CS4803DGC Design and Programming of Game Consoles

Spring 2011

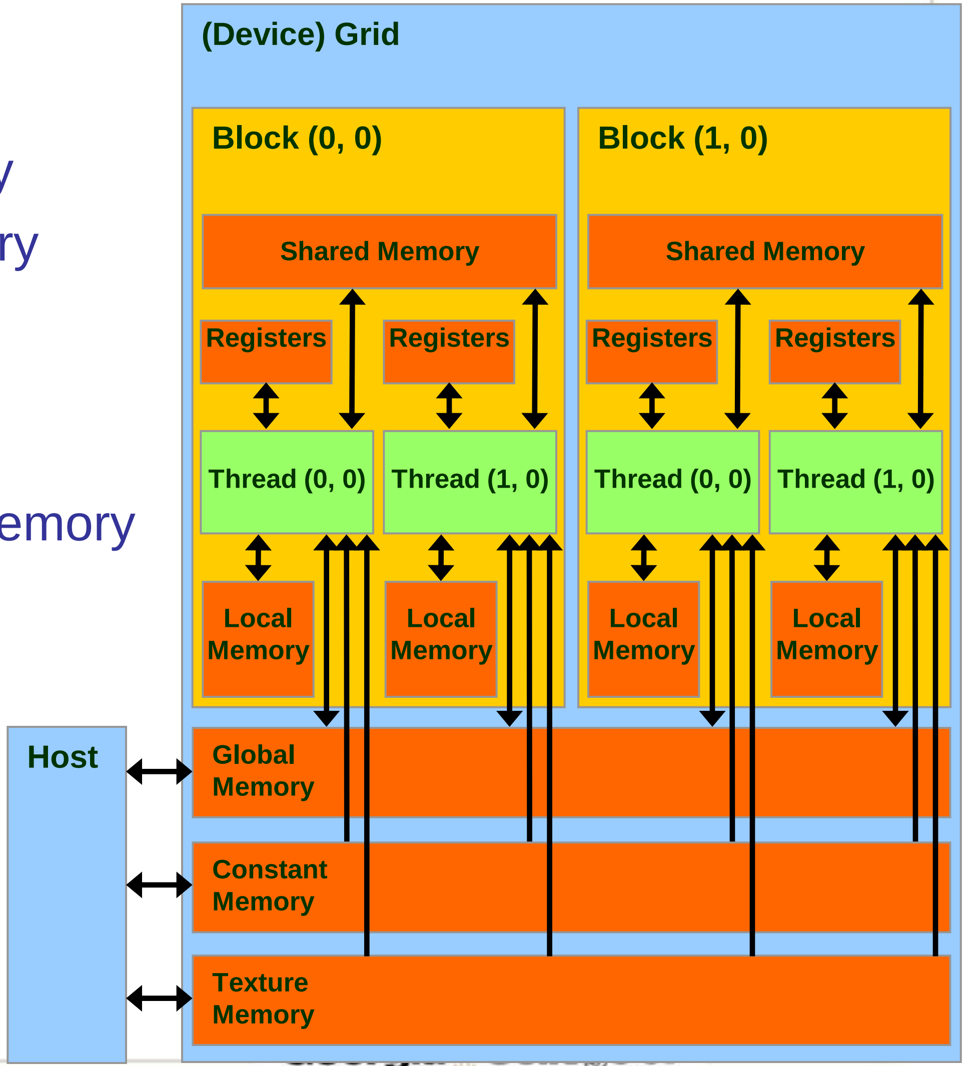
Prof. Hyesoon Kim

Georgia Tech College of Computing

CUDA Device Memory Space Review

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memories

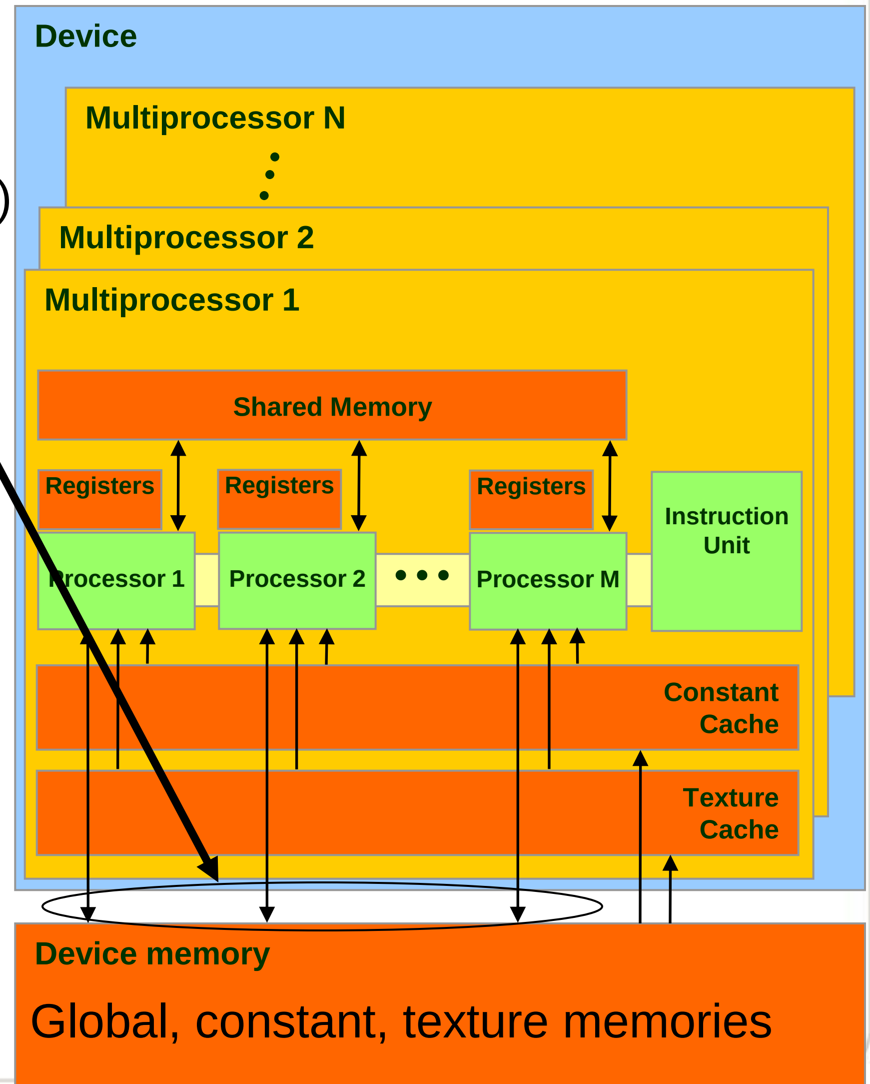


Access Times

- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - single cycle
- Local Memory – DRAM, no cache - *slow*
- Global Memory – DRAM, no cache - *slow*
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached

How about performance?

- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 4B/s of memory bandwidth/FLOPS
  - 86.4 GB/s limits the code at 21.6 GFLOPS
- The actual code should run at about 15 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak 346.5 GFLOPS



Idea: Use Shared Memory to reuse global memory data

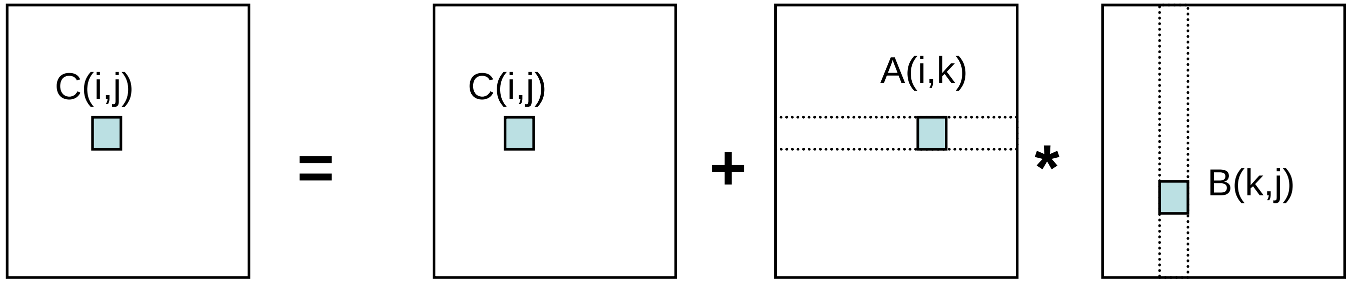
- Each input element is read by WIDTH threads.
- If we load each element into Shared Memory and have several threads use the local version, we can drastically reduce the memory bandwidth
  - Load all the matrix ?
  - Tiled algorithms

