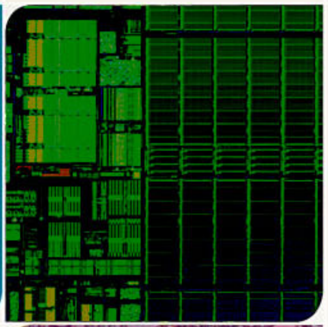
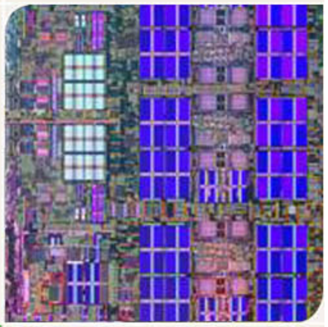
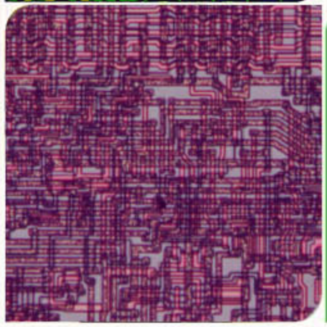
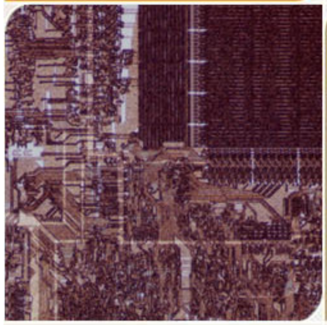





CS4803DGC Design Game Consoles

Spring 2009

Prof. Hyesoon Kim



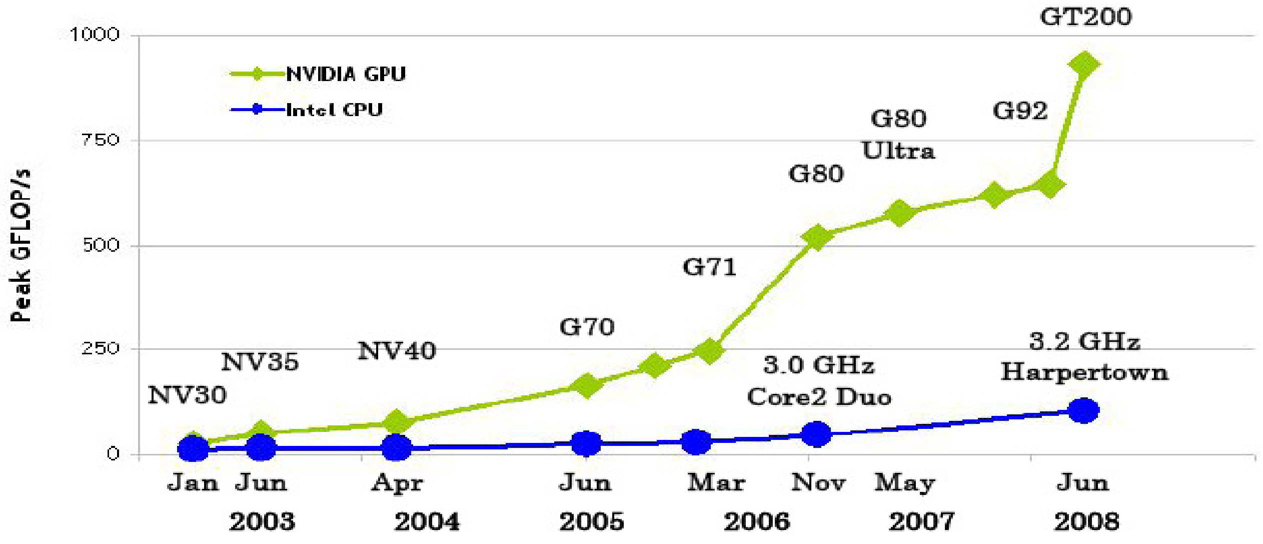
Georgia Tech College of Computing

CUDA

- “Compute Unified Device Architecture”
- Available for GeForce 8, 9 Series, Quadro FX5600/4600, and Tesla solutions
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute - graphics free API
- Cuda provides general DRAM memory addressing (just like CPU)

Why Programming with GPU?

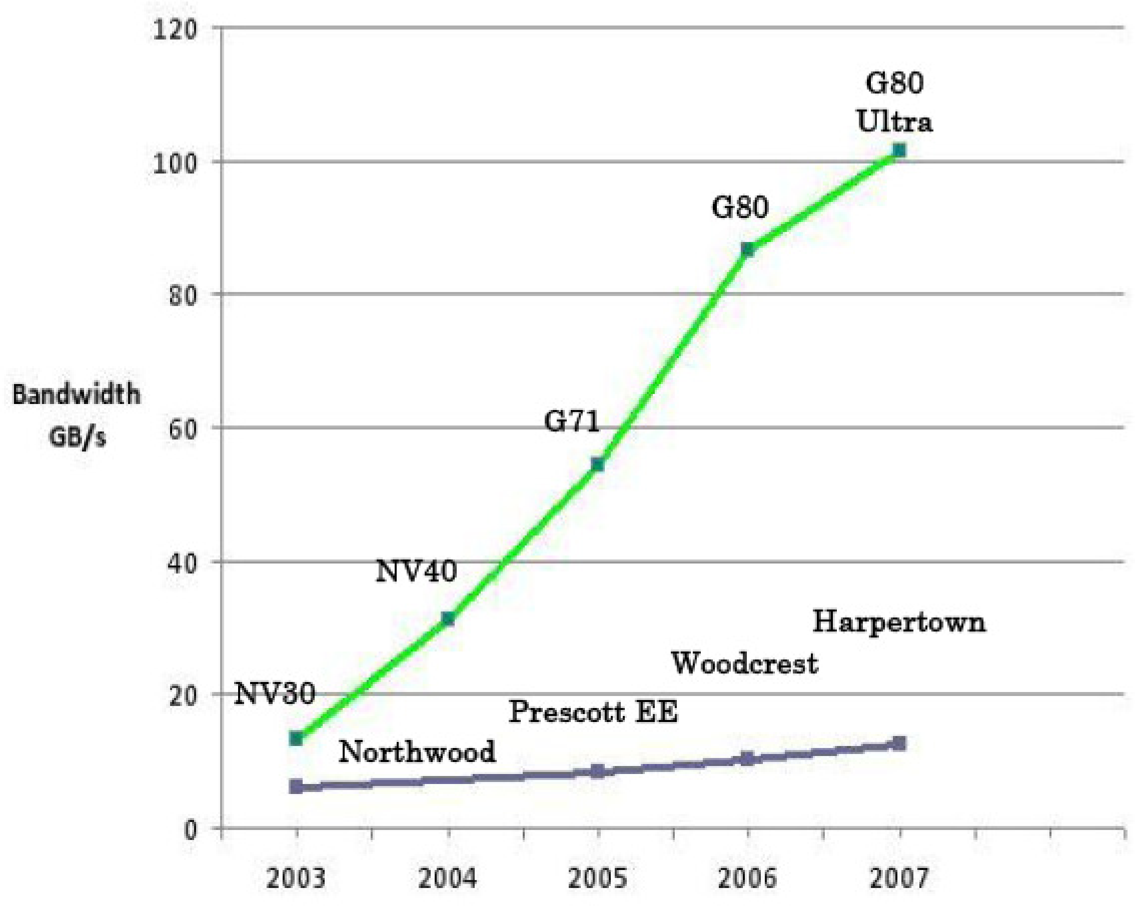
- A quiet revolution and potential build-up
  - Calculation: 367 GFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
- Until a few years, programmed through graphics API



