

并行与分布式计算

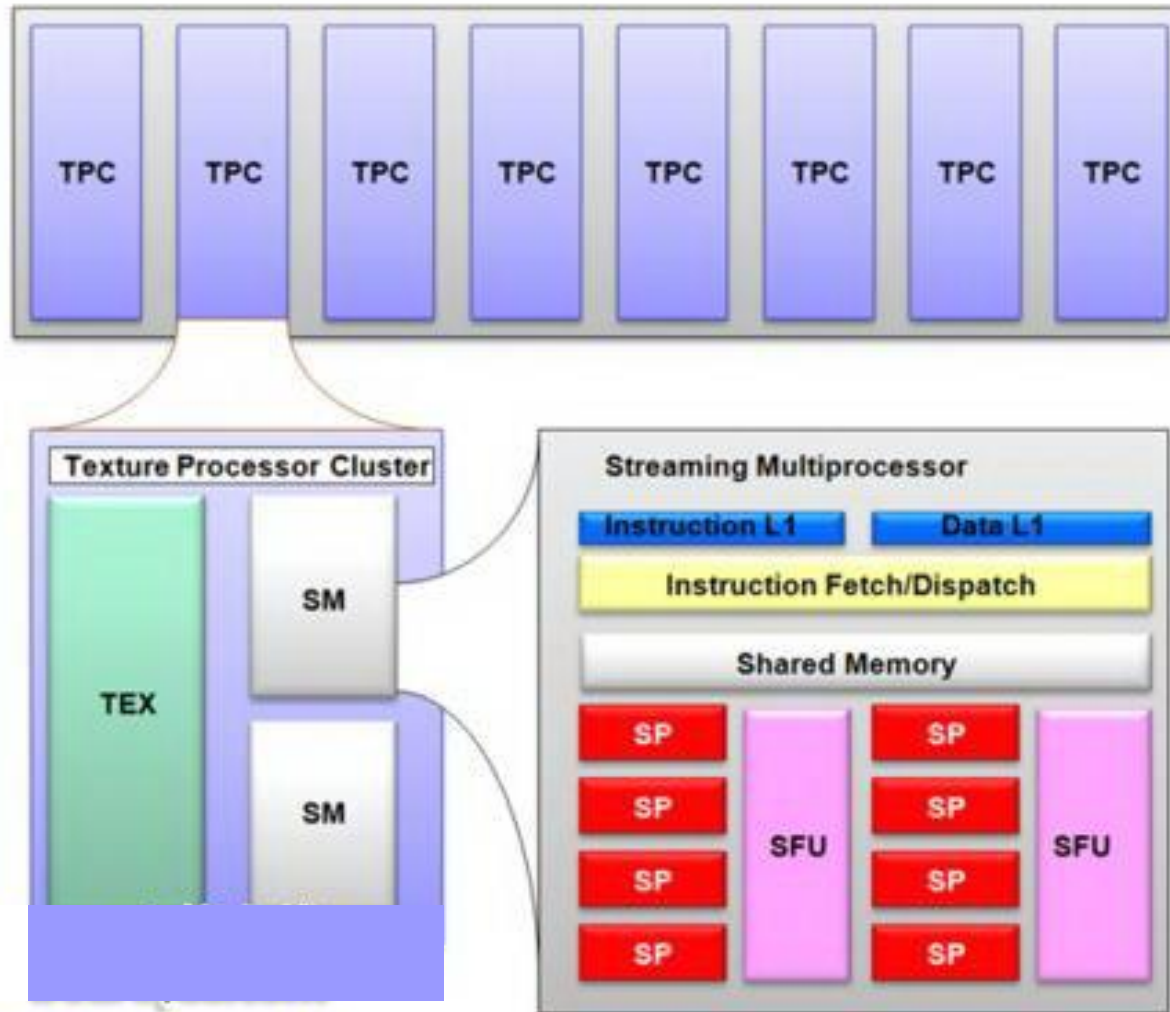
Ying Liu, Prof., Ph.D

School of Computer Science and Technology
University of Chinese Academy of Sciences
Data Mining and High Performance Computing Lab

CUDA Memory

- Memory model
- Programming using global memory, shared memory

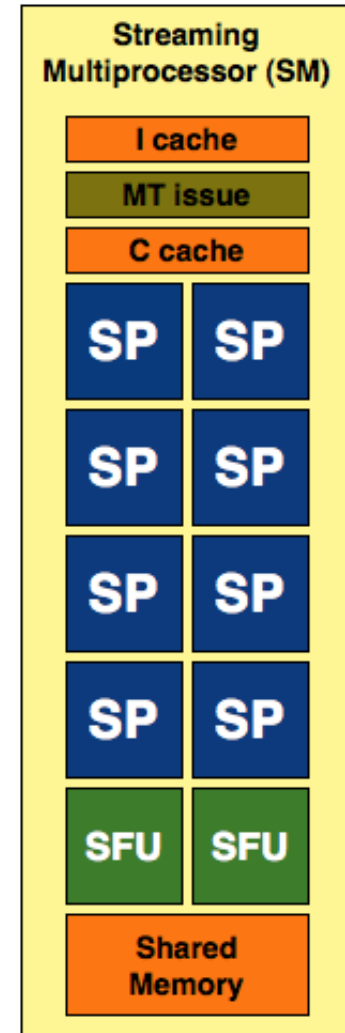
G80 CUDA Mode



Streaming Multiprocessor (SM)

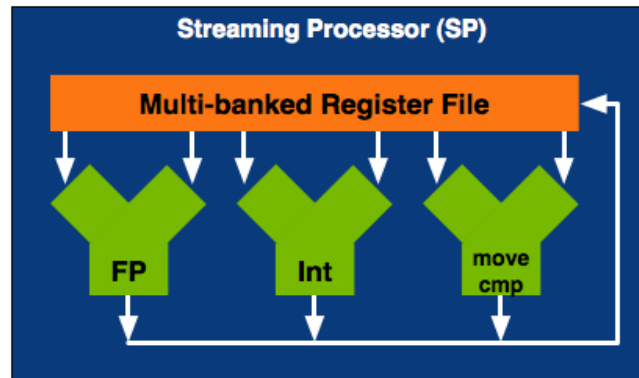
■ An array of SPs

- 8 streaming processors
- 2 Special Function Units (SFU)
 - Transcendental operations (e.g. sin, cos) and interpolation
- A 16KB read/write shared memory
 - Not a cache, but a software-managed data store
- Multithreading issuing unit
 - Dispatch instructions
- Instruction cache
- Constant cache