- Pattern
  - Copy data from global to shared memory
  - Synchronization
  - Computation (iteration)
  - Synchronization
  - Copy data from shared to global memory

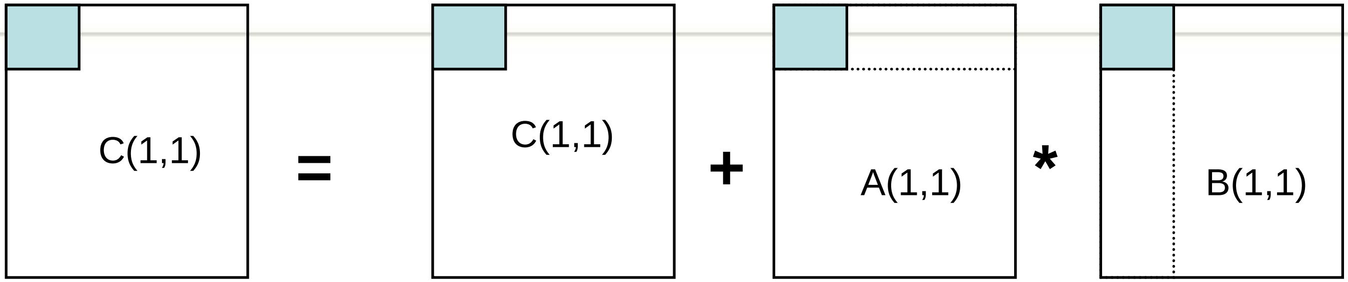
Blocked (Tiled) Matrix Multiply

Consider A,B,C to be N by N matrices of b by b subblocks where b=n / N is called the block size

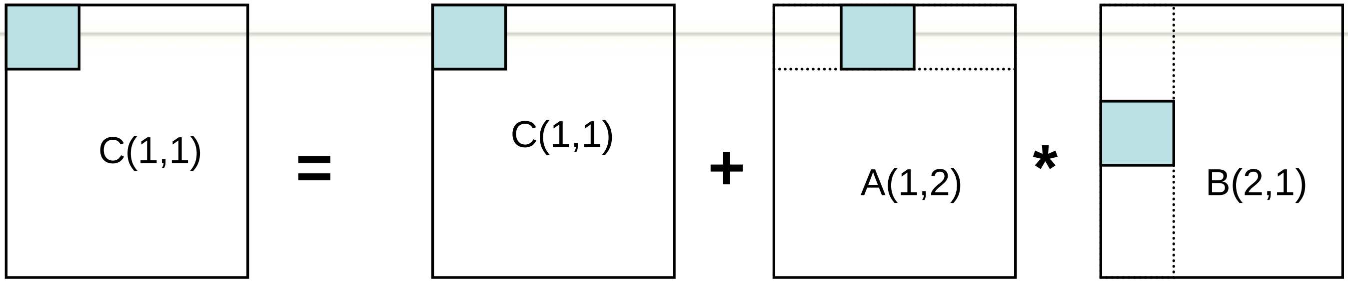
```
for i = 1 to N
   for j = 1 to N
      {read block C(i,j) into shared memory}
      for k = 1 to N
            {read block A(i,k) into shared memory}
            {read block B(k,j) into shared memory}
             C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
      {write block C(i,j) back to global memory}
```