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

Georgia Tech College of Computing

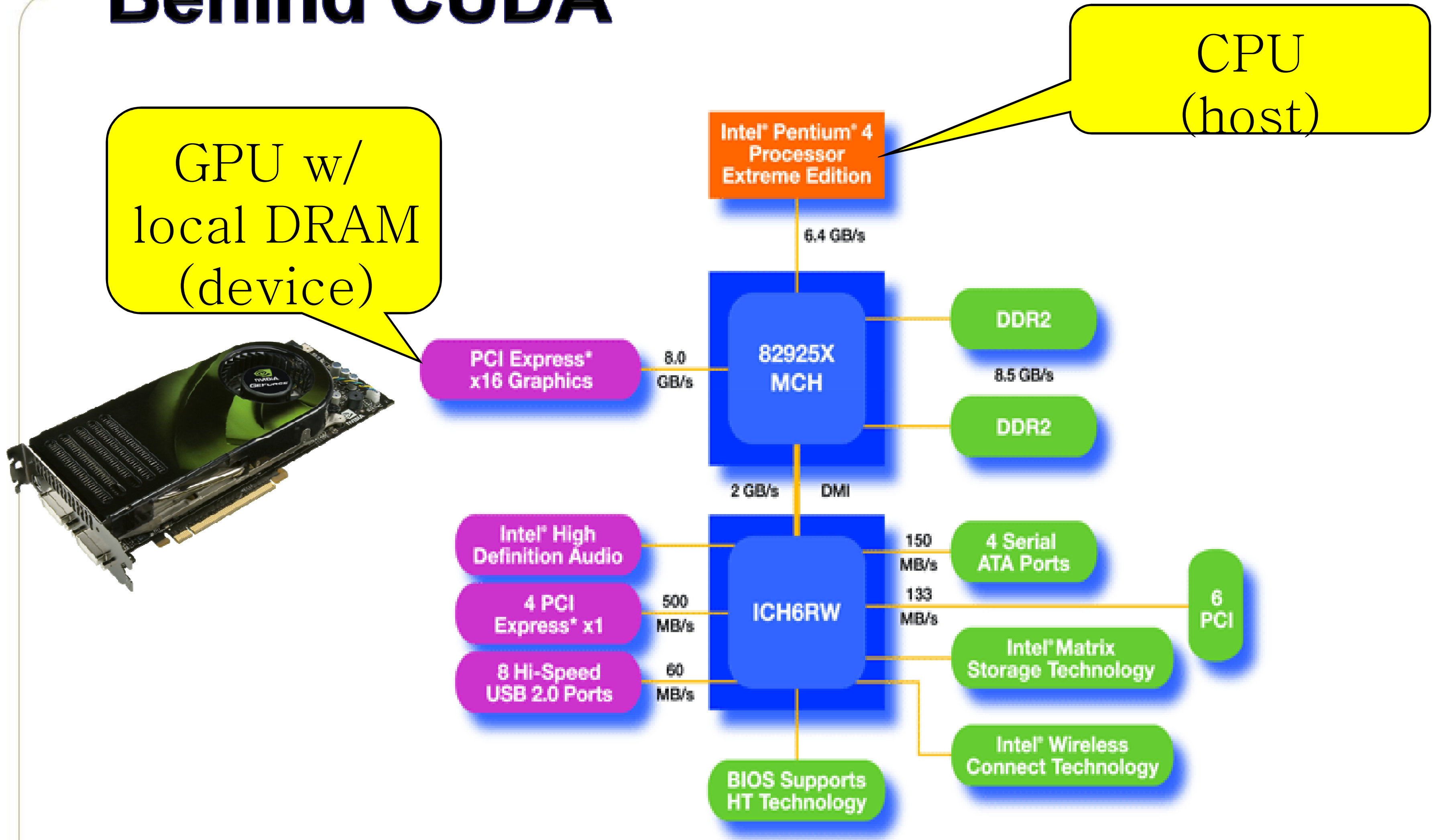
Memory Bandwidth for the CPU and GPU



CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

An Example of Physical Reality Behind CUDA

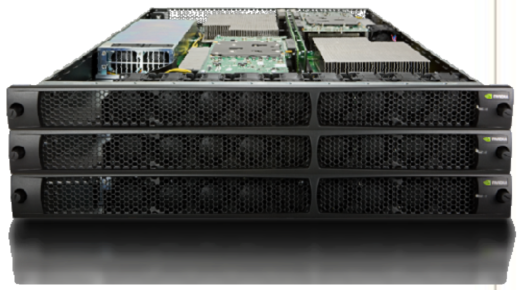

CPU (host)
GPU w/ local DRAM (device)
Intel® Pentium® 4 Processor Extreme Edition
6.4 GB/s
82925X MCH
PCI Express* x16 Graphics
8.0 GB/s
DDR2
8.5 GB/s
DDR2
2 GB/s
DMI
ICH6RW
Intel® High Definition Audio
4 PCI Express* x1
500 MB/s
8 Hi-Speed USB 2.0 Ports
60 MB/s
150 MB/s
4 Serial ATA Ports
133 MB/s
6 PCI
Intel® Matrix Storage Technology
Intel® Wireless Connect Technology
BIOS Supports HT Technology

Parallel Computing on a GPU

- NVIDIA GPU Computing Architecture
  - Via a separate HW interface
  - In laptops, desktops, workstations, servers
- 8-series GPUs deliver 50 to 200 GFLOPS on compiled parallel C applications
- GPU parallelism is doubling every year
- Programming model scales transparently
- Programmable in C with CUDA tools
- Multithreaded SPMD model uses application data parallelism and thread parallelism

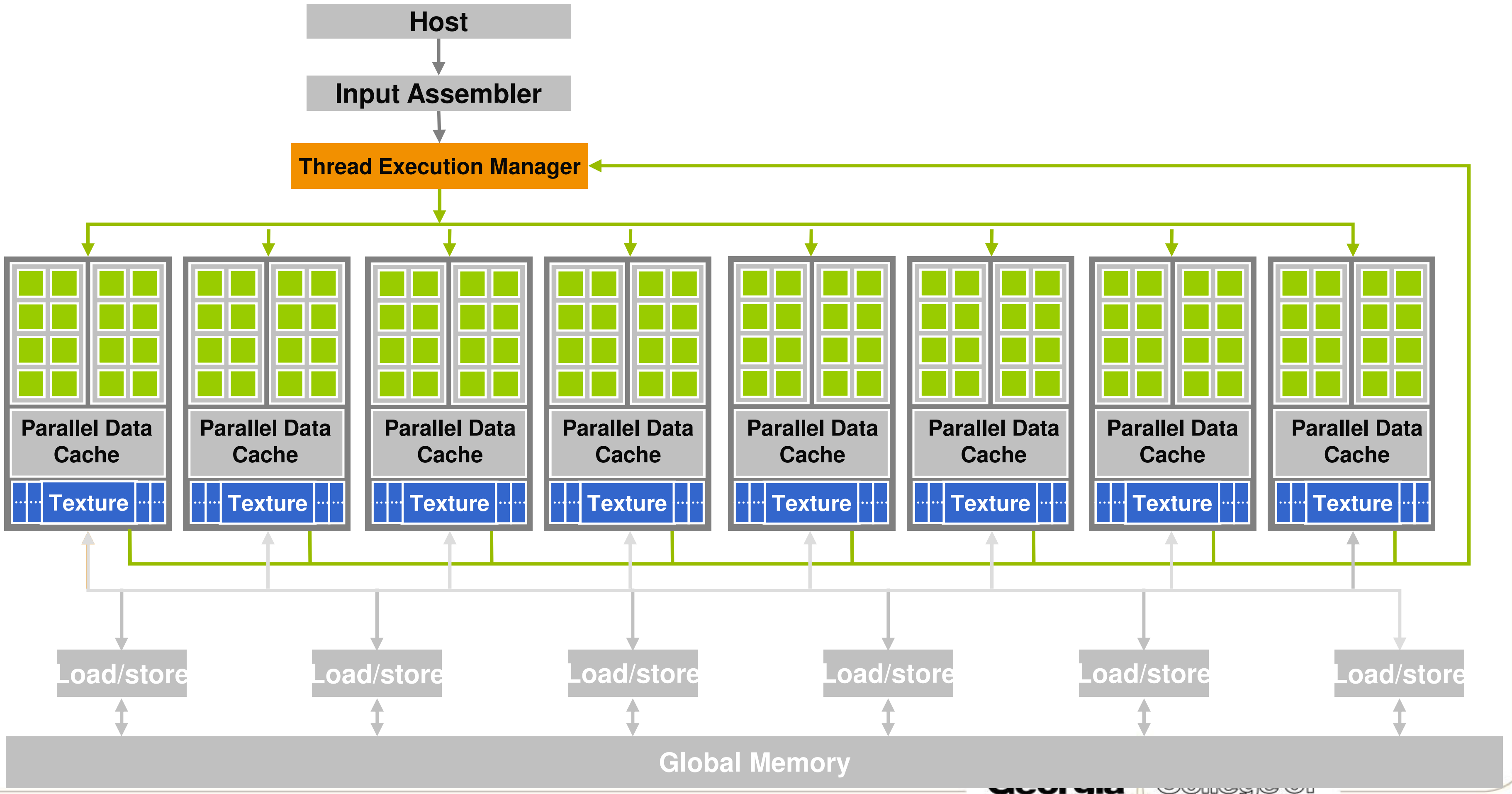
GeForce 8800

Tesla D870


Tesla S870

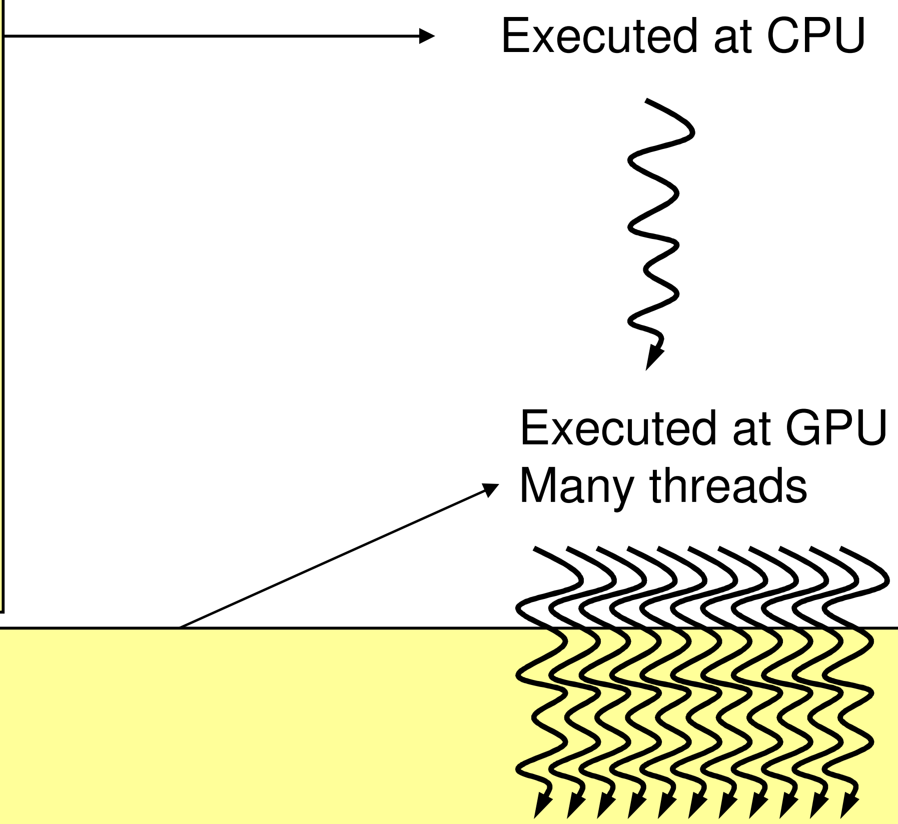
GeForce 8800

16 highly threaded SM's, >128 FPU's, 367 GFLOPS, 768 MB DRAM, 86.4 GB/S Mem BW, 4GB/S BW to CPU



