{02} * B_{20} + A_{03} * B_{30}$
- $C_{01} = A_{00} * B_{01} + A_{01} * B_{11} + A_{02} * B_{21} + A_{03} * B_{31}$
- $C_{10} = A_{10} * B_{00} + A_{11} * B_{10} + A_{12} * B_{20} + A_{13} * B_{30}$
- $C_{11} = A_{10} * B_{01} + A_{11} * B_{11} + A_{12} * B_{21} + A_{13} * B_{31}$



Load the elements (tile) of matrix A and B to the shared memory!

Work for Block (0,0) (cont'd)

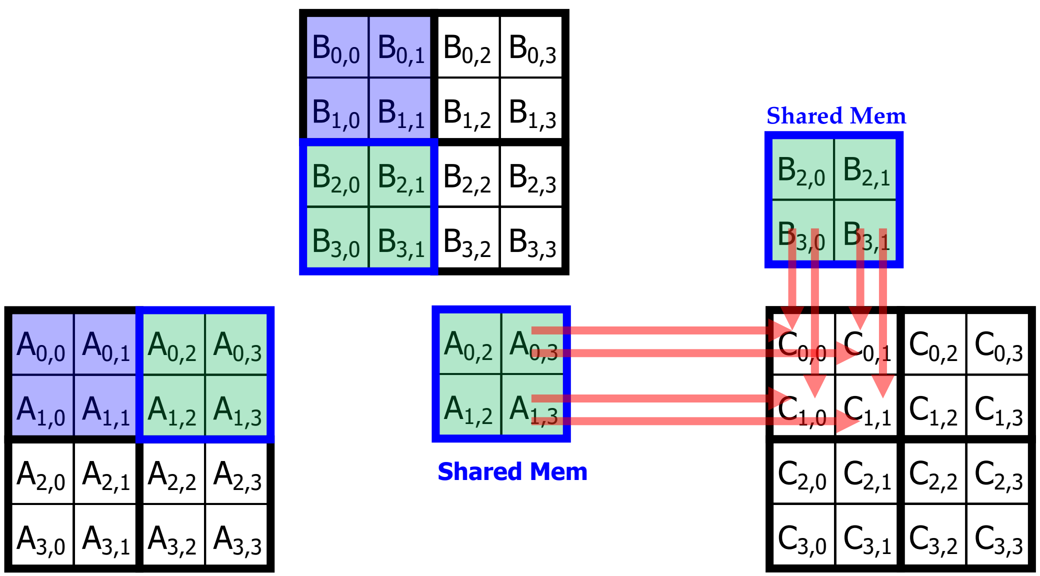
- $C_{00} = A_{00} * B_{00} + A_{01} * B_{10} + A_{02} * B_{20} + A_{03} * B_{30}$
- $C_{01} = A_{00} * B_{01} + A_{01} * B_{11} + A_{02} * B_{21} + A_{03} * B_{31}$
- $C_{10} = A_{10} * B_{00} + A_{11} * B_{10} + A_{12} * B_{20} + A_{13} * B_{30}$
- $C_{11} = A_{10} * B_{01} + A_{11} * B_{11} + A_{12} * B_{21} + A_{13} * B_{31}$



Calculate the **partial sum** for each element of the matrix C (result) with the elements stored in the shared memory

Work for Block (0,0) (cont'd)

- $C_{00} = A_{00} * B_{00} + A_{01} * B_{10} + A_{02} * B_{20} + A_{03} * B_{30}$
- $C_{01} = A_{00} * B_{01} + A_{01} * B_{11} + A_{02} * B_{21} + A_{03} * B_{31}$
- $C_{10} = A_{10} * B_{00} + A_{11} * B_{10} + A_{12} * B_{20} + A_{13} * B_{30}$
- $C_{11} = A_{10} * B_{01} + A_{11} * B_{11} + A_{12} * B_{21} + A_{13} * B_{31}$