Blocked (Tiled) Matrix Multiply

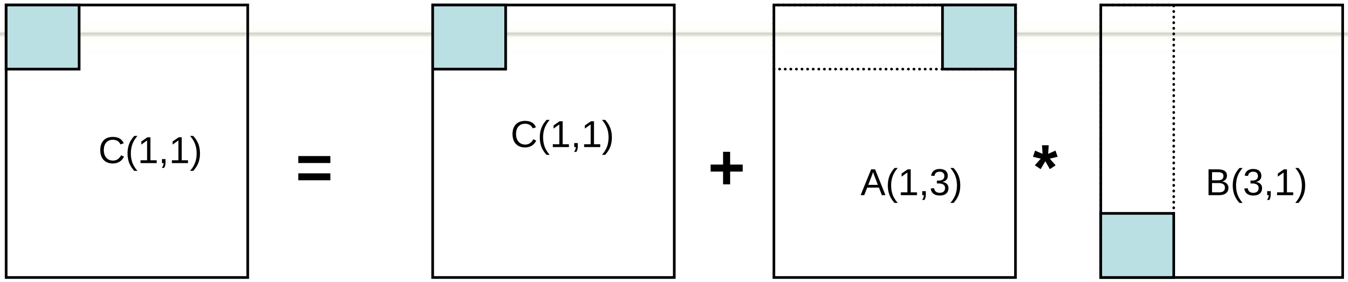


Blocked (Tiled) Matrix Multiply



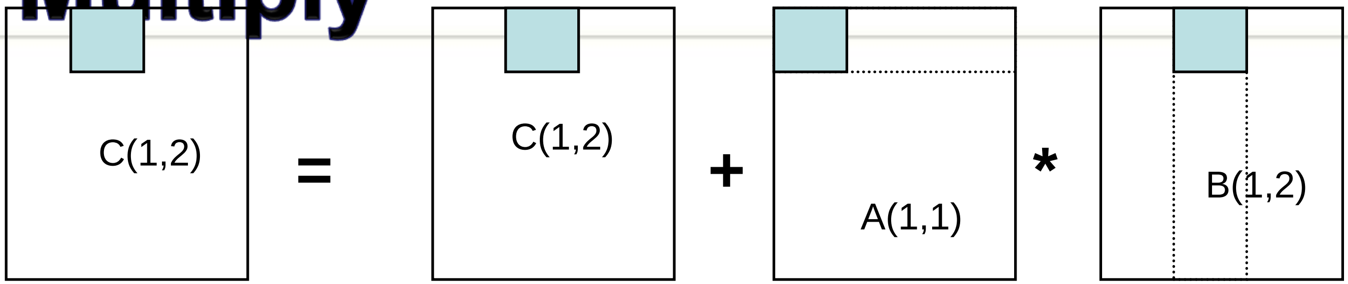
www.sdsc.edu/~allans/cs260/lectures/matmul.ppt