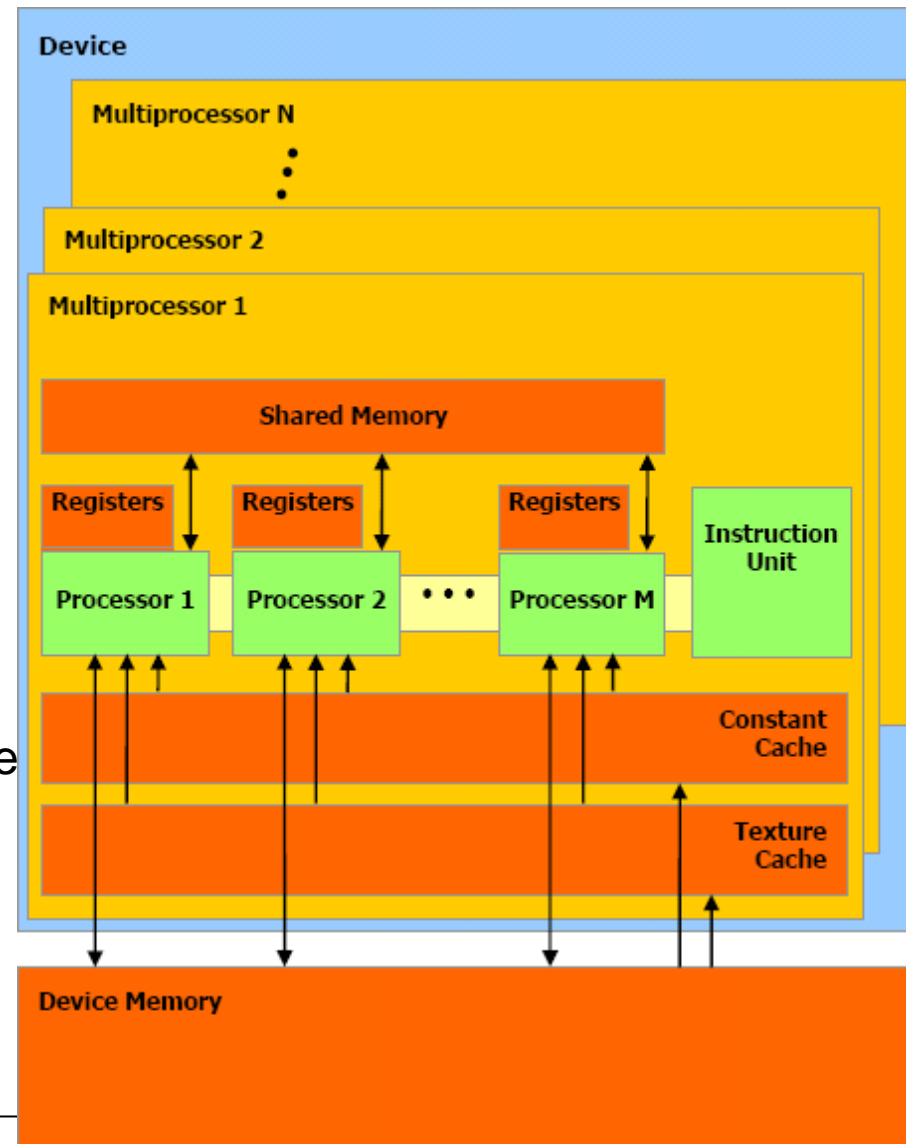
Streaming Processor (SP)

- A fully pipelined, single-issue, in-order microprocessor
 - 2 ALUs and a FPU
 - Register file
 - 32-bit processing
 - No cache



Memory Hierarchy

- R/W per-thread **registers**
 - 1-cycle latency
- R/W per-thread **local memory**
 - Slow – register spilling to global memory
- R/W per-block **shared memory**
 - 1-cycle latency
 - But bank conflicts may drag down
- R/W per-grid **global memory**
 - ~500-cycle latency
 - But coalescing accessing could hide latency
- Read only per-grid **constant and texture memories**
 - ~500-cycle latency
 - But cached



Registers

- Scope — per-thread
- Fast — 1-cycle latency
- Delay when R/W dependencies or register memory bank conflicts
- The delay can be ignored when ≥ 192 threads active
- Users have no control over the bank conflicts
- Best results achieved when the number of threads per block is 64X
- No access from/to the host

Registers

- Read-after-write dependency
 - Instruction's result can be read ~22 cycles later
 - Scenarios:
 - CUDA via PTX



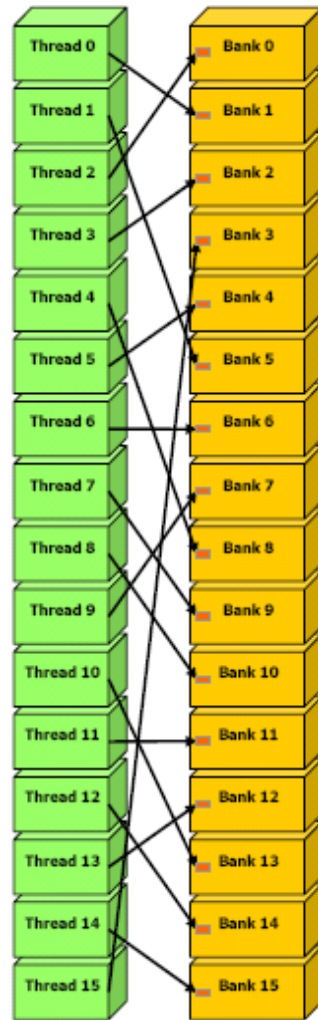
Local Memory

- Scope — per-thread
- Global memory
- Store 'registers' in global memory when running out of registers
- For array accesses if the compiler cannot determine
- Slow — as global memory
- No access from/to the host
- 16 banks of 32-bit words

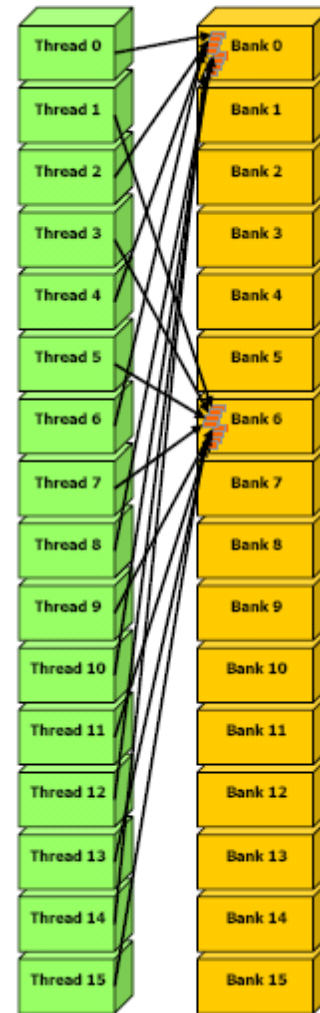
Shared Memory

- On-chip
- Scope — per-block
- Fast — 1-cycle latency
- Delay when bank conflict occurs
 - Memory is divided into equally-sized banks
 - Request to n ***distinct*** memory banks can be served simultaneously
 - Hardware serialized bank conflicts into separate conflict-free requests, thus reducing the bandwidth
- Minimize bank conflicts
- No access from/to the host

Shared Memory (Cont.)