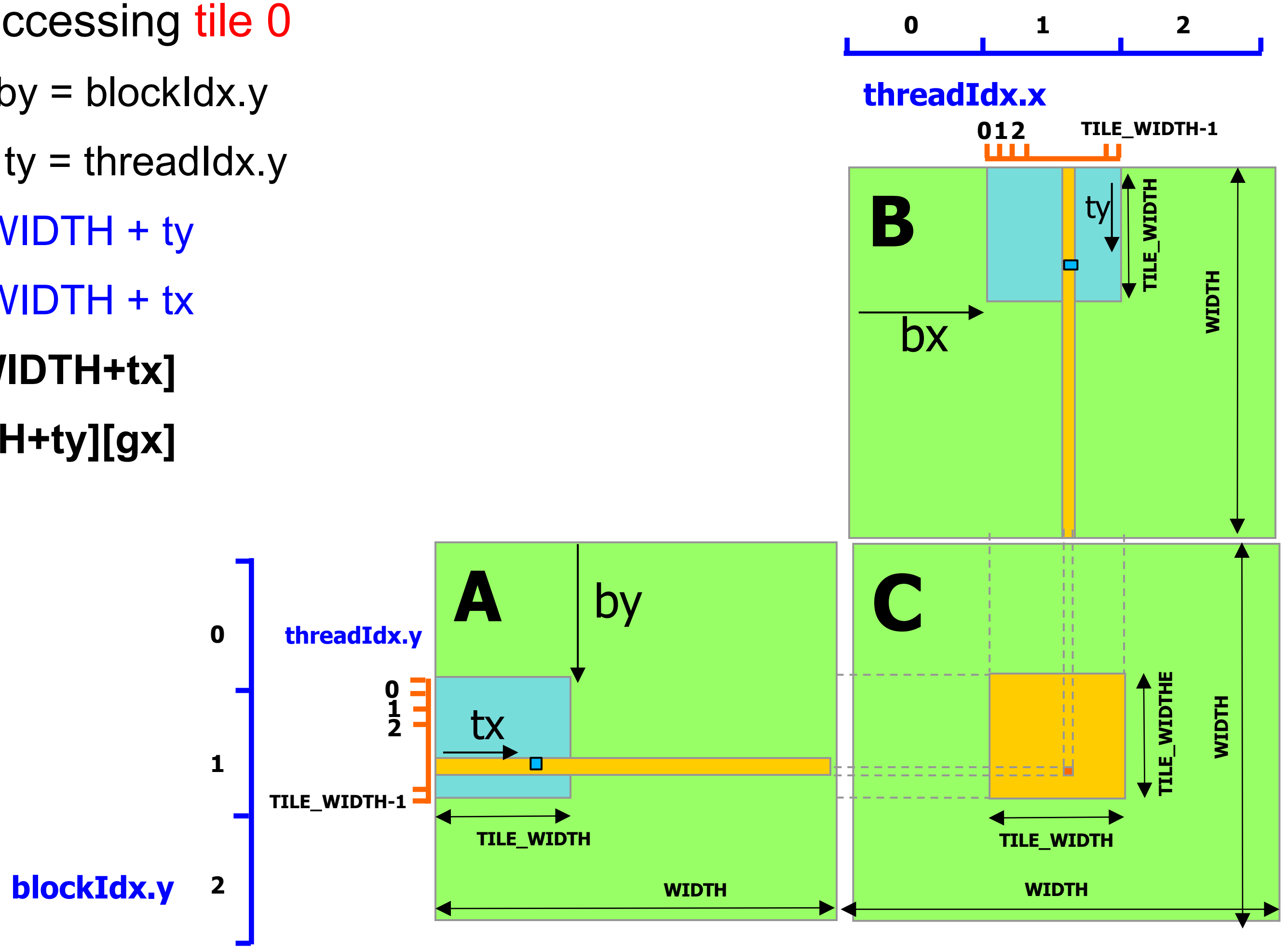


Calculate the **partial sum** for each element of the matrix C (result) with the elements stored in the shared memory

Loading an Input Tile

- All threads in a thread block participate
  - loads one tile of matrix A and one tile of matrix B