Blocked (Tiled) Matrix Multiply

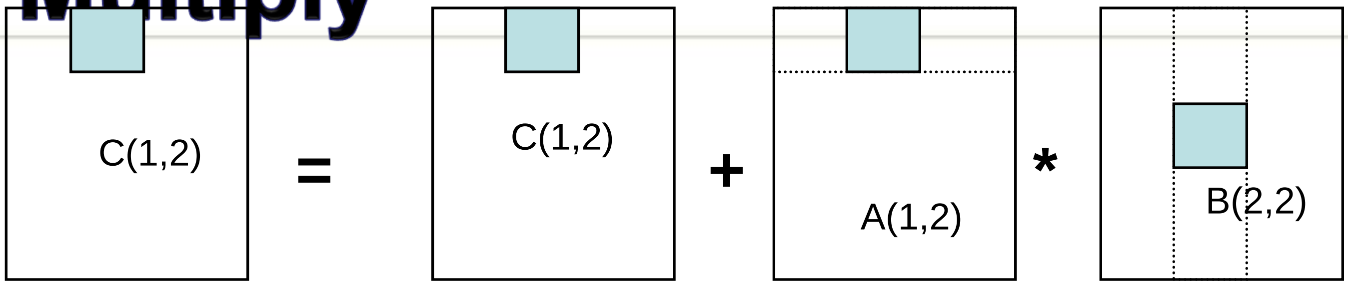


www.sdsc.edu/~allans/cs260/lectures/matmul.ppt

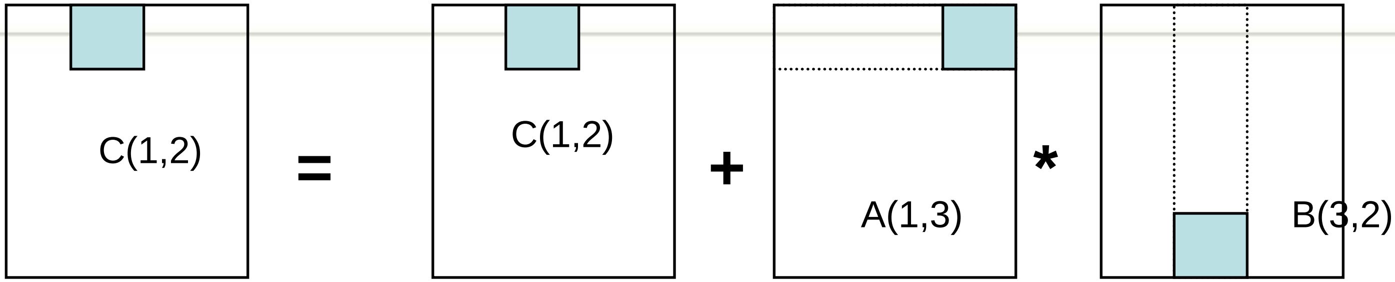
Blocked (Tiled) Matrix Multiply



Blocked (Tiled) Matrix Multiply



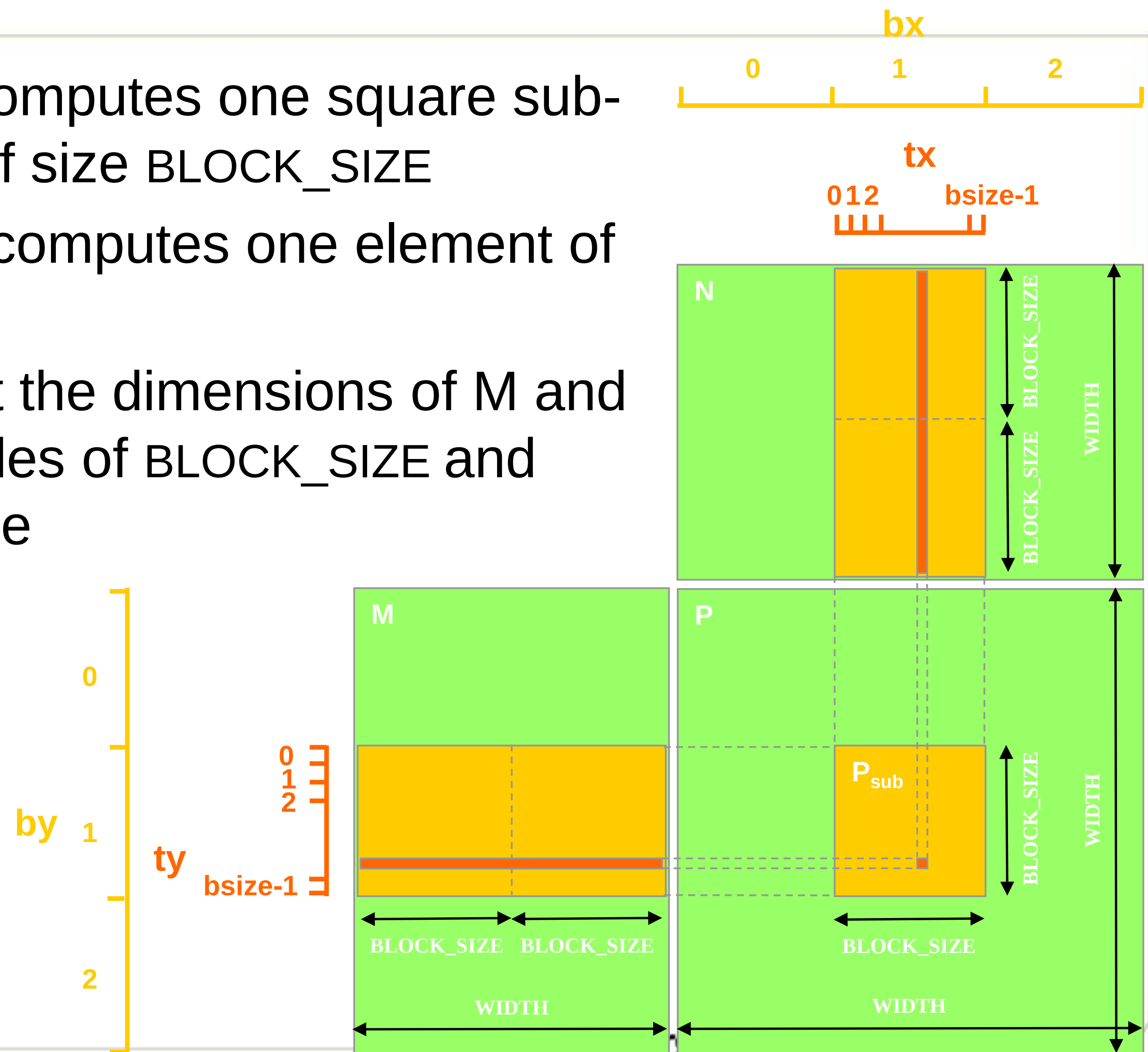
Blocked (Tiled) Matrix Multiply



Georgia Tech College of Computing

Tiled Multiply Using Thread Blocks

- One block computes one square sub-matrix P_{sub} of size BLOCK_SIZE
- One thread computes one element of P_{sub}
- Assume that the dimensions of M and N are multiples of BLOCK_SIZE and square shape



Shared Memory Usage

- Each SMP has 16KB shared memory
  - Each Thread Block uses 2 *256*4B = 2KB of shared memory. [2: two matrix, 256 = 16*16, 4B (floating point)]
  - Can potentially have up to 8 Thread Blocks actively executing
  - Initial load:
    - For BLOCK_SIZE = 16, this allows up to 8*512 = 4,096 pending loads (8 blocks, 2 loads * 256)
    - In practice, there will probably be up to half of this due to scheduling to make use of SPs.
  - The next BLOCK_SIZE 32 would lead to 2*32*32*4B= 8KB shared memory usage per Thread Block, allowing only up to two Thread Blocks active at the same time

CUDA Code – Kernel Execution Configuration

```
// Setup the execution configuration
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(N.width  / dimBlock.x,
             M.height / dimBlock.y);
```

For very large N and M dimensions, one will need to add another level of blocking and execute the second-level blocks sequentially.

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
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 < M.width/BLOCK_SIZE; ++m) {
      code from the next few slides };
```

CUDA Code - Load Data to Shared Memory

```
// Get a pointer to the current sub-matrix Msub of M
Matrix Msub = GetSubMatrix(M, m, by);

// Get a pointer to the current sub-matrix Nsub of N
Matrix Nsub = GetSubMatrix(N, bx, m);

__shared__ float Ms[BLOCK_SIZE][BLOCK_SIZE];
__shared__ float Ns[BLOCK_SIZE][BLOCK_SIZE];

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

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

GetSubMatrix(M, m, by)

```
//Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
//located col sub-matrices to the right and row sub-matrices down
//from the upper-left corner of A
__device__ Matrix GetSubMatrix(Matrix A, const int row, const int col)
{