Code Example (HelloWorld)

```
helloworld.cu
Int main()
{
  CUT_DEVICE_INIT();

   dim3 threads (1, 2, 4);
   dim3 grid (2,1);

 helloworld<<< grid, threads >>> ();
 return;
}
```



```
helloworld_kernel.cu
__global__ void
helloworld()
{
  printf("hello world! I'm a thread with block Id:{%d %d}, Thread Id{%d %d %d}\n",
 blockIdx.x, blockIdx.y, threadIdx.x, threadIdx.y, threadIdx.z);
}
```

Output of Helloworld

```
dim3 threads (1, 2, 4);
dim3 grid (2,1);
helloworld<<< grid, threads >>> ();
```

```
Hello World! I am a thread with BlockId: {0,0}, ThreadId:{0,0,0}
Hello World! I am a thread with BlockId: {0,0}, ThreadId:{0,1,0}
Hello World! I am a thread with BlockId: {0,0}, ThreadId:{0,0,1}
Hello World! I am a thread with BlockId: {0,0}, ThreadId:{0,1,1}
Hello World! I am a thread with BlockId: {0,0}, ThreadId:{0,0,2}
Hello World! I am a thread with BlockId: {0,0}, ThreadId:{0,1,2}
Hello World! I am a thread with BlockId: {0,0}, ThreadId:{0,0,3}
Hello World! I am a thread with BlockId: {0,0}, ThreadId:{0,1,3}
Hello World! I am a thread with BlockId: {1,0}, ThreadId:{0,0,0}
Hello World! I am a thread with BlockId: {1,0}, ThreadId:{0,1,0}
Hello World! I am a thread with BlockId: {1,0}, ThreadId:{0,0,1}
Hello World! I am a thread with BlockId: {1,0}, ThreadId:{0,1,1}
Hello World! I am a thread with BlockId: {1,0}, ThreadId:{0,0,2}
Hello World! I am a thread with BlockId: {1,0}, ThreadId:{0,1,2}
Hello World! I am a thread with BlockId: {1,0}, ThreadId:{0,0,3}
Hello World! I am a thread with BlockId: {1,0}, ThreadId:{0,1,3}
```

Extended C

- **Declspecs**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function launch**

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

G80 Characteristics

- 367 GFLOPS peak performance (25-50 times of current high-end microprocessors)
- Massively parallel, 128 cores, 90W
- Massively threaded, sustains 1000s of threads per app
- 30-100 times speedup over high-end microprocessors on scientific and media applications: medical imaging, molecular dynamics

CUDA

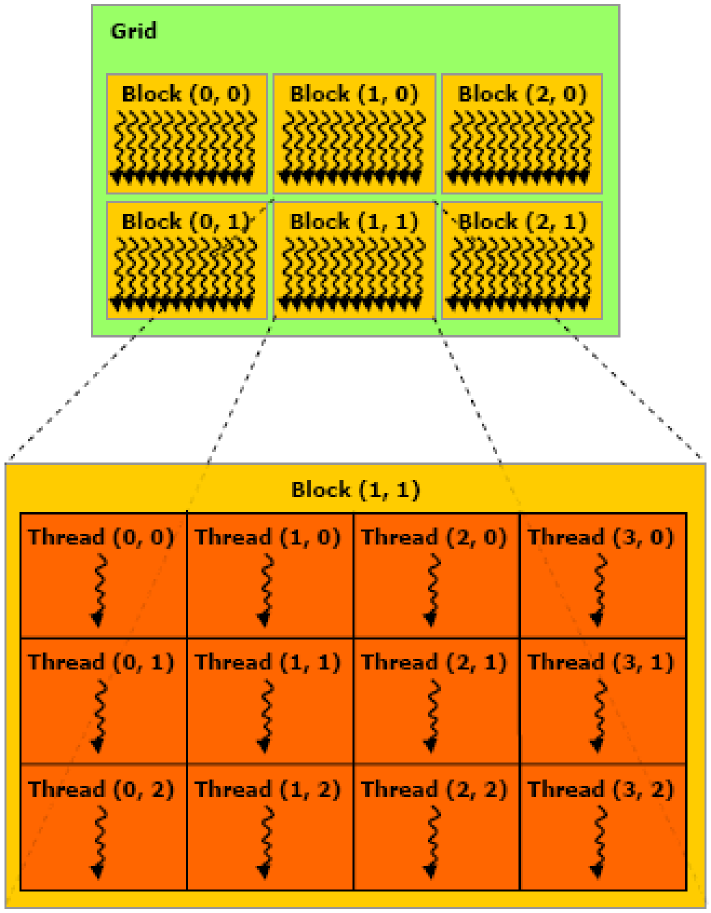
- CUDA is a programming system for utilizing the G80 processor for compute
  - CUDA follows the architecture very closely
- General purposed programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel processor

Matches architecture features

Specific parameters are not exposed

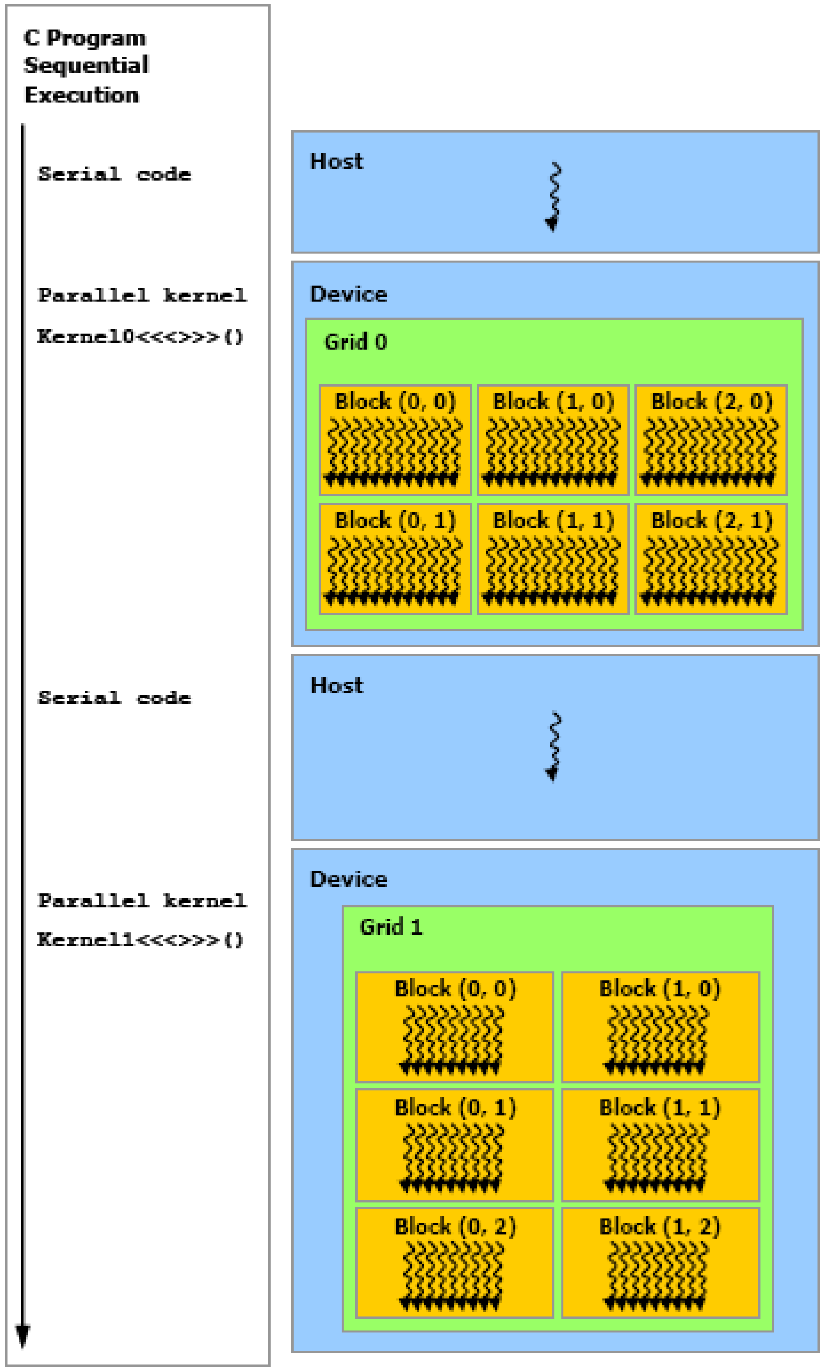
Programming model: Block and Thread IDs

- A kernel is executed as a grid of thread blocks
- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …