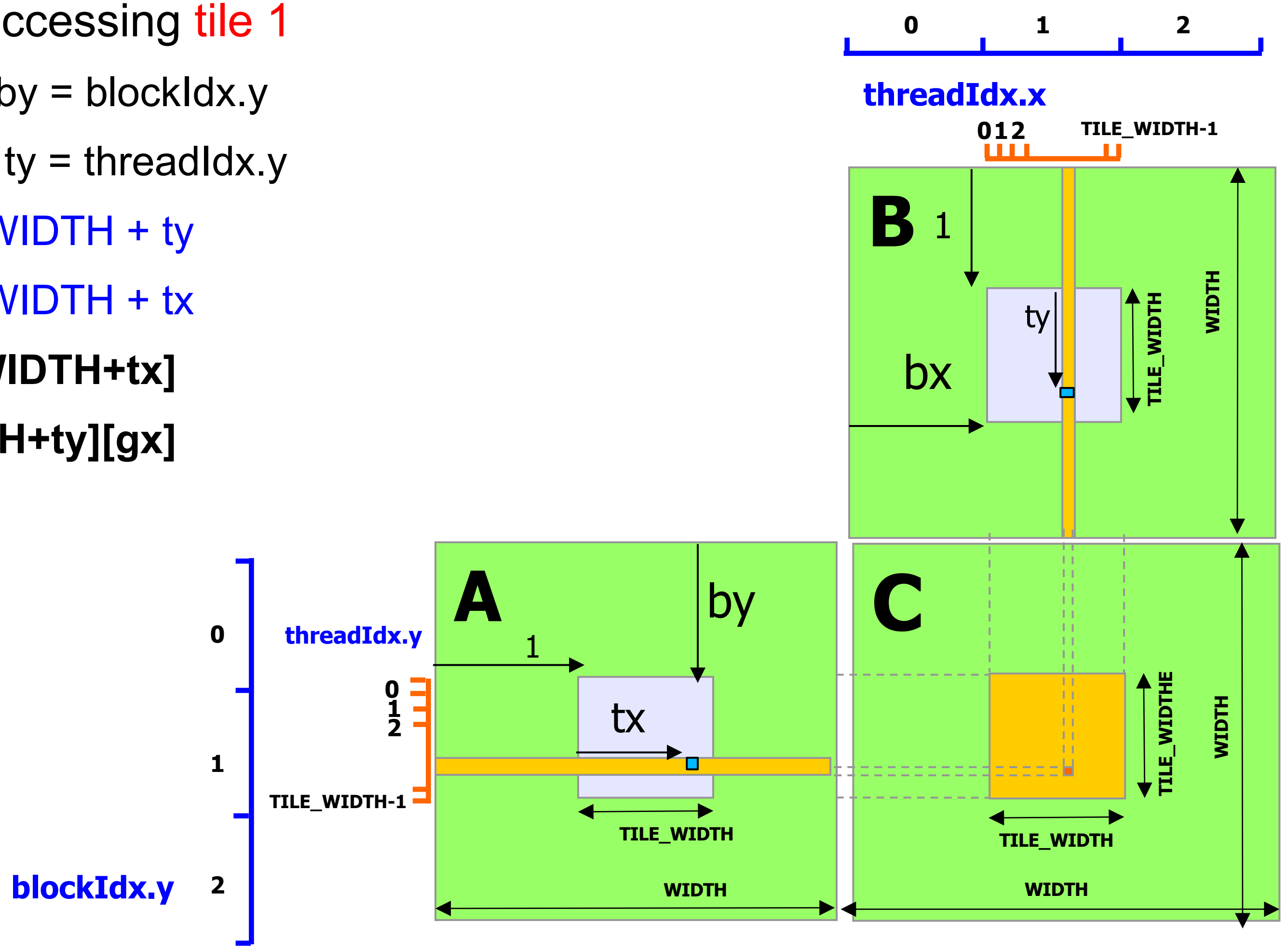
- 2D indexing for accessing tile 0
  - bx = blockIdx.x, by = blockIdx.y
  - tx = threadIdx.x, ty = threadIdx.y
  - gy = by * TILE_WIDTH + ty
  - gx = bx * TILE_WIDTH + tx
  - **A[gy][0*TILE_WIDTH+tx]**
  - **B[0*TILE_WIDTH+ty][gx]**



Loading an Input Tile (cont'd)

- All threads in a thread block participate
  - loads one tile of matrix A and one tile of matrix B