Access without bank conflicts



Access with bank conflicts

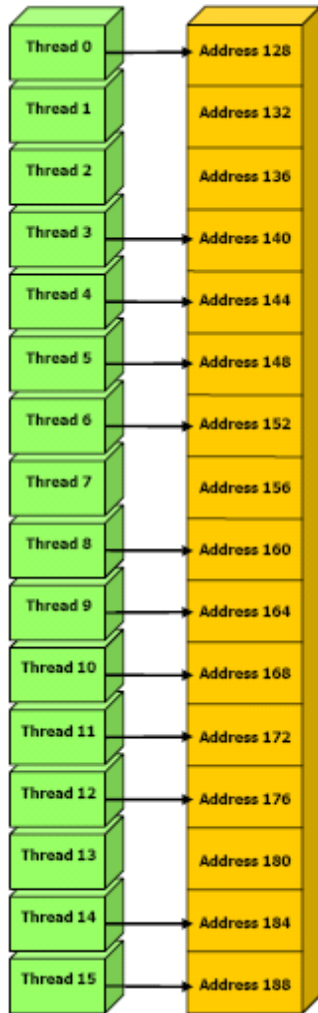
Shared Memory (Cont.)

- As fast as registers when no bank conflicts
- The fast case:
 - If all threads of a half-warp access different banks, there is no bank conflict
 - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)
- The slow case:
 - Bank conflict: multiple threads in the same half-warp access the same bank
 - Must serialize the accesses

Global Memory

- Device memory
- Scope — per-grid
- ~500-cycle latency
- Maximize bandwidth
 - **Coalescing** accessing could hide latency
 - Use the alignment specifiers `_align_(8)` or `_align_(16)` to adjust the size of any struct
- Accessible from/to the host

Global Memory (Cont.)



Coalesced Access



Non-coalesced Access

Constant Memory

- Scope — per-grid
- Global memory
- Accessible from/to the host
- Associated with on-chip cache
 - Slow case: 1 cache miss, incurring 1 memory read from the global memory
 - Fast case: no cache miss, as fast as registers

Texture Memory

- Scope — per-grid
- Global memory
- Accessible from/to the host
- Associated with on-chip cache
 - Slow case: 1 cache miss, incurring 1 memory read from the global memory
 - Fast case: no cache miss, as fast as registers
- The texture cache is optimized for 2D spatial locality

Host-Device Data Transfer

- `cudaMemcpy()`
 - Memory data transfer
 - Requires four parameters
 - Pointer to destination
 - Pointer to source
 - Number of bytes copied
 - Type of transfer
 - Host to Host
 - Host to Device
 - Device to Host
 - Device to Device

CUDA Memory

- Memory model
- Programming using global memory, shared memory

Typical Programming Model

- Transfer data from host to device memory
- Load data from device memory to shared memory
- Synchronize all the threads of the block
- Process the data in shared memory
- Synchronize again if necessary to make sure that shared memory has been updated with the results
- Write the results back to device memory
- Transfer results from device to host

Typical Programming Model (Cont.)

- Constant memory resides in device memory (DRAM) - much slower access than shared memory
 - But... cached!
 - Highly efficient access for read-only data
- Carefully divide data according to access patterns
 - R/Only → constant memory (very fast if in cache)
 - R/W shared within block → shared memory (very fast)
 - R/W within each thread → registers (very fast)
 - R/W inputs/results → global memory (very slow)

Matrix Multiplication

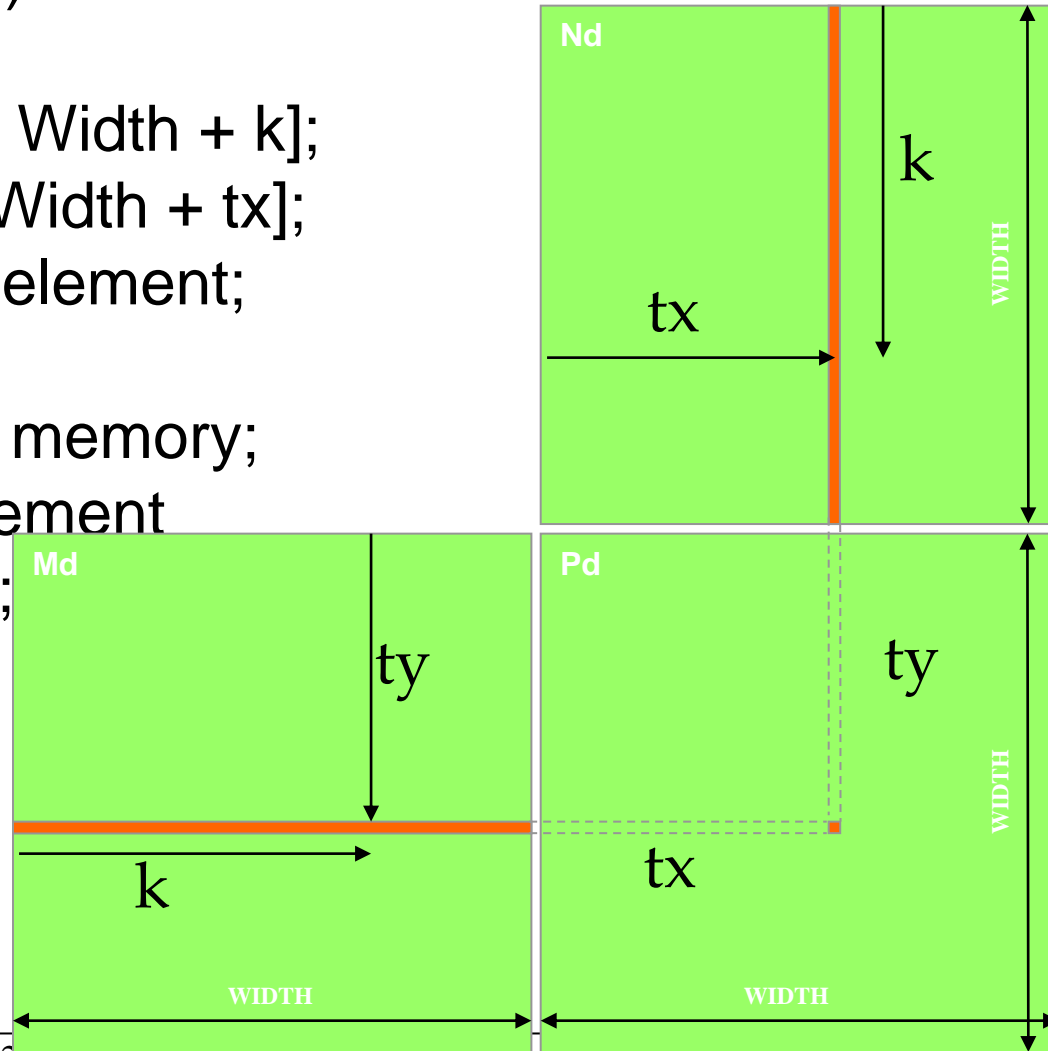
- **Tile data** to take advantage of fast shared memory
 - **Partition** data into **subsets** that fit into shared memory
 - Handle **each data subset with one thread block** by:
 - Loading the subset from global memory to shared memory
 - **Using multiple threads to exploit memory-level parallelism**
 - Performing the computation on the subset from shared memory
 - Each thread efficiently multi-pass over any data element
 - Copying results from shared memory to global memory

GeForce 8800 GTX Configuration

# stream processor	128
# stream multiprocessor	16
# registers per SM	8192 (32KB)
# threads per block	Up to 512
# threads per SM	Up to 768
# blocks per SM	Up to 8
# blocks per grid	Up to 65535 each dimension
global memory	768MB
constant memory	64KB
shared memory per SM	16KB
peak	346.5 GFlops/s
memory bandwidth	86.4 GB/s