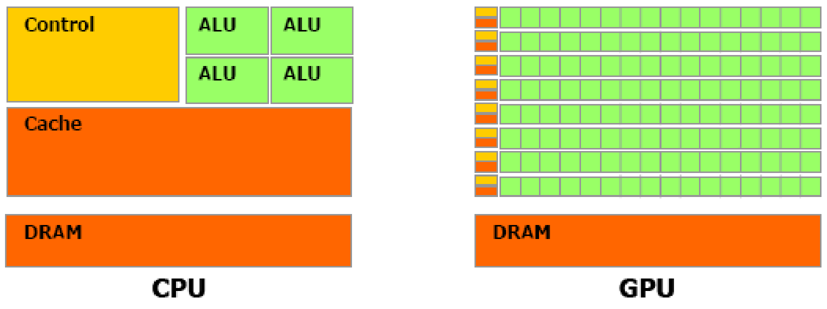


Courtesy: NDVIA

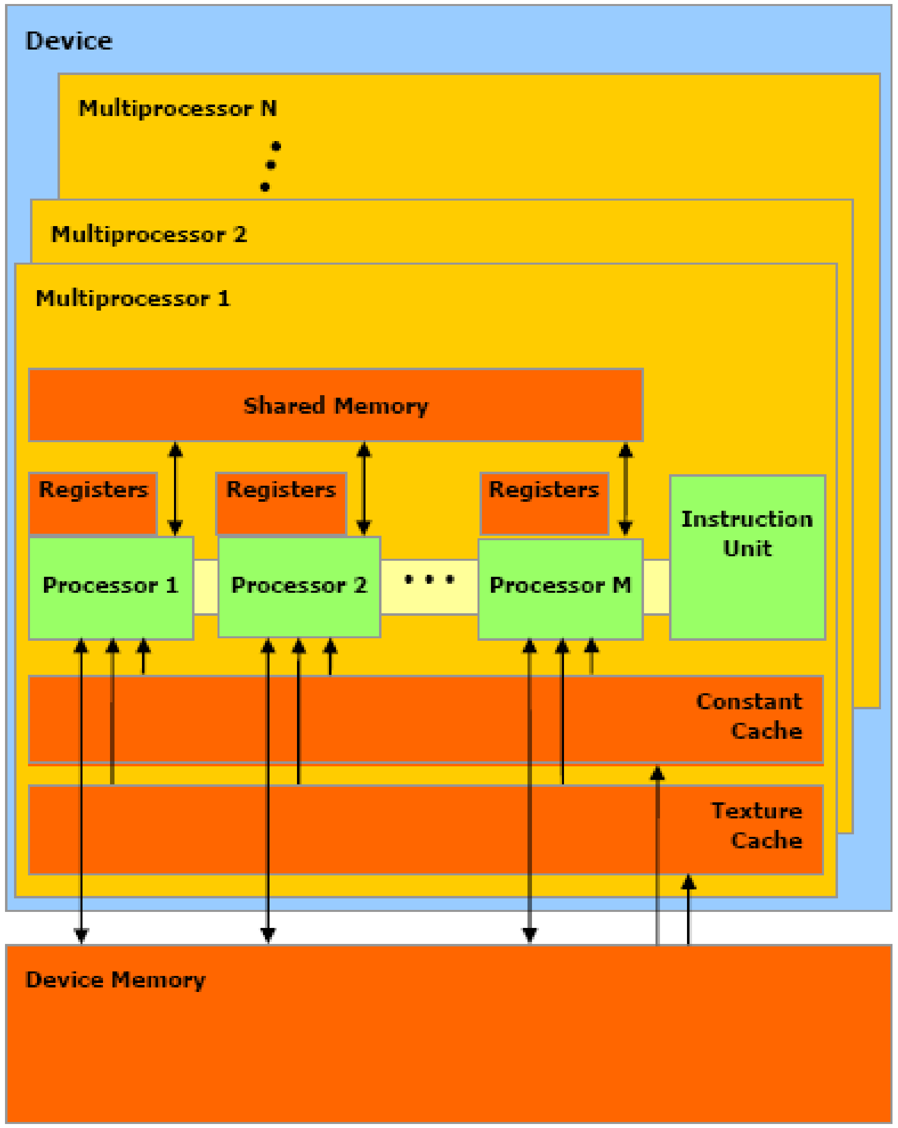
Georgia Tech College of Computing



Serial code executes on the host while parallel code executes on the device.



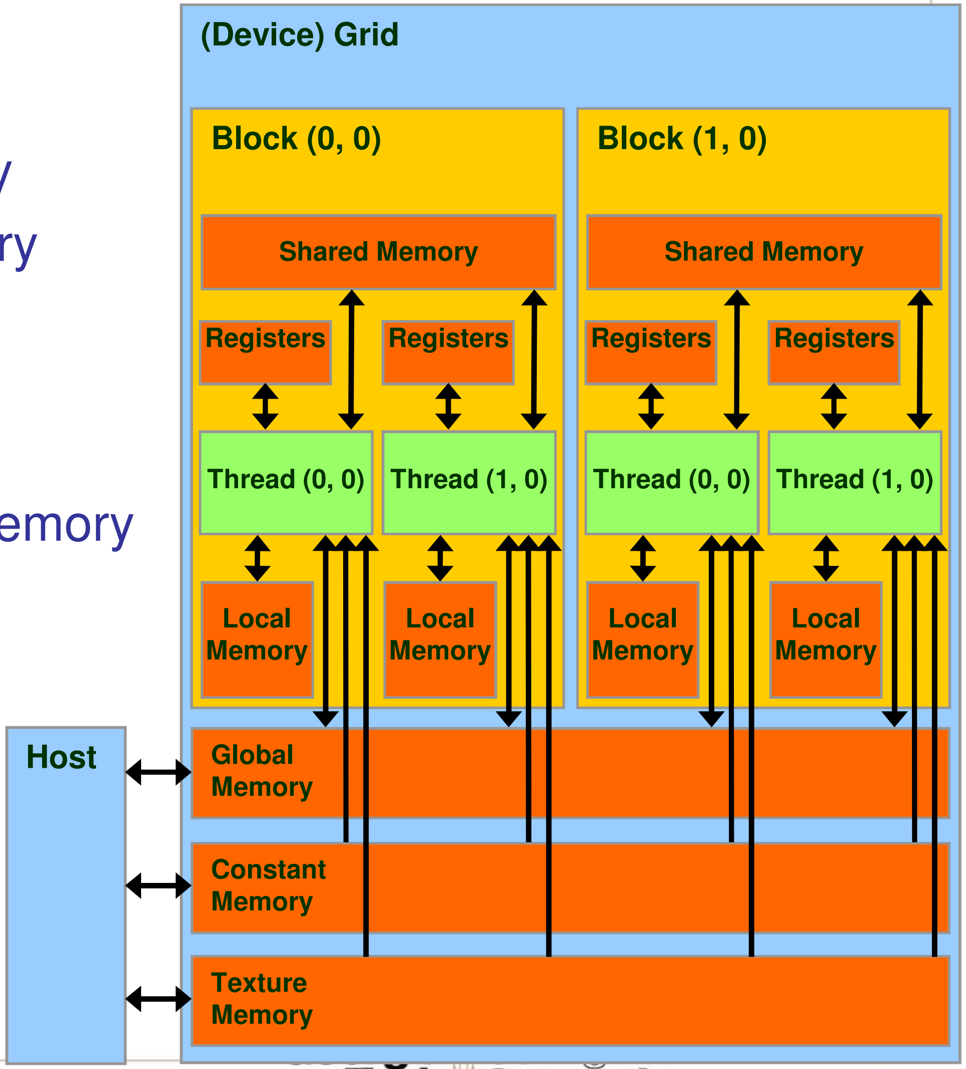
Hardware Model



CUDA Device Memory Space Overview

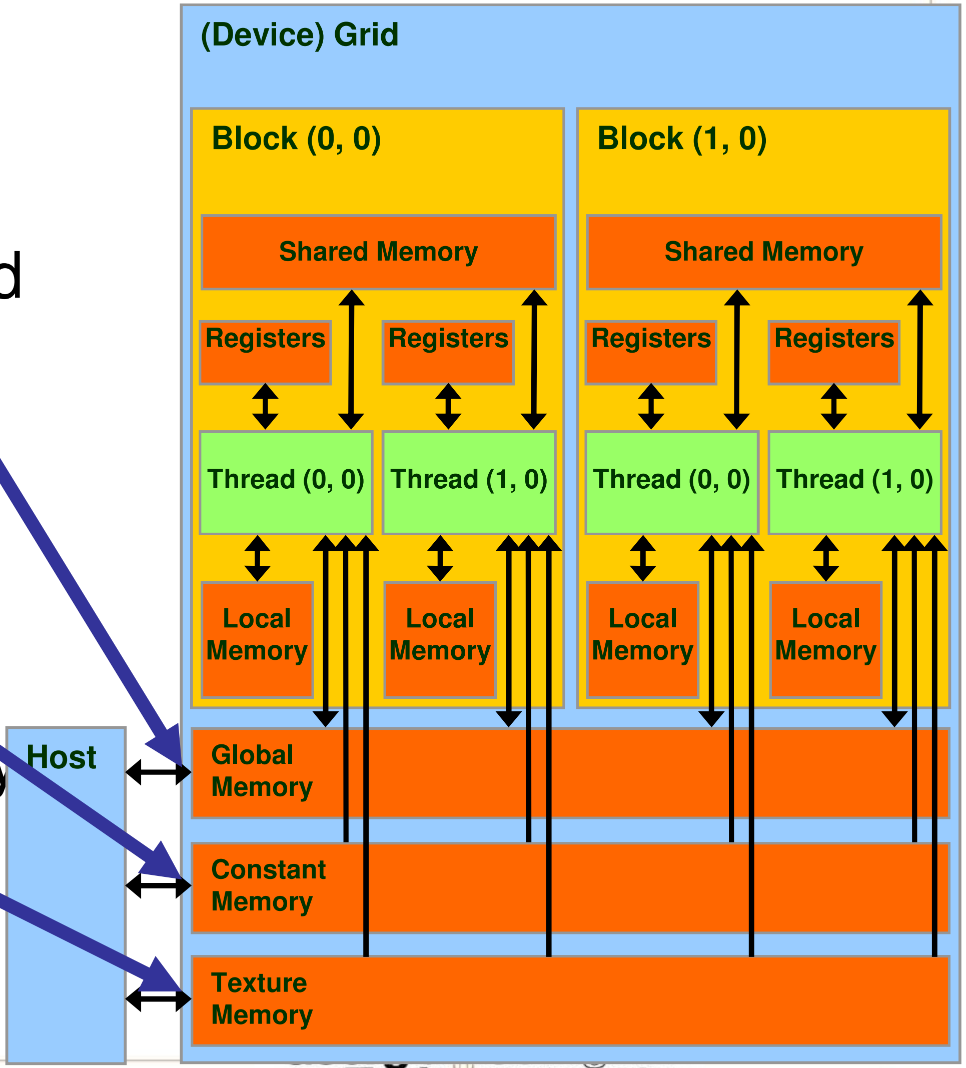
- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