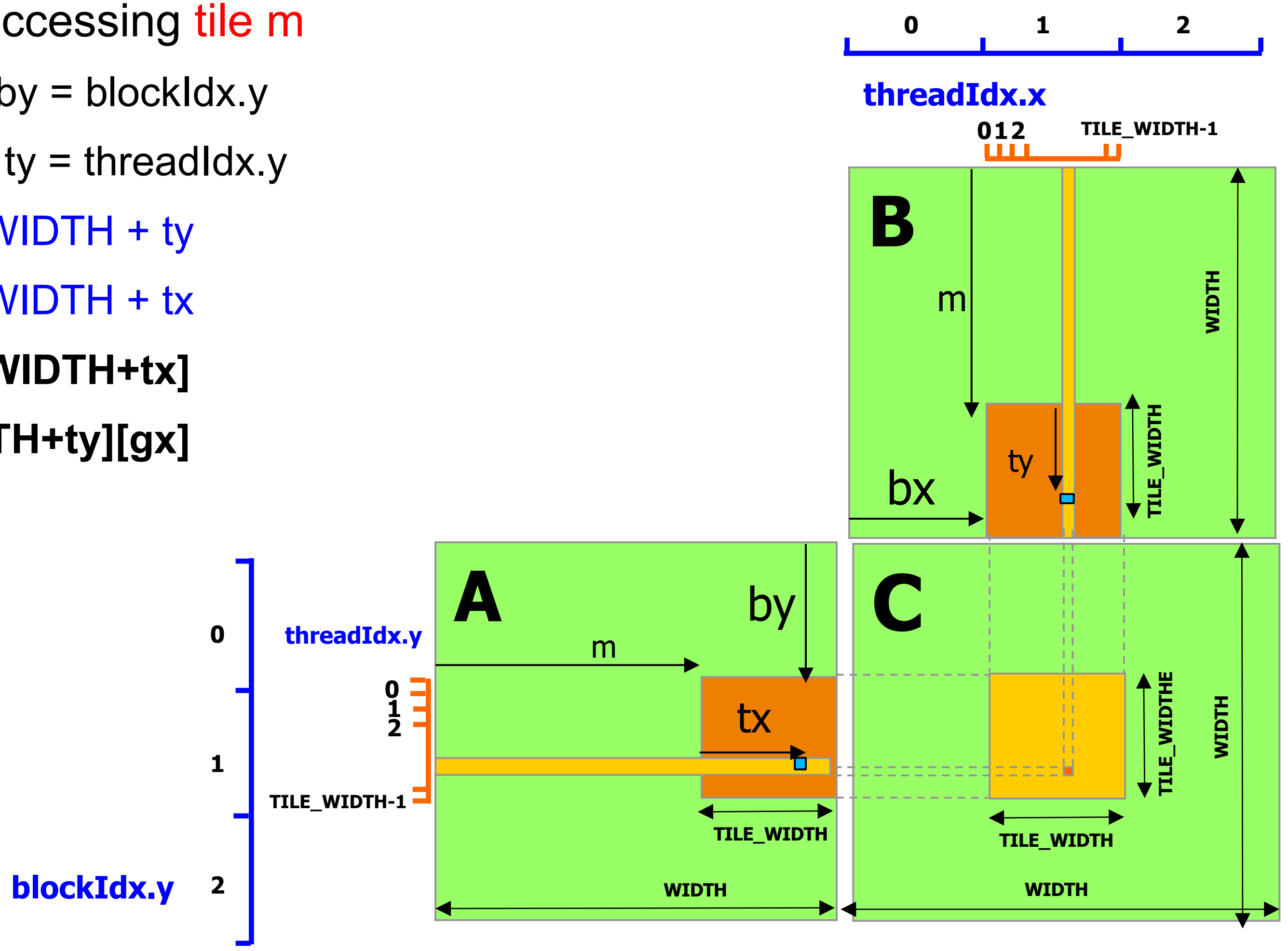
- 2D indexing for accessing tile 1
  - bx = blockIdx.x, by = blockIdx.y
  - tx = threadIdx.x, ty = threadIdx.y
  - gy = by * TILE_WIDTH + ty
  - gx = bx * TILE_WIDTH + tx
  - **A[gy][1*TILE_WIDTH+tx]**
  - **B[1*TILE_WIDTH+ty][gx]**