Step 4: Kernel Function (Cont.)

```
for (int k = 0; k < Width; ++k)
{
    float Melement = Md[ty * Width + k];
    float Nelement = Nd[k * Width + tx];
    Pvalue += Melement * Nelement;
}
// Write the matrix to device memory;
// each thread writes one element
Pd[ty * Width + tx] = Pvalue;
}
```

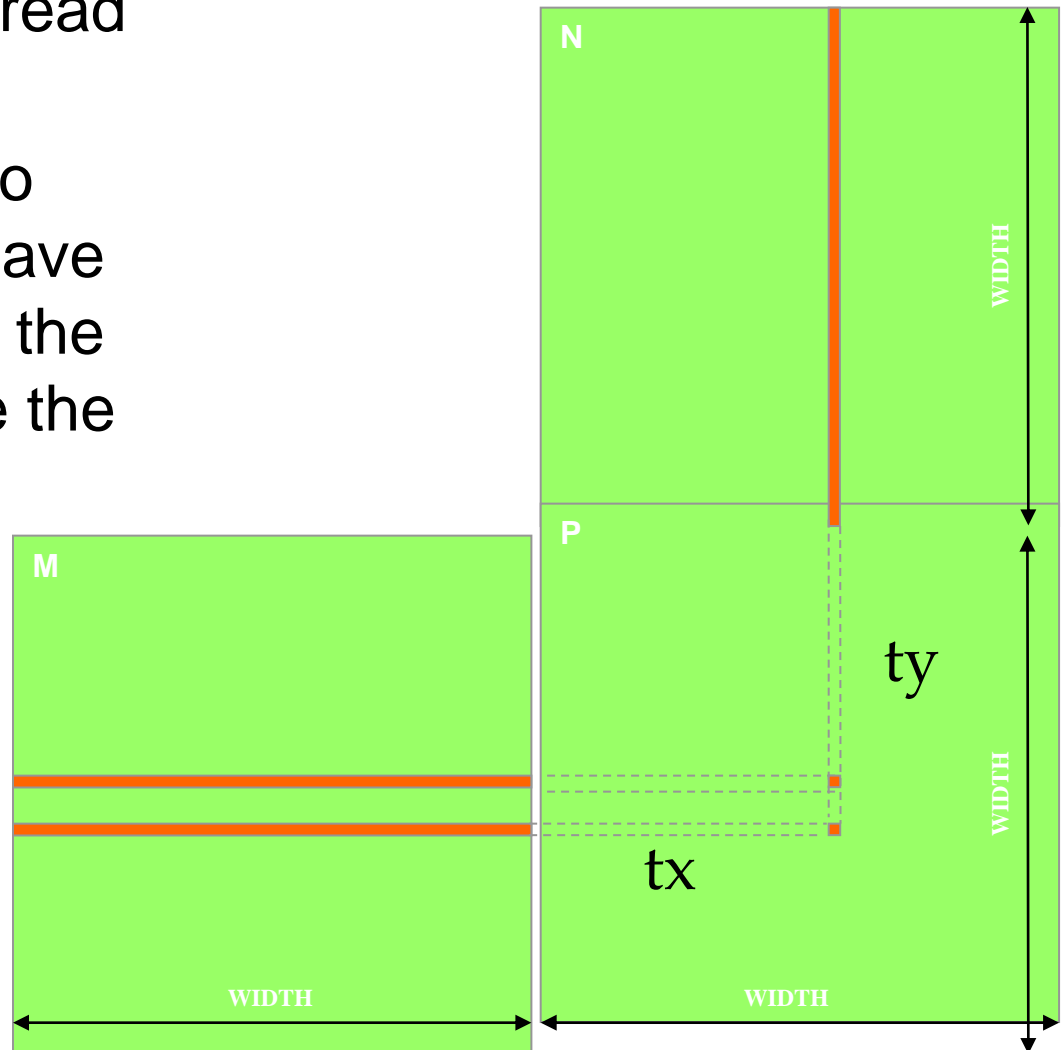


Performance on G80

- All threads access global memory for their input matrix elements
 - Two memory accesses (8 bytes) per floating point multiply-add
 - 86.4 GB/s limits the code at 21.6 GFLOPS
 - 346.5 GFLOPS peak FLOP rating
- The actual code runs at about 15 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak 346.5 GFLOPS

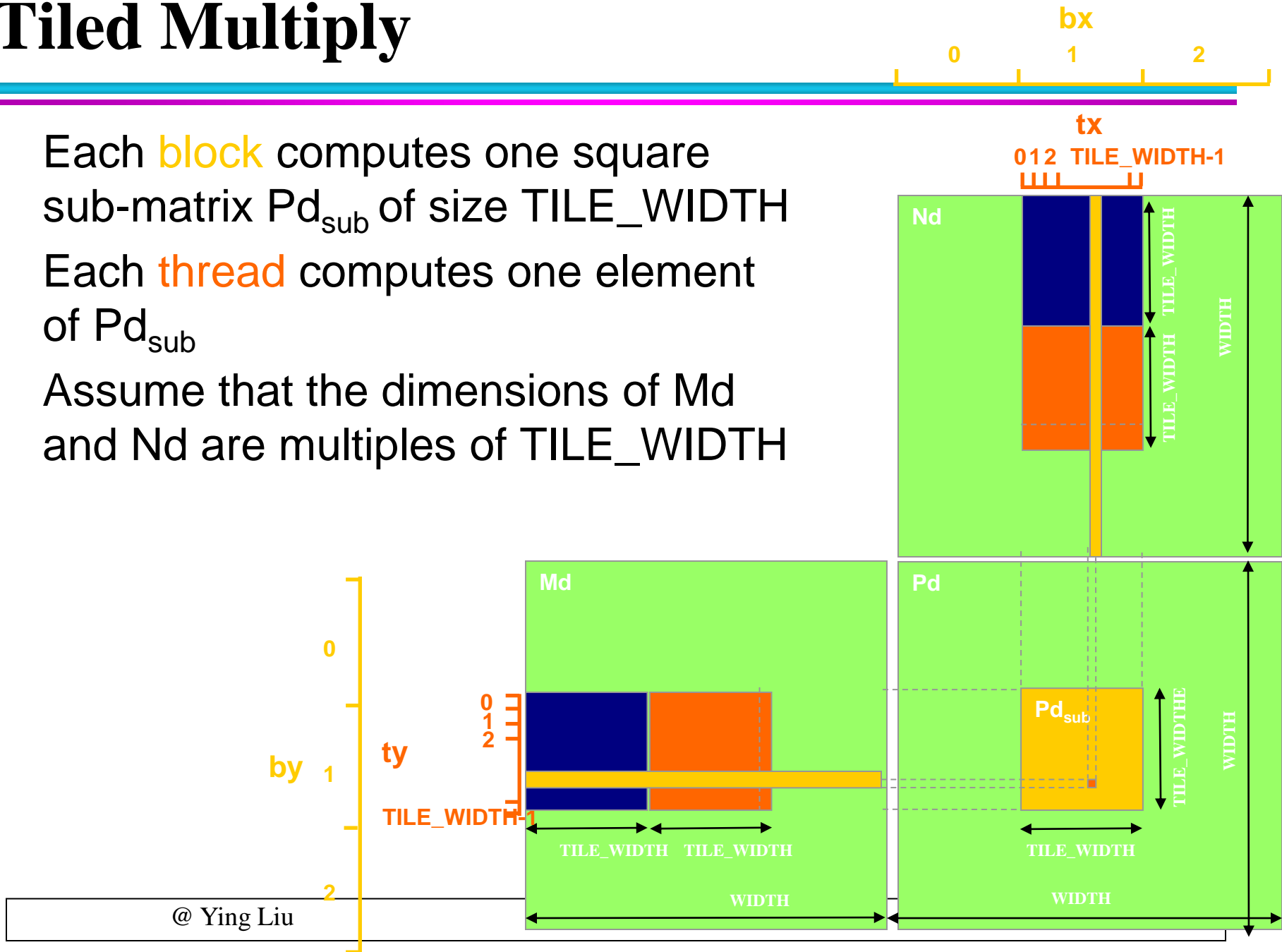
Use Shared Memory to Reuse Global Memory Data