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



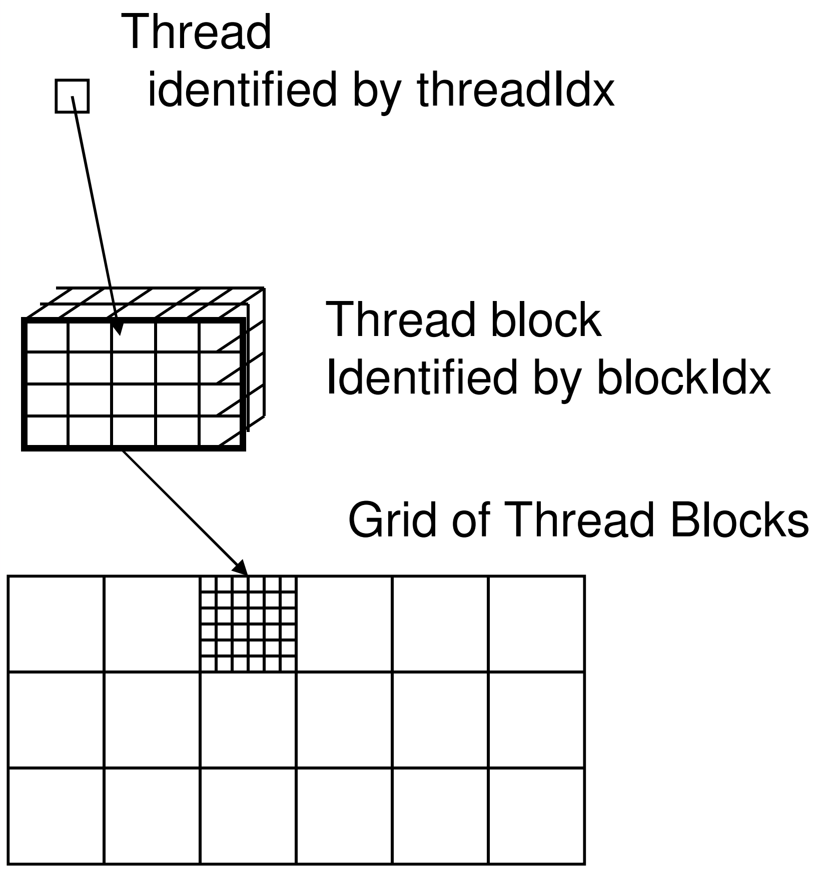
Global, Constant, and Texture Memories (Long Latency Accesses)

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads



Courtesy: NDVIA

Execution Model



Multiple levels of parallelism

-Thread block

- -Up to 512 threads per block
- -Communicate through shared memory
- -Threads guaranteed to be resident
- -threadIdx, blockIdx
- -__syncthreads()

-Grid of thread blocks

- -F <<< nblocks, nthreads >>> (a, b, c)

- CUDA – API

CUDA Highlights: Easy and Lightweight

- The API is an extension to the ANSI C programming language

  → Low learning curve

- The hardware is designed to enable lightweight runtime and driver

  → High performance

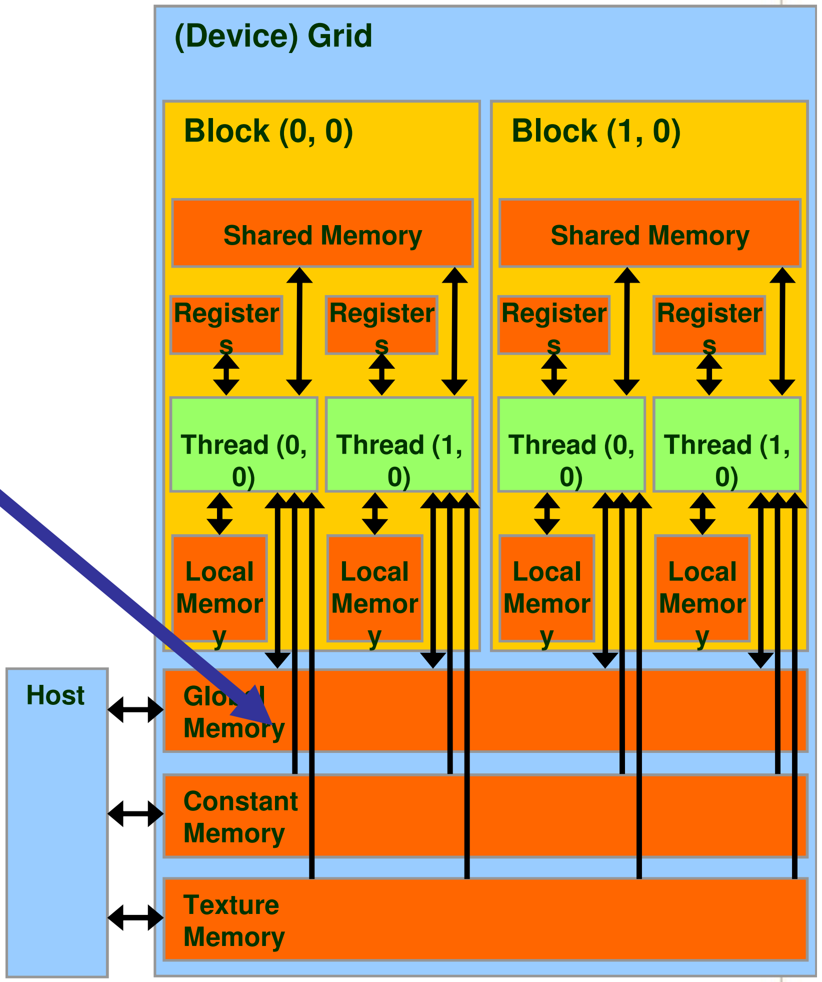
A Small Detour: A Matrix Data Type

- NOT part of CUDA
- It will be frequently used in many code examples
  - 2 D matrix
  - single precision float elements
  - width * height elements
  - pitch is meaningful when the matrix is actually a sub-matrix of another matrix
  - data elements allocated and attached to elements

```
typedef struct {
    int width;
    int height;
    int pitch;
    float* elements;
} Matrix;
```

CUDA Device Memory Allocation

- cudaMalloc()
  - Allocates object in the device <u>Global Memory</u>
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object
- cudaFree()
  - Frees object from device Global Memory
    - Pointer to freed object



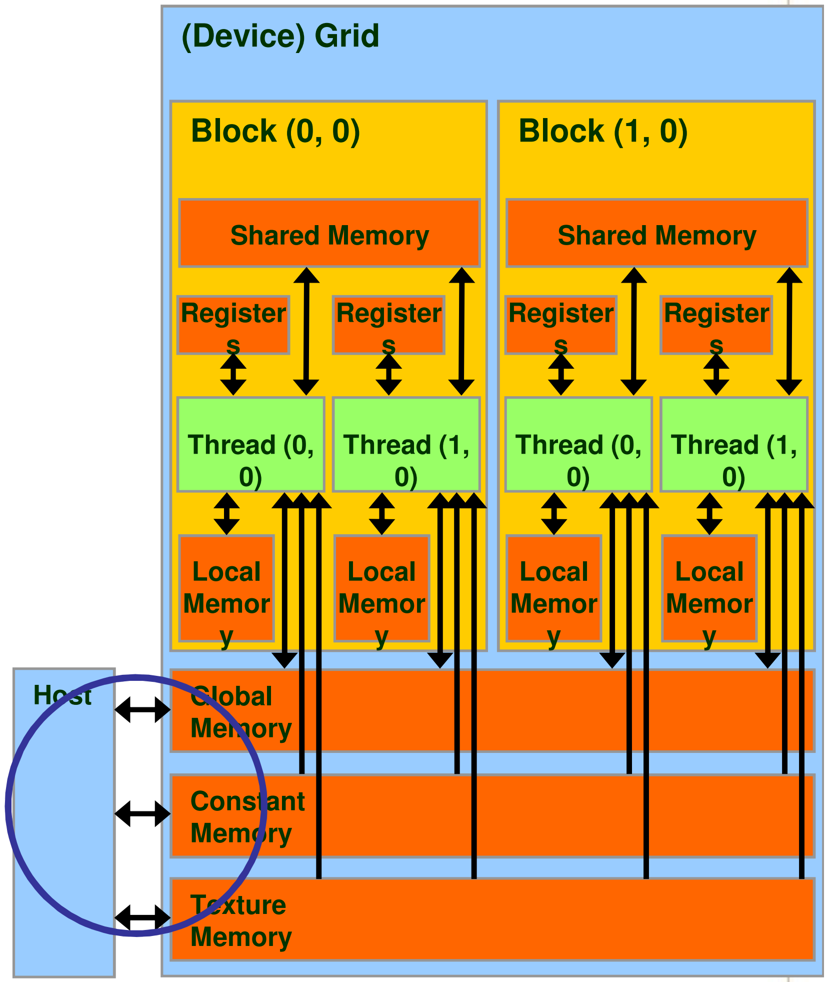
CUDA Device Memory Allocation (cont.)

- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to Md.elements
  - “d” is often used to indicate a device data structure

```
BLOCK_SIZE = 64;
Matrix Md
int size = BLOCK_SIZE * BLOCK_SIZE * sizeof(float);

cudaMalloc((void**)&Md.elements, size);
cudaFree(Md.elements);
```

CUDA Host-Device Data Transfer

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Asynchronous in CUDA 1.0



CUDA Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

```
cudaMemcpy(Md.elements, M.elements, size,
    cudaMemcpyHostToDevice);

cudaMemcpy(M.elements, Md.elements, size,
    cudaMemcpyDeviceToHost);
```

CUDA Function Declarations

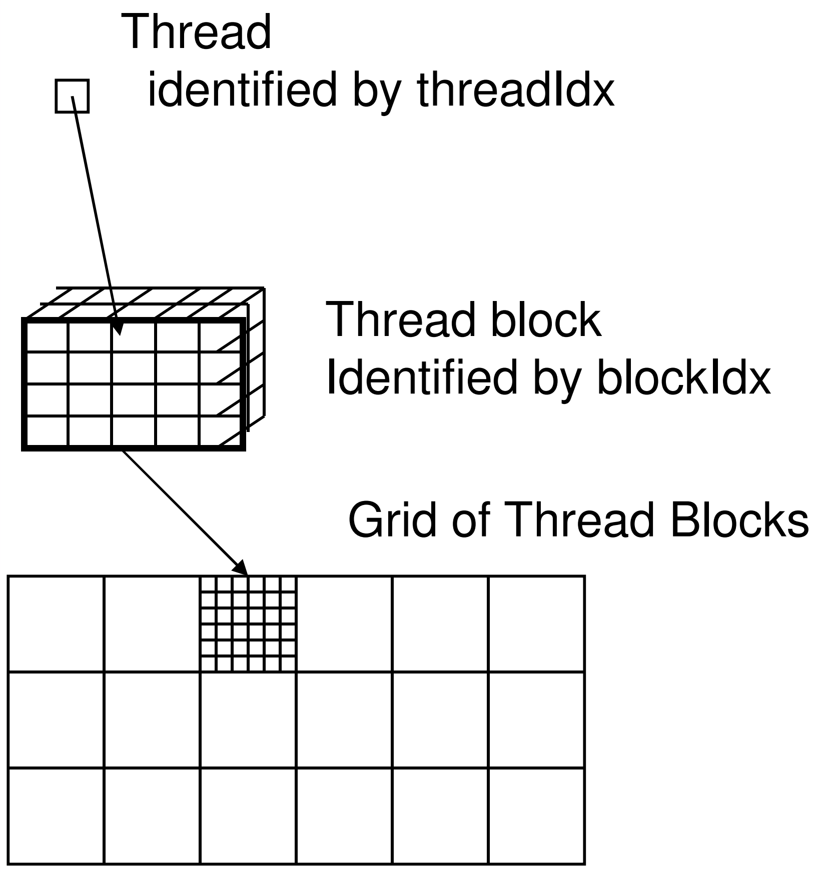
	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void  KernelFunc()`	device	host
`__host__   float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Must return `void`

CUDA Function Declarations (cont.)

- `__device__` functions cannot have their address taken
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

Review: Execution Model



Multiple levels of parallelism

-Thread block

- -Up to 512 threads per block
- -Communicate through shared memory
- -Threads guaranteed to be resident
- -threadIdx, blockIdx
- -__syncthreads()

-Grid of thread blocks

- -F <<< nblocks, nthreads >>> (a, b, c)

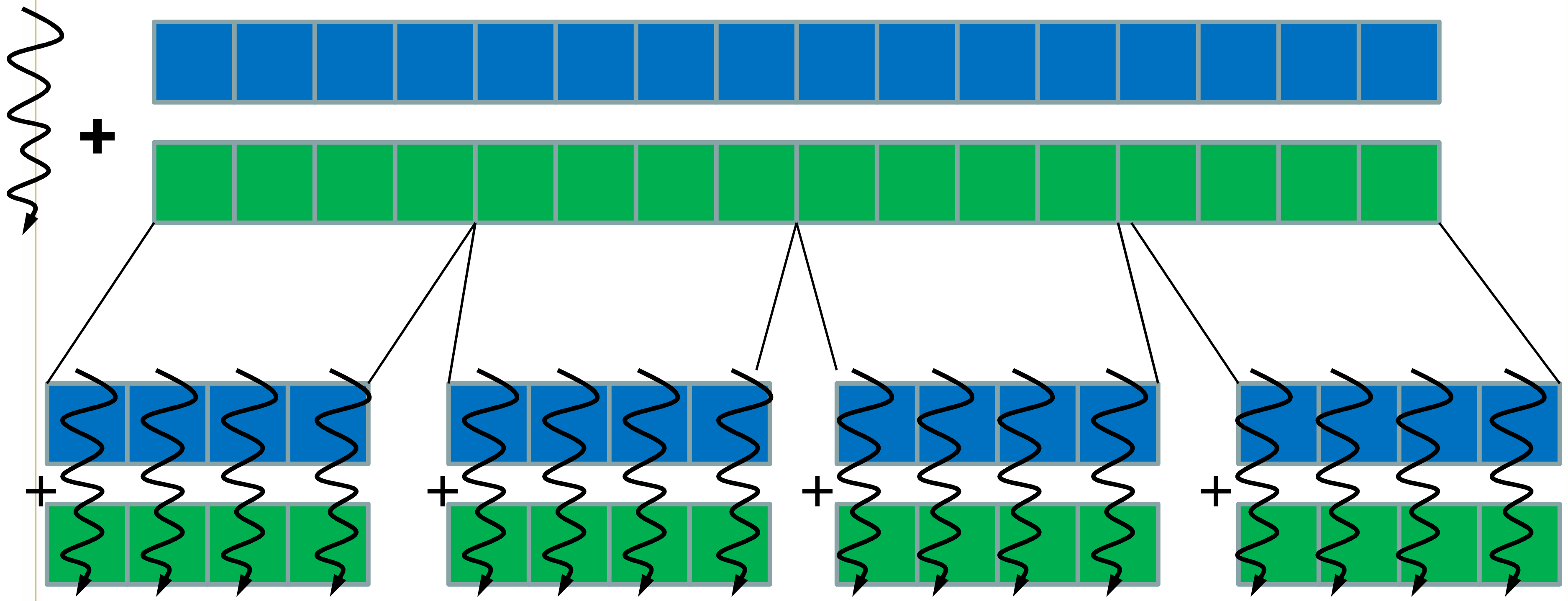
Review: Calling a Kernel Function – Thread Creation

- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3   DimGrid(100, 50);     // 5000 thread blocks
dim3   DimBlock(4, 8, 8);    // 256 threads per
   block
size_t SharedMemBytes = 64; // 64 bytes of shared
   memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes
   >>>(...);
```

Elementwise Matrix Addition

- Let’s assume N=16, blockDim=4 → 4 blocks



blockIdx.x = 0
blockDim.x = 4
threadIdx.x = 0,1,2,3
Idx= 0,1,2,3

blockIdx.x = 1
blockDim.x = 4
threadIdx.x = 0,1,2,3
Idx= 4,5,6,7

blockIdx.x = 2
blockDim.x = 4
threadIdx.x = 0,1,2,3
Idx= 8,9,10,11

blockIdx.x = 3
blockDim.x = 4
threadIdx.x = 0,1,2,3
Idx= 12,13,14,15

Elementwise Matrix Addition

CPU Program

```
void add matrix
 ( float *a, float* b, float *c, int N) {
    int index;
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j) {
         index = i + j*N;
         c[index] = a[index] + b[index];
        }
 }

int main () {

  add matrix (a, b, c, N);
}
```

GPU Program

```
__global__  add_matrix
  ( float *a, float *b, float *c, int N) {
int i = blockIdx.x *  blockDim.x + threadIdx.x;
Int j = blockIdx.y * blockDim.y  + threadIdx.y;
int index = i + j*N;
 if (i < N && j < N)
   c[index] = a[index]+b[index];
}