  Matrix Asub;
  Asub.width    = BLOCK_SIZE;
  Asub.height   = BLOCK_SIZE;
  Asub.stride   = A.stride;
  Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row + BLOCK_SIZE * col];
  return Asub;
}
```

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 < BLOCK_SIZE; ++k)
    Pvalue += Ms[ty][k] * Ns[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 P
Matrix Psub = GetSubMatrix(P, bx, by);


// Write the block sub-matrix to device memory;
// each thread writes one element
SetMatrixElement(Psub, tx, ty, Pvalue);
```

Macro functions will be provided.

Device Runtime Component: Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a block
- Once all threads have reached this point, execution resumes normally
- Used to avoid RAW/WAR/WAW hazards when accessing shared or global memory
- Allowed in conditional constructs only if the conditional is uniform across the entire thread block

```
if (tid>16) {__syncthreads(); code1 …}
else { code1; }
```

DON'T DO IT

- Some Useful Information on Tools

Compilation

- Any source file containing CUDA language extensions must be compiled with nvcc
- nvcc is a compiler driver
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...
- nvcc can output:
  - Either C code
    - That must then be compiled with the rest of the application using another tool
  - Or object code directly

Debugging Using the Device Emulation Mode

- An executable compiled in device emulation mode (`nvcc -deviceemu`) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver (??)
  - Each device thread is emulated with a host thread
- When running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. `printf`) and vice-versa
  - Detect deadlock situations caused by improper usage of `__synchthreads`

Device Emulation Mode Pitfalls

- Emulated device threads execute sequentially, so simultaneous accesses of the same memory location by multiple threads could produce different results.
- Dereferencing device pointers on the host or host pointers on the device can produce correct results in device emulation mode, but will generate an error in device execution mode
- Results of floating-point computations will slightly differ because of:
  - Different compiler outputs, instruction sets
  - Use of extended precision for intermediate results
    - There are various options to force strict single precision on the host