- Each input element is read by WIDTH threads
- Load each element into Shared Memory and have several threads to use the local version to reduce the memory bandwidth
 - Tiled algorithms



Tiled Multiply

- Each **block** computes one square sub-matrix Pd_{sub} of size $TILE_WIDTH$
- Each **thread** computes one element of Pd_{sub}
- Assume that the dimensions of Md and Nd are multiples of $TILE_WIDTH$



First-order Size Considerations in G80

- Each **thread block** should have many threads
 - TILE_WIDTH of 16 gives $16 \times 16 = 256$ threads
- There should be many thread blocks
 - A 1024×1024 Pd gives $64 \times 64 = 4096$ Thread Blocks
- Each thread block perform $2 \times 256 = 512$ float loads from global memory for $256 * (2 \times 16) = 8,192$ mul/add operations
 - Memory bandwidth no longer a limiting factor

CUDA Code – Kernel Execution Configuration

```
// Setup the execution configuration  
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);  
dim3 dimGrid(Width / TILE_WIDTH, Width / TILE_WIDTH);
```

CUDA Code – Kernel Overview

```
// Block index
```

```
int bx = blockIdx.x;
```

```
int by = blockIdx.y;
```

```
// Thread index
```

```
int tx = threadIdx.x;
```

```
int ty = threadIdx.y;
```

```
// Pvalue stores the element of the block sub-matrix
```

```
// that is computed by the thread – automatic variable!
```

```
float Pvalue = 0;
```

```
// Loop over all the sub-matrices of M and N
```

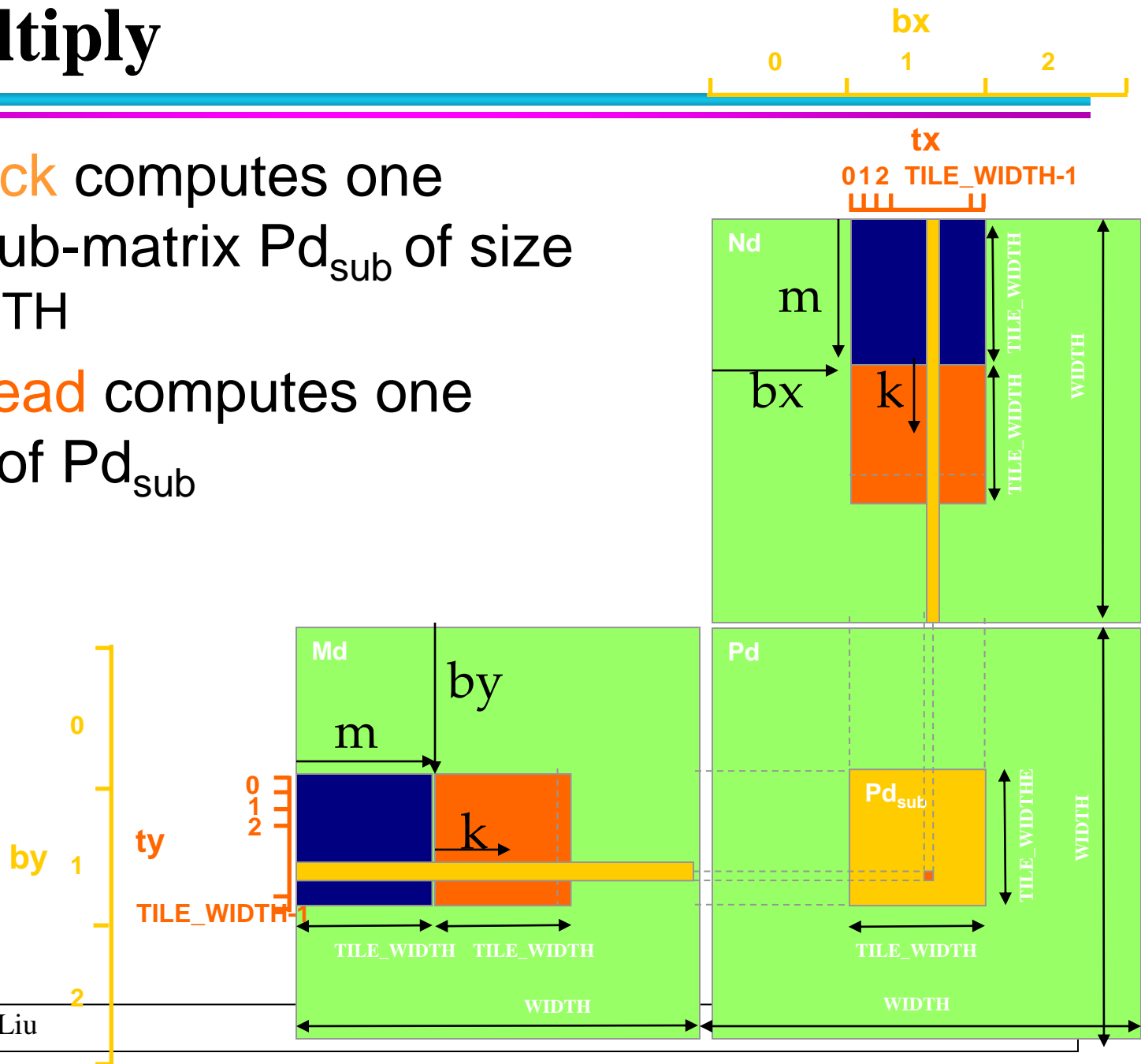
```
// required to compute the block sub-matrix
```

```
for (int m = 0; m < Width/TILE_WIDTH; ++m) {
```

```
    code from the next few slides };
```

Tiled Multiply

- Each **block** computes one square sub-matrix Pd_{sub} of size TILE_WIDTH
- Each **thread** computes one element of Pd_{sub}



CUDA Code - Load Data to Shared Memory

// Get a pointer to the current sub-matrix Mdsb of Md

```
float* Mdsb = GetSubMatrix(Md, m, by, Width);
```

// Get a pointer to the current sub-matrix Ndsb of Nd

```
float* Ndsb = GetSubMatrix(Nd, bx, m, Width);
```

GetSubMatrix(Md, x, y, Width)