Int main() {
  dim3 dimBlock( blocksize, blocksize) ;
  dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N);
}
```

A Simple Running Example Matrix Multiplication

- A straightforward matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  - Leave shared memory usage until later
  - Local, register usage
  - Thread ID usage
  - Memory data transfer API between host and device

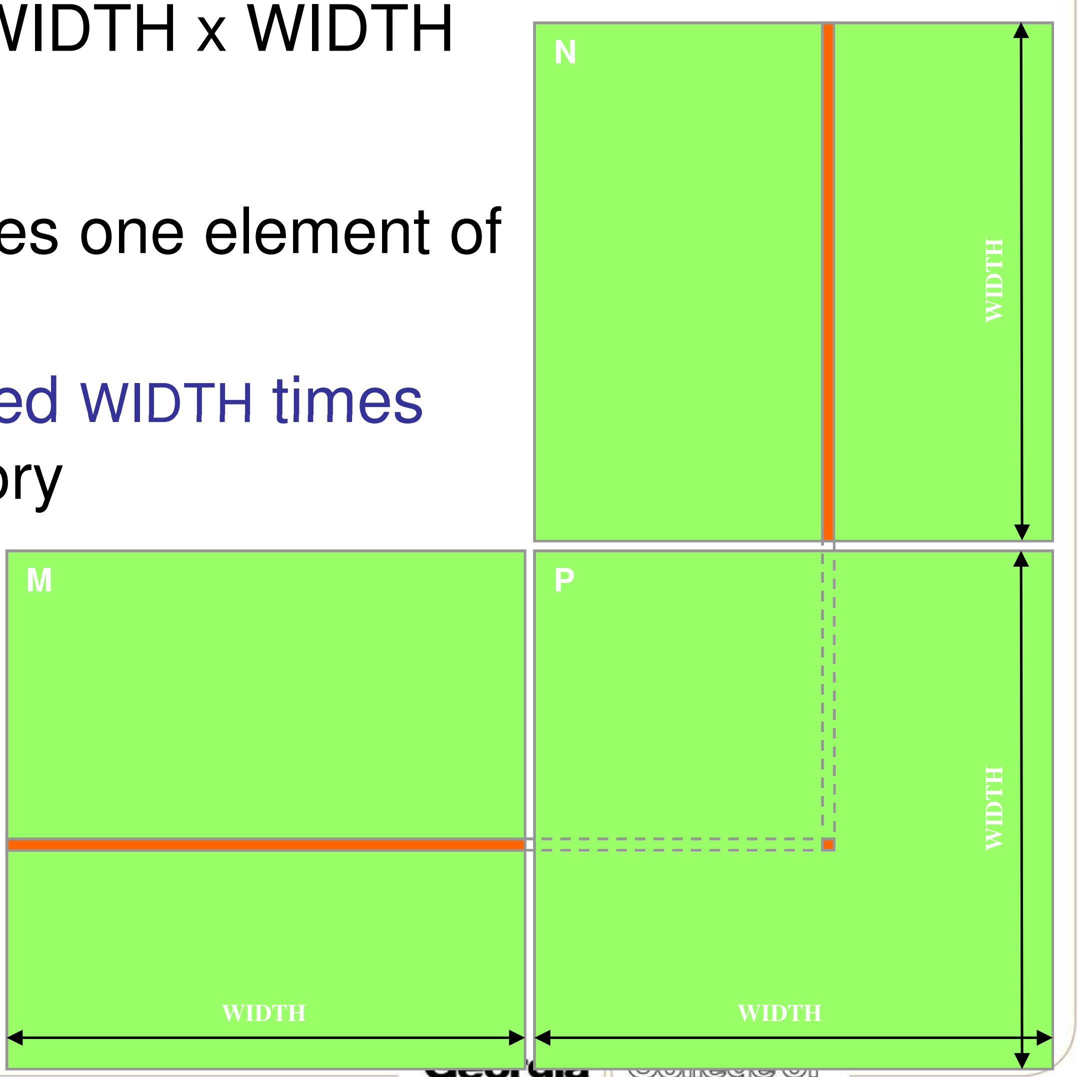
A Small Detour: A Matrix Data Type

- NOT part of CUDA
- It will be frequently used in many code examples
  - 2 D matrix
  - single precision float elements
  - width * height elements
  - pitch is meaningful when the matrix is actually a sub-matrix of another matrix
  - data elements allocated and attached to elements

```
typedef struct {
    int width;
    int height;
    int pitch;
    float* elements;
} Matrix;
```

Programming Model: Square Matrix Multiplication Example

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



Step 1: Matrix Data Transfers

```
// Allocate the device memory where we will copy M to
Matrix Md;
Md.width  = WIDTH;
Md.height = WIDTH;
Md.pitch  = WIDTH;
int size = WIDTH * WIDTH * sizeof(float);
cudaMalloc((void**)&Md.elements, size);

// Copy M from the host to the device
cudaMemcpy(Md.elements, M.elements, size,
   cudaMemcpyHostToDevice);

// Read M from the device to the host into P
cudaMemcpy(P.elements, Md.elements, size,
   cudaMemcpyDeviceToHost);
...
// Free device memory
cudaFree(Md.elements);
```

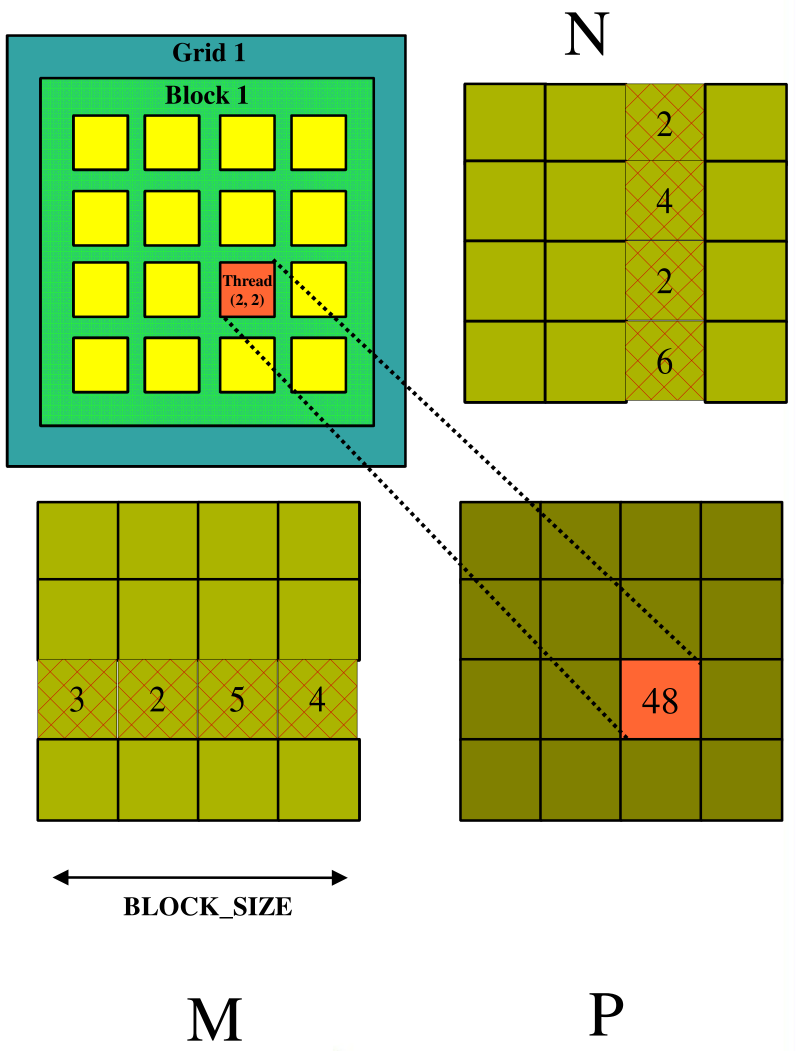
Georgia Tech College of Computing

Step 2: Matrix Multiplication A Simple Host Code in C

```
// Matrix multiplication on the (CPU) host in double precision
// for simplicity, we will assume that all dimensions are equal

void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
    for (int i = 0; i < M.height; ++i)
        for (int j = 0; j < N.width; ++j) {
            double sum = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k];
                double b = N.elements[k * N.width + j];
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
}
```

Multiply Using One Thread Block

- One Block of threads compute 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



Step 3: Matrix Multiplication Host-side Main Program Code

```
int main(void) {
// Allocate and initialize the matrices
    Matrix  M  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  N  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  P  = AllocateMatrix(WIDTH, WIDTH, 0);

// M * N on the device
    MatrixMulOnDevice(M, N, P);

// Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);
return 0;
}
```

Step 3: Matrix Multiplication Host-side code

```
// Matrix multiplication on the device
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P)
{
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);
    CopyToDeviceMatrix(Pd, P); // Clear memory
```

Georgia Tech College of Computing

Step 3: Matrix Multiplication Host-side Code (cont.)

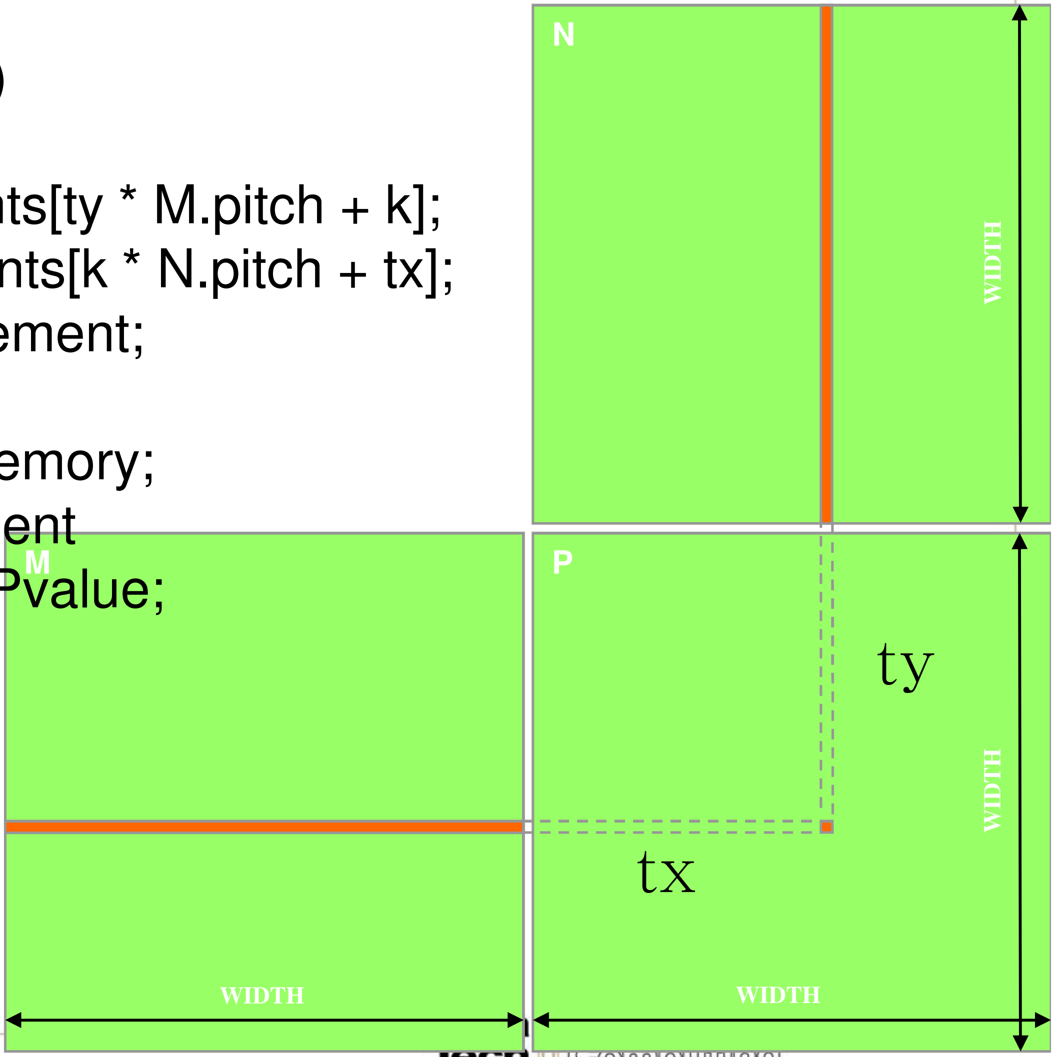
```
// Setup the execution configuration
    dim3 dimBlock(WIDTH, WIDTH);
    dim3 dimGrid(1, 1);