Loading an Input Tile

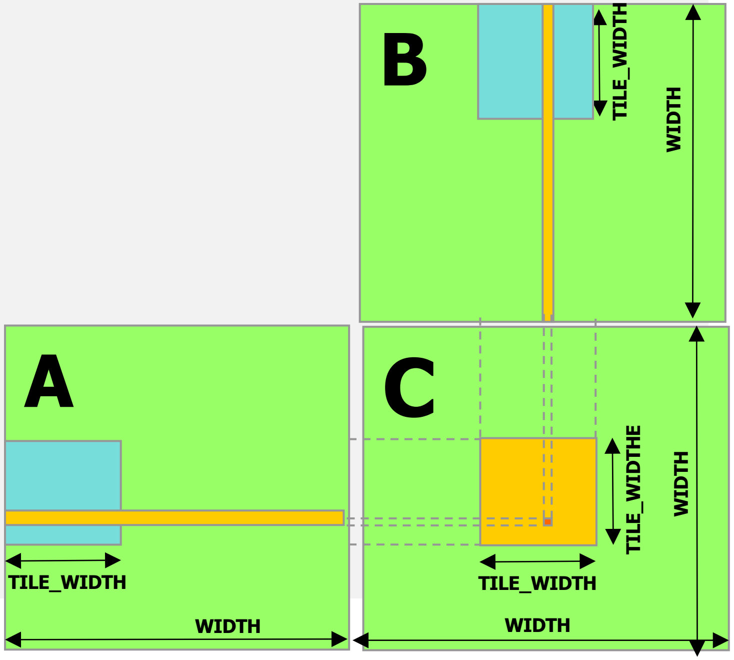
- All threads in a thread block participate
  - loads one tile of matrix A and one tile of matrix B

- 2D indexing for accessing tile m
  - bx = blockIdx.x, by = blockIdx.y
  - tx = threadIdx.x, ty = threadIdx.y
  - gy = by * TILE_WIDTH + ty
  - gx = bx * TILE_WIDTH + tx
  - **A[gy][m*TILE_WIDTH+tx]**
  - **B[m*TILE_WIDTH+ty][gx]**



Tiled Matrix Multiplication Kernel

```
__global__ void matmul(float* g_C, const float* g_A, const float* g_B, const int width) {
    __shared__ float s_A[TILE_WIDTH][TILE_WIDTH];
    __shared__ float s_B[TILE_WIDTH][TILE_WIDTH];
    int by = blockIdx.y; int bx = blockIdx.x;
    int ty = threadIdx.y; int tx = threadIdx.x;
    int gy = by * TILE_WIDTH + ty; // global y index
    int gx = bx * TILE_WIDTH + tx; // global x index
    float sum = 0.0F;
    for (register int m = 0; m < width / TILE_WIDTH; ++m) {
        // read into the shared memory blocks
        s_A[ty][tx] = g_A[gy * width + (m * TILE_WIDTH + tx)];
        s_B[ty][tx] = g_B[(m * TILE_WIDTH + ty) * width + gx];
        __syncthreads();
        // use the shared memory blocks to get the partial sum
        for (register int k = 0; k < TILE_WIDTH; ++k) {
            sum += s_A[ty][k] * s_B[k][tx];
        }
        __syncthreads();
    }
    g_C[gy * width + gx] = sum;
}
```



Use of Barriers in mat_mul

- Two barriers per phase:
  - __syncthreads() after all data is loaded into __shared__ memory
  - __syncthreads() after all data is read from __shared__ memory