- Md + y*TILE_WIDTH*Width + x*TILE_WIDTH)

Tying Up Some Loose Ends

```
__shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
```

```
__shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
```

```
// each thread loads one element of the sub-matrix  
Mds[ty][tx] = GetMatrixElement(Mdsub, tx, ty);
```

```
// each thread loads one element of the sub-matrix  
Nds[ty][tx] = GetMatrixElement(Ndsub, tx, ty);
```

```
GetMatrixElement(Mdsub,tx,ty)
```

- `*(Mdsub+ty*Width+tx);`

CUDA Code - Compute Result

```
// Synchronize to make sure the sub-matrices are loaded
// before starting the computation
__syncthreads();

// each thread computes one element of the block sub-matrix
for (int k = 0; k < TILE_WIDTH; ++k)
    Pvalue += Mds[ty][k] * Nds[k][tx];

// Synchronize to make sure that the preceding computation is done
// before loading two new sub-matrices of M and N in the next iteration
__syncthreads();
```

CUDA Code - Save Result

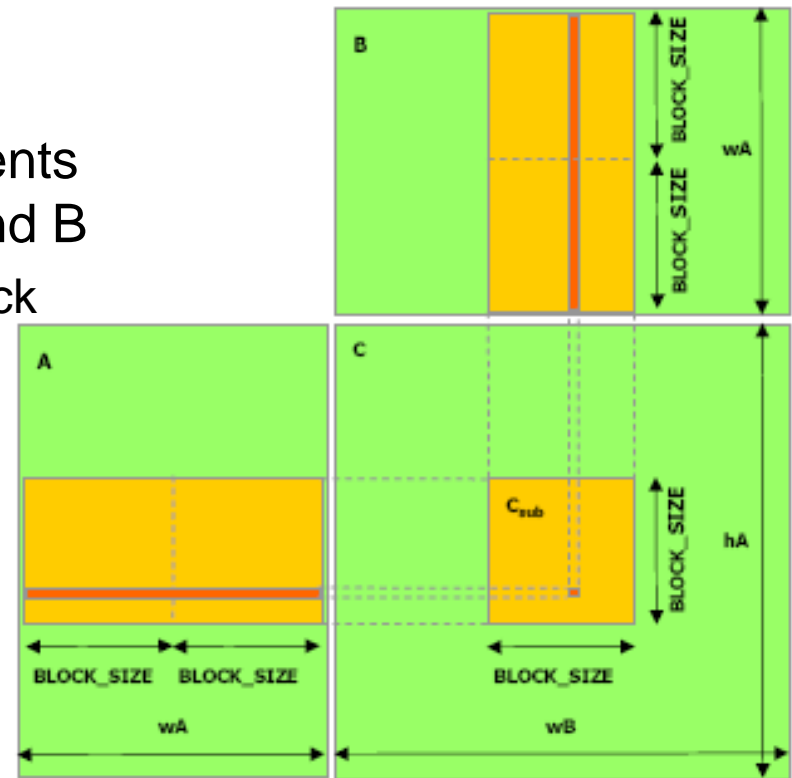
```
// Get a pointer to the block sub-matrix of Pd
float* Pdsb = GetSubMatrix(Pd, bx, by, Width);

// Write the block sub-matrix to device memory
// each thread writes one element
SetMatrixElement(Pdsb, tx, ty, Pvalue);
    Pdsb[ty*Width+tx] = Pvalue;
```

This code runs at about 45 GFLOPS on G80

Arbitrary Shaped Matrix Multiplication

- A 2D thread block
 - Computes a $\text{BLOCK_SIZE} \times \text{BLOCK_SIZE}$ sub-matrix of the result matrix
 - Each has $(\text{BLOCK_SIZE})^2$ threads
 - Use shared memory to store elements in the corresponding blocks of A and B
 - Shared by all threads in thread block
- Generate a 2D grid of $\text{HA}/\text{BLOCK_SIZE} \times \text{WB}/\text{BLOCK_SIZE}$ blocks



CUDA Implementation - Host

```
// Thread block size
#define BLOCK_SIZE 16
// Forward declaration of the device multiplication function
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
void Mul(const Matrix A, const Matrix B, Matrix C) {
    // Load A and B to the device
    float *Ad, *Bd;
    size = A.height * A.width * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    size = B.height * B.width * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
```

CUDA Implementation - Host

```
// Allocate P on the device
```

```
float* Cd;
```

```
size = A.height * B.width * sizeof(float);
```

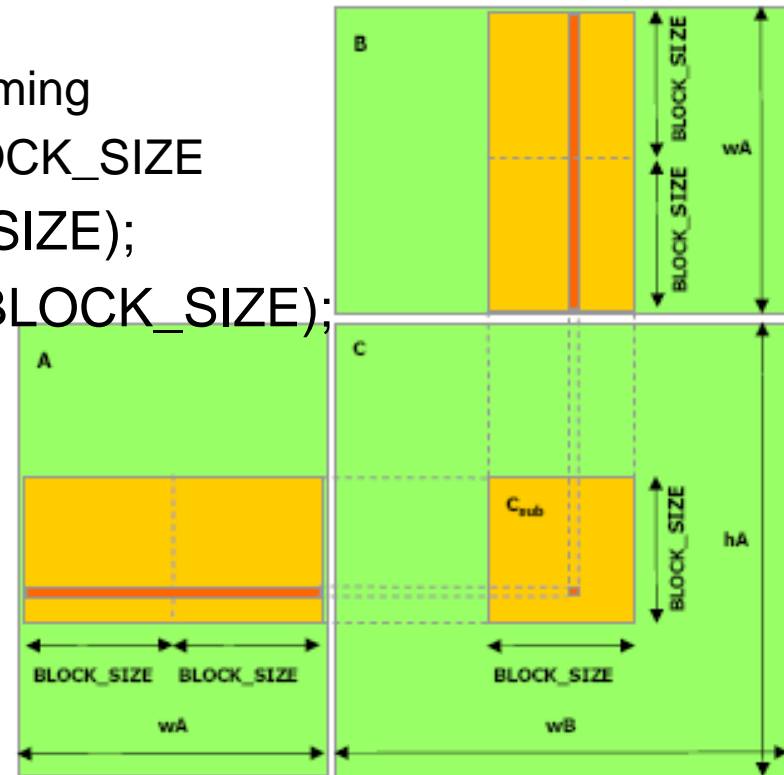
```
cudaMalloc((void**)&Cd, size);
```

```
// Compute the execution configuration assuming
```

```
// the matrix dimensions are multiples of BLOCK_SIZE
```

```
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
```

```
dim3 dimGrid(wB / BLOCK_SIZE, hA / BLOCK_SIZE);
```



CUDA Implementation - Host

```
// Launch the device computation
Muld<<<dimGrid, dimBlock>>>(Ad, Bd, A.width, B.width, Cd);

// Read C from the device
cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(Ad);
cudaFree(Bd);
cudaFree(Cd);
}
```

CUDA Implementation - Kernel

```
// Device multiplication function called by Mul()
// Compute  $C = A \times B$ 
// wA is the width of A and wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;
```

CUDA Implementation - Kernel

```
// Index of the first sub-matrix of A processed  
// by the block
```

```
int aBegin = wA * BLOCK_SIZE * by;
```

```
// Index of the last sub-matrix of A processed  
// by the block
```

```
int aEnd = aBegin + wA - 1;
```

```
// Step size used to iterate through the sub-  
// matrices of A
```

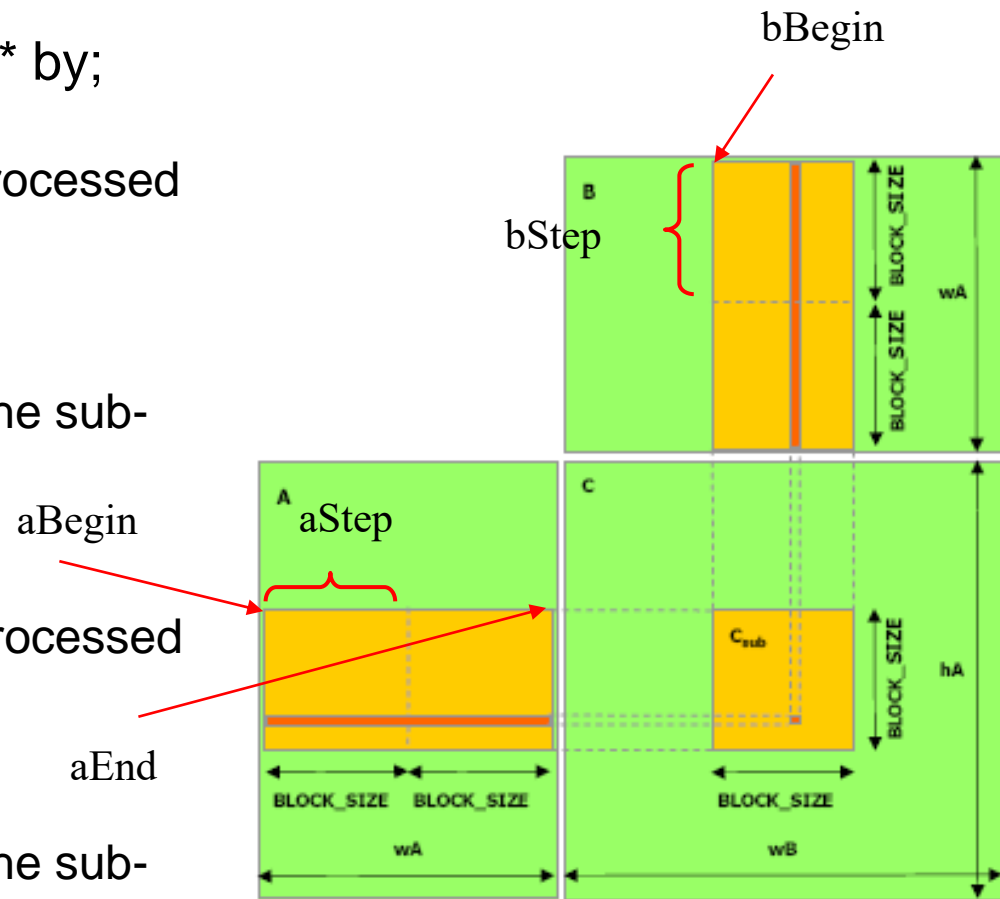
```
int aStep = BLOCK_SIZE;
```

```
// Index of the first sub-matrix of B processed  
// by the block
```

```
int bBegin = BLOCK_SIZE * bx;
```

```
// Step size used to iterate through the sub-  
// matrices of B
```

```
int bStep = BLOCK_SIZE * wB;
```



CUDA Implementation - Kernel

```
// The element of the block sub-matrix that is
    computed by the thread
```

```
float Csub = 0;
```

```
// Loop over all the sub-matrices of A and B
    required to
```

```
// compute the block sub-matrix
```

```
for (int a = aBegin, b = bBegin; a <=
    aEnd; a += aStep, b += bStep) {
```

```
    // Shared memory for the sub-matrix of A
```

```
    __shared__ float
```

```
    As[BLOCK_SIZE][BLOCK_SIZE];
```

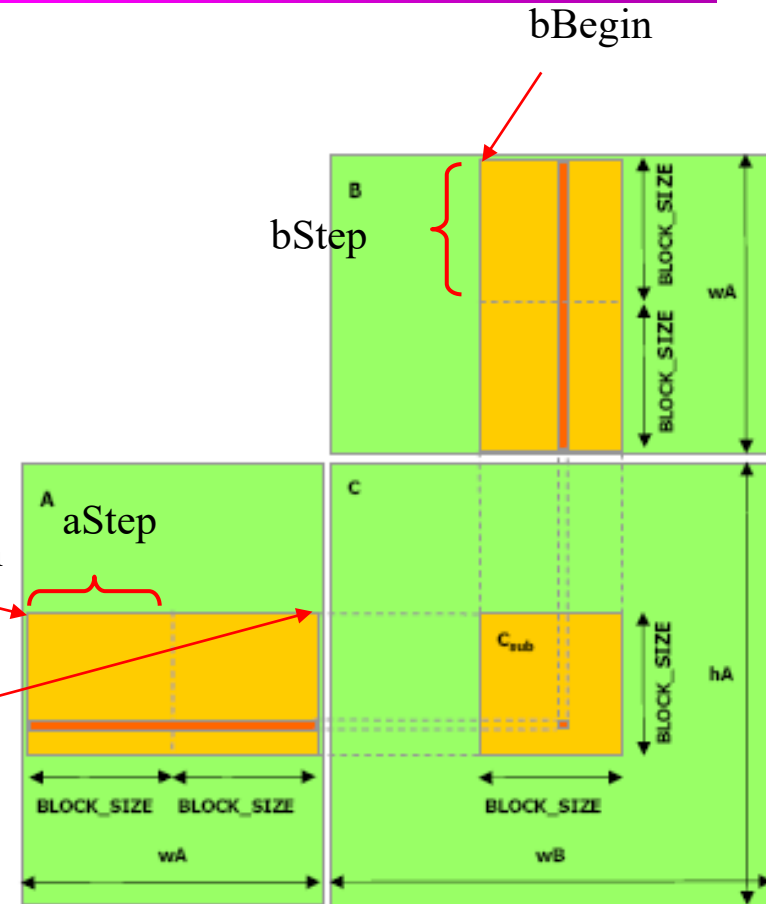
aBegin

aEnd

```
    // Shared memory for the sub-matrix of B
```

```
    __shared__ float
```

```
    Bs[BLOCK_SIZE][BLOCK_SIZE];
```



CUDA Implementation - Kernel

```
// Load the matrices from global memory to shared  
memory;
```

```
// each thread loads one element of each matrix
```

```
As[ty][tx] = A[a + wA * ty + tx];
```

```
Bs[ty][tx] = B[b + wB * ty + tx];
```

```
// Synchronize to make sure the matrices are  
loaded
```