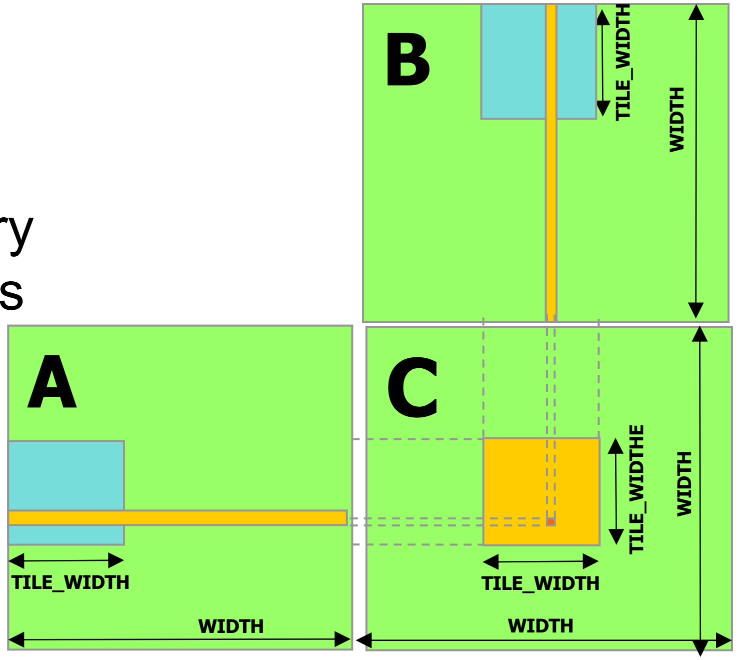
```
__global__ void matmul(float* g_C, const float* g_A, const float* g_B, const int width) {
    __shared__ float s_A[TILE_WIDTH][TILE_WIDTH];
    __shared__ float s_B[TILE_WIDTH][TILE_WIDTH];
      ….
    for (register int m = 0; m < width / TILE_WIDTH; ++m) {
        // read into the shared memory blocks
        s_A[ty][tx] = g_A[gy * width + (m * TILE_WIDTH + tx)];
        s_B[ty][tx] = g_B[(m * TILE_WIDTH + ty) * width + gx];
        __syncthreads();
        // use the shared memory blocks to get the partial sum
        for (register int k = 0; k < TILE_WIDTH; ++k) {
            sum += s_A[ty][k] * s_B[k][tx];
        }
        __syncthreads();
    }
    g_C[gy * width + gx] = sum;
}
```

Block Size Consideration

- **Each thread block should have many threads**
  - TILE_WIDTH of 16 gives 16*16 = 256 threads
  - TILE_WIDTH of 32 gives 32*32 = 1024 threads

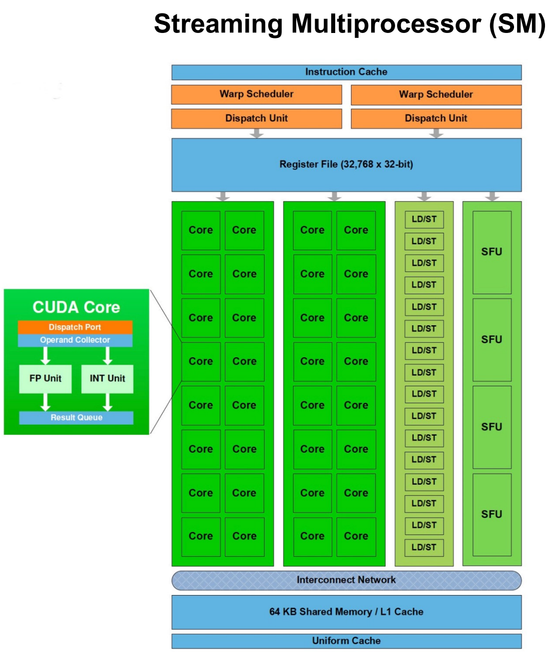
- For 16, each block performs
  2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 operations (mul & add).
  - 8,192 op / 512 load = 16 op / load

- For 32, each block performs
  2*1024 = 2048 float loads from global memory
  for 1024 * (2*32) = 65,536 mul/add operations
  - 65,536 op / 2,048 load = 32 op / load



Block Size Consideration (cont'd)