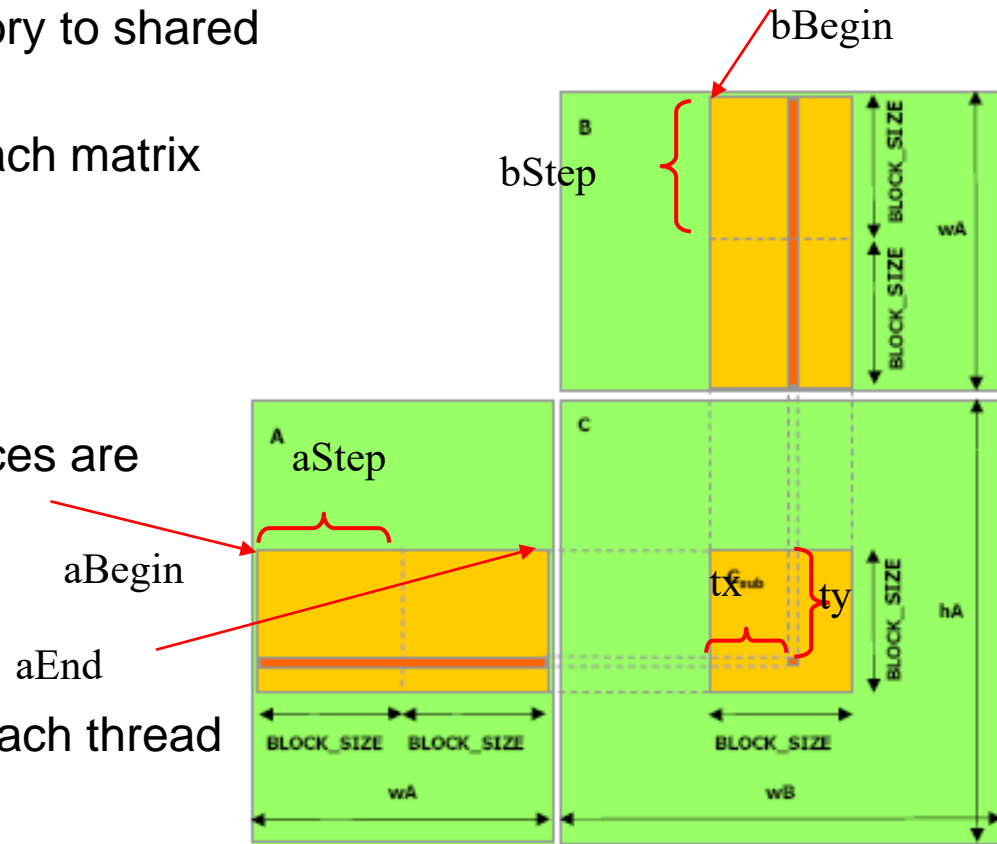
```
__syncthreads();
```

```
// Multiply the two matrices together; each thread  
computes
```

```
// one element of the block sub-matrix
```

```
for (int k = 0; k < BLOCK_SIZE; ++k)
```

```
    Csub += As[ty][k] * Bs[k][tx];
```



CUDA Implementation - Kernel

```
// Synchronize to make sure that the
// preceding computation is done
// before loading two new sub-matrices of
// and B in the next iteration
__syncthreads();
```

```
}
```

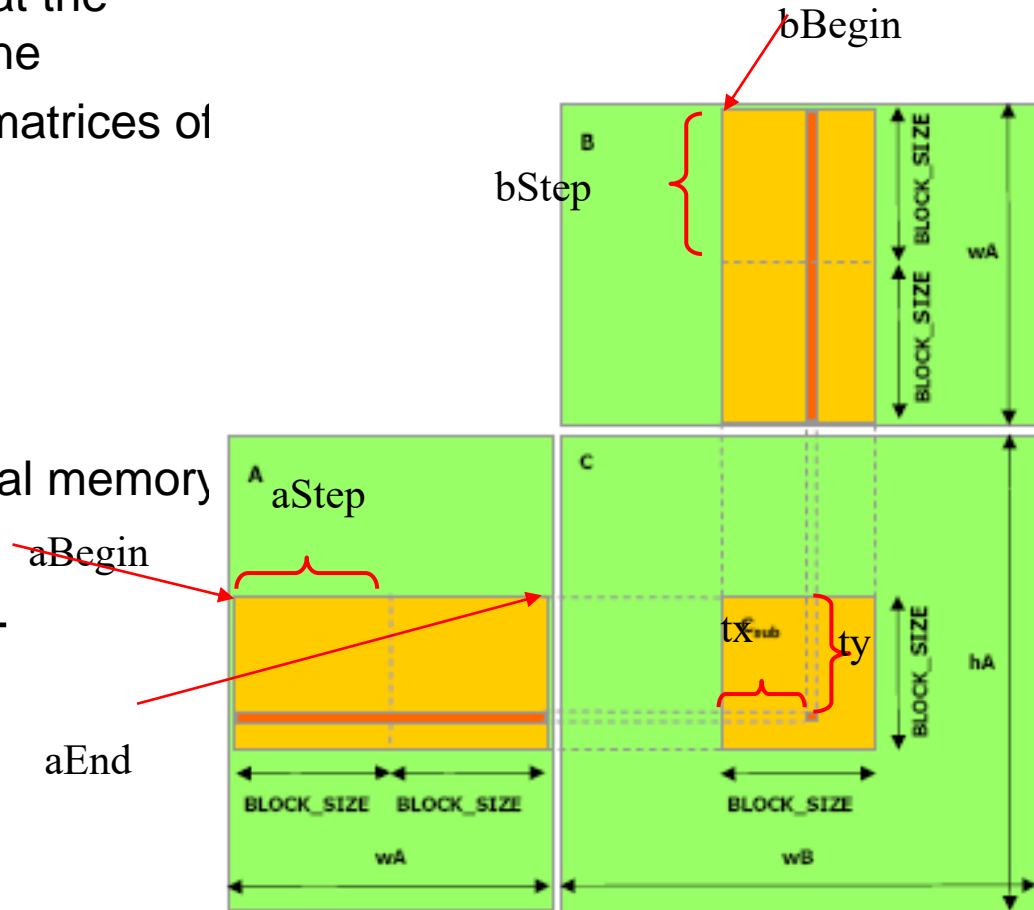
```
// Write the block sub-matrix to global memory
```

```
// each thread writes one element
```

```
int c = wB * BLOCK_SIZE * by +
      BLOCK_SIZE * bx;
```

```
C[c + wB * ty + tx] = Csub;
```

```
}
```



Summary - Typical Structure of a CUDA Program

■ Global variables declaration

- `__host__`
- `__device__... __global__, __constant__, __texture__`

■ Function prototypes

- `__global__ void kernelOne (...)`
- `float handyFunction (...)`

■ Main ()

- allocate memory space on the device – `cudaMalloc (&d_GlblVarPtr, bytes)`
- transfer data from host to device – `cudaMemcpy (d_GlblVarPtr, h_Gl...`
- execution configuration setup
- kernel call – `kernelOne<<<execution configuration>>>(args...);`
- transfer results from device to host – `cudaMemcpy (h_GlblVarPtr,...)`
- optional: compare against golden (host computed) solution



*repeat
as needed*

Summary- Typical Structure of a CUDA Program (Cont.)

- Kernel – void kernelOne (type args,...)
 - variables declaration - `__local__`, `__shared__`
 - automatic variables transparently assigned to registers or local memory
 - `Syncthreads ()`...
- Other functions
 - `float handyFunction (int inVar...);`