- Suppose each SM has 48KB shared memory
  - Shared memory size is implementation dependent!
  - For TILE_WIDTH = 16,
    - each thread block uses 2*256*4B = 2KB of shared memory
    - 2 tiles, 256 elements in a tile, element size is 4B
    - SM potentially have up to 24 active thread blocks
      - This allows up to 24 * 2 * 256 = 12K pending loads (2 per thread, 256 threads per block)
  - For TILE_WIDTH = 32,
    - each thread blocks uses 2*32*32*4B= 8KB of shares memory
    - 2 tiles, 32*32 elements in a tile, element size is 4B
    - SM potentially have up to 6 active thread blocks
      - This allows up to 6 * 2 * 1024 = 12K pending loads. (2 per thread, 1,024 threads per block)



matmul-shared.cu (cont'd)

```
#include <cstdio>
#include <stdlib.h> // for rand(), malloc(), free()
#include "common.h"
#include <sys/time.h>


//CUDA kernel size settings

const int  WIDTH =                                    1024;           // total width is 1024*1024
const int  TILE_WIDTH =    32;                 // block will be (TILE_WIDTH,TILE_WIDTH)
const int  GRID_WIDTH = (WIDTH / TILE_WIDTH);  // grid will be (GRID_WDITH,GRID_WDITH)


//random data generation
void genData(float* ptr, unsigned int size) {
   while (size--) {
       *ptr++ = (float)(rand() % 1000) / 1000.0F;
   }
}

.....
```

matmul-shared.cu (cont'd)

```
__global__ void matmul(float* g_C, const float* g_A, const float* g_B, const int width) {
   __shared__ float s_A[TILE_WIDTH][TILE_WIDTH];
   __shared__ float s_B[TILE_WIDTH][TILE_WIDTH];
   int by = blockIdx.y; int bx = blockIdx.x;
   int ty = threadIdx.y; int tx = threadIdx.x;
   int gy = by * TILE_WIDTH + ty; // global y index
   int gx = bx * TILE_WIDTH + tx; // global x index
   float sum = 0.0F;
   for (register int m = 0; m < width / TILE_WIDTH; ++m) {
      // read into the shared memory blocks
      s_A[ty][tx] = g_A[gy * width + (m * TILE_WIDTH + tx)];
      s_B[ty][tx] = g_B[(m * TILE_WIDTH + ty) * width + gx];
      __syncthreads();
      // use the shared memory blocks to get the partial sum
      for (register int k = 0; k < TILE_WIDTH; ++k) {
         sum += s_A[ty][k] * s_B[k][tx];
      }
      __syncthreads();
   }
   g_C[gy * width + gx] = sum;
}
```

matmul-shared.cu (cont'd)

```
int main(void) {
   float* pA = NULL;
   float* pB = NULL;
   float* pC = NULL;
   struct timeval start_time, end_time;

    // malloc memories on the host-side
    pA = (float*)malloc(WIDTH * WIDTH * sizeof(float));
    pB = (float*)malloc(WIDTH * WIDTH * sizeof(float));
    pC = (float*)malloc(WIDTH * WIDTH * sizeof(float));
    // generate source data
    genData(pA, WIDTH * WIDTH);
    genData(pB, WIDTH * WIDTH);

    // CUDA: allocate device memory
    float* pAdev = NULL;
    float* pBdev = NULL;
    float* pCdev = NULL;
    CUDA_CHECK( cudaMalloc((void**)&pAdev, WIDTH * WIDTH * sizeof(float)) );
    CUDA_CHECK( cudaMalloc((void**)&pBdev, WIDTH * WIDTH * sizeof(float)) );
    CUDA_CHECK( cudaMalloc((void**)&pCdev, WIDTH * WIDTH * sizeof(float)) );


    // copy from host to device
    CUDA_CHECK( cudaMemcpy(pAdev, pA, WIDTH * WIDTH * sizeof(float),
                                                      cudaMemcpyHostToDevice) );
    CUDA_CHECK( cudaMemcpy(pBdev, pB, WIDTH * WIDTH * sizeof(float),
                                                      cudaMemcpyHostToDevice) );
```

matmul-shared.cu (cont'd)

```
    //get current time
    cudaThreadSynchronize();
    gettimeofday(&start_time, NULL);
   // CUDA: launch the kernel
    dim3 dimGrid(GRID_WIDTH, GRID_WIDTH, 1);
    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
    matmul <<< dimGrid, dimBlock>>>(pCdev, pAdev, pBdev, WIDTH);
    CUDA_CHECK( cudaPeekAtLastError() );
    //get current time
    cudaThreadSynchronize();
    gettimeofday(&start_time, NULL);
    double operating_time = (double)(end_time.tv_sec)+(double)(end_time.tv_usec)/1000000.0-
((double)(start_time.tv_sec)+(double)(start_time.tv_usec)/1000000.0);
    printf("Elapsed: %f seconds\n", (double)operating_time);

    // copy from device to host
    CUDA_CHECK( cudaMemcpy(pC, pCdev, WIDTH * WIDTH * sizeof(float), cudaMemcpyDeviceToHost) );
    // free device memory
    CUDA_CHECK( cudaFree(pAdev) );
    CUDA_CHECK( cudaFree(pBdev) );
    CUDA_CHECK( cudaFree(pCdev) );
    // print sample cases
    int i, j;
    i = 0; j = 0;
   printf("c[%4d][%4d] = %f\n", i, j, pC[i * WIDTH + j]);
                  i = WIDTH / 2; j = WIDTH / 2;
   printf("c[%4d][%4d] = %f\n", i, j, pC[i * WIDTH + j]);
                  i = WIDTH - 1; j = WIDTH - 1;
   printf("c[%4d][%4d] = %f\n", i, j, pC[i * WIDTH + j]);

   // done
    return 0;
}
```

Comparisons

- Matrix multiplication with CPU
  - 6.7 sec
- Tiled matrix multiplication kernel
  - 0.00676 sec
- Tiled matrix multiplication kernel using shared memory
  - 0.005275 sec

Agenda

- **Matrix Multiplication**
  - **Basic Version**
  - **Tiled Version**

- Review: Memory Hierarchy
- Importance of Memory Access Efficiency

- **GPU Memory Hierarchy**
- **Improving Tiled Matrix Multiplication**
- **Impact of Memory on Parallelism**

Memory Resources as Limit to Parallelism

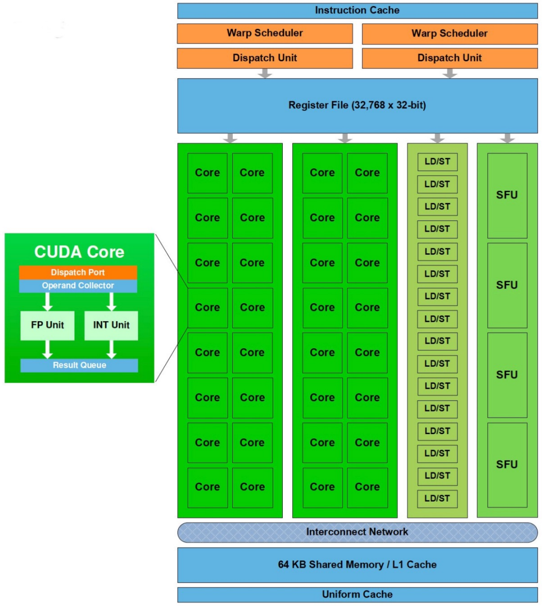
- Effective use of different memory resources reduces the number of accesses to global memory
- **But, these resources are finite!**
- More memory space each thread requires

  → the fewer threads an SM can accommodate

  → the fewer threads run concurrently on an SM

  → fewer number of warps available for scheduling

  → decreasing the ability of the processor to find useful work to hide long-latency operations



Memory Resources as Limit to Parallelism

- **The number of registers per thread limits the level of parallelism**
  - ex) SM can accommodate up to 1536 threads and have 16384 registers
    - To support 1536 threads, each thread can use only 10 registers (=16384/1536)
    - If each thread uses 11 registers, the number of threads concurrently executed in each SM will be reduced
    - If each block contains 512 threads, two blocks can be concurrently executed → Only 1024 threads will be executed concurrently → **low occupancy**

- **Shared memory usage per block limits the level of parallelism**
  - ex) SM can accommodate up to 8 blocks, 1536 threads, and have 16KB shared memory
    - If each block uses 3KB of shared memory, no more than 5 blocks can be concurrently executed in each SM.
      - 5*3KB= 15KB of the shared memory will be used
    - If each block contains 256 threads and uses 2KB of shared memory, no more than 6 blocks can be concurrently executed in each SM
      - 6*2KB = 12KB of the shared memory will be used

Next?

- **Performance Consideration**
  - More on Memory Parallelism
  - Warps and SIMD Hardware
  - Dynamic Partitioning of Resources
  - ..