    // Launch the device computation threads!
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

    // Read P from the device
    CopyFromDeviceMatrix(P, Pd);

    // Free device matrices
    FreeDeviceMatrix(Md);
    FreeDeviceMatrix(Nd);
    FreeDeviceMatrix(Pd);
}
```

Step 4: Matrix Multiplication Device-side Kernel Function

```
// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P)
{
    // 2D Thread ID
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
```

Step 4: 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 * P.pitch + tx] = Pvalue;
}
```



Step 5: Some Loose Ends

```
// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M)
{
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMalloc((void**)&Mdevice.elements, size);
    return Mdevice;
}

// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}

void FreeMatrix(Matrix M) {
    free(M.elements);
}
```

Step 5: Some Loose Ends (cont.)

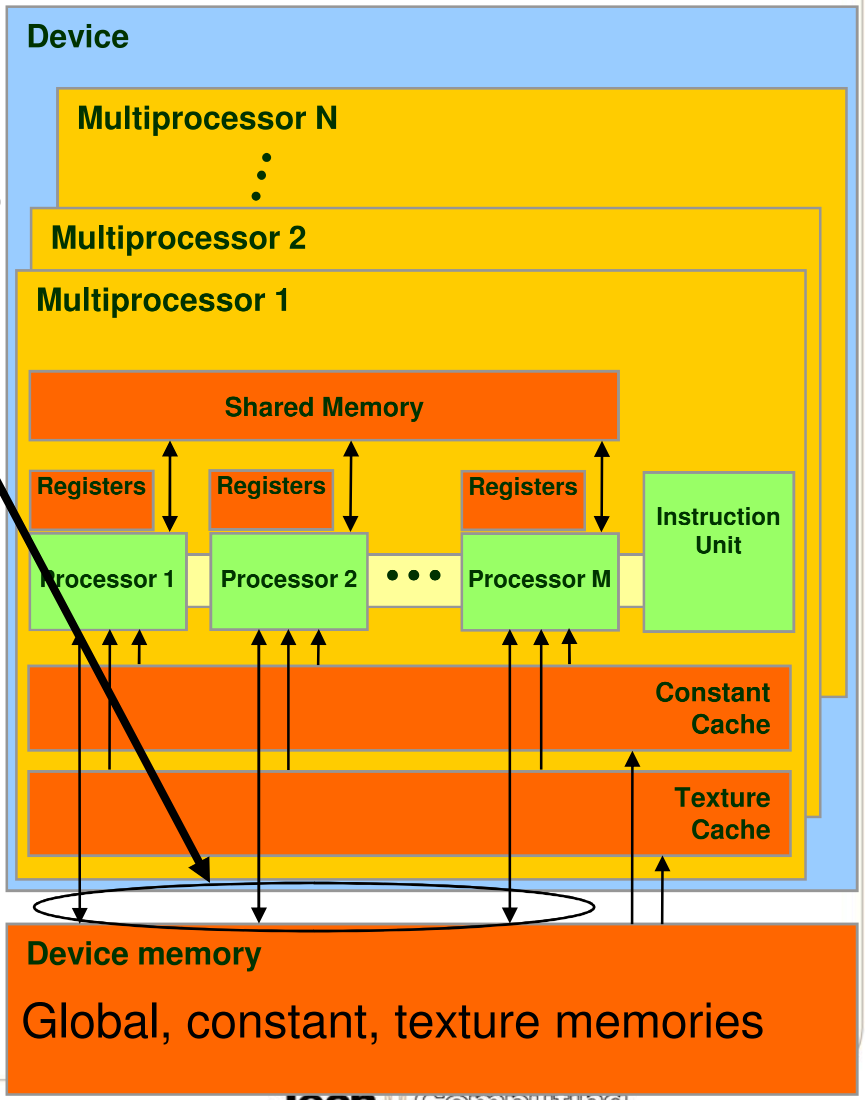
```
// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost)
{
    int size = Mhost.width * Mhost.height * sizeof(float);
    cudaMemcpy(Mdevice.elements, Mhost.elements, size,
            cudaMemcpyHostToDevice);
}

// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice)
{
    int size = Mdevice.width * Mdevice.height * sizeof(float);
    cudaMemcpy(Mhost.elements, Mdevice.elements, size,
            cudaMemcpyDeviceToHost);
}
```

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

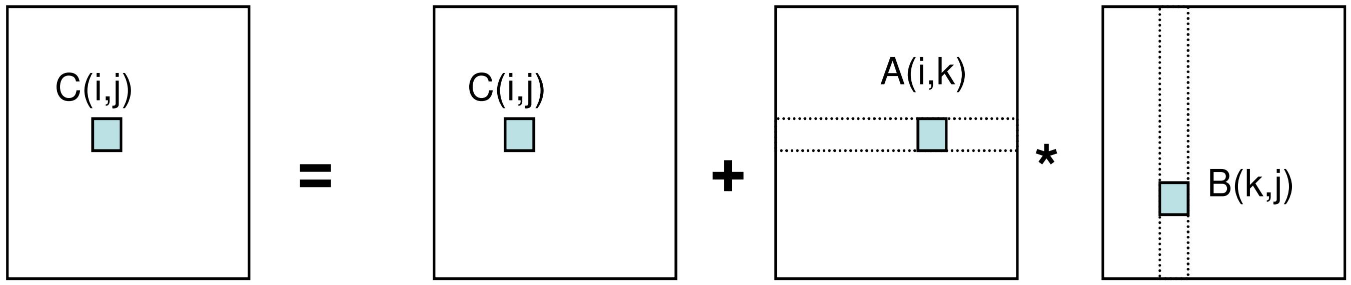
- Each input element is read by WIDTH threads.
- If we load each element into Shared Memory and have several threads use the local version, we can drastically reduce the memory bandwidth
  - Load all the matrix ?
  - Tiled algorithms

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

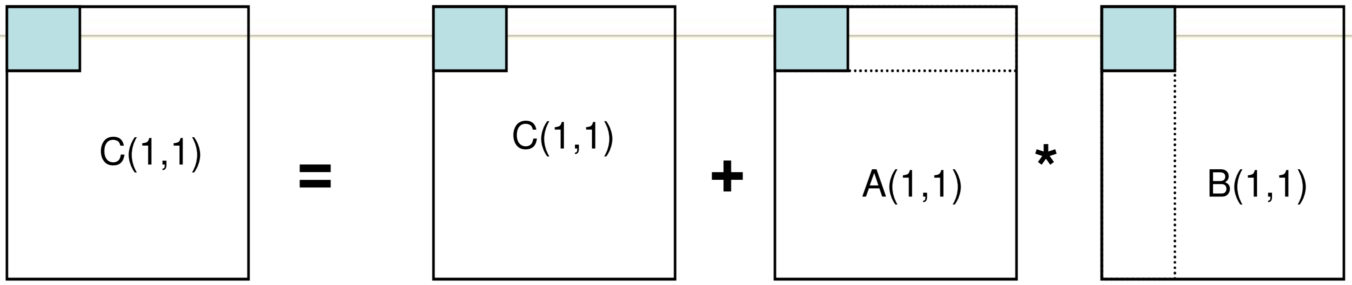
Blocked (Tiled) Matrix Multiply

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

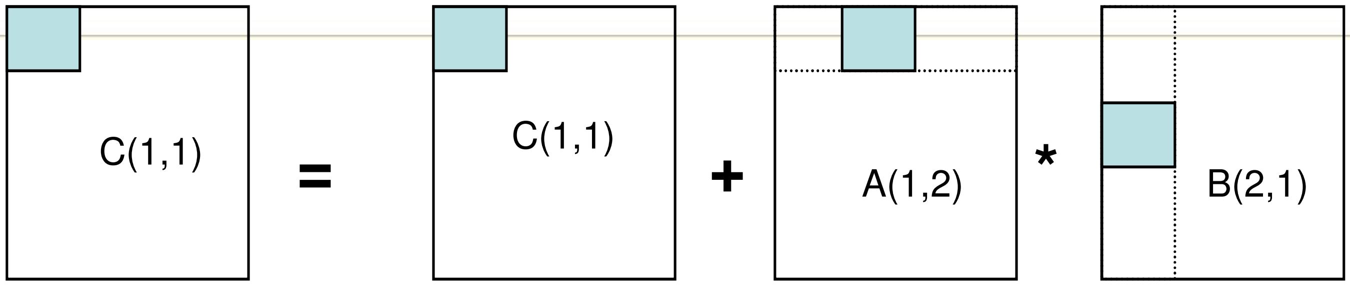
```
for i = 1 to N
   for j = 1 to N
      {read block C(i,j) into shared memory}
      for k = 1 to N
           {read block A(i,k) into shared memory}
           {read block B(k,j) into shared memory}
            C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
       {write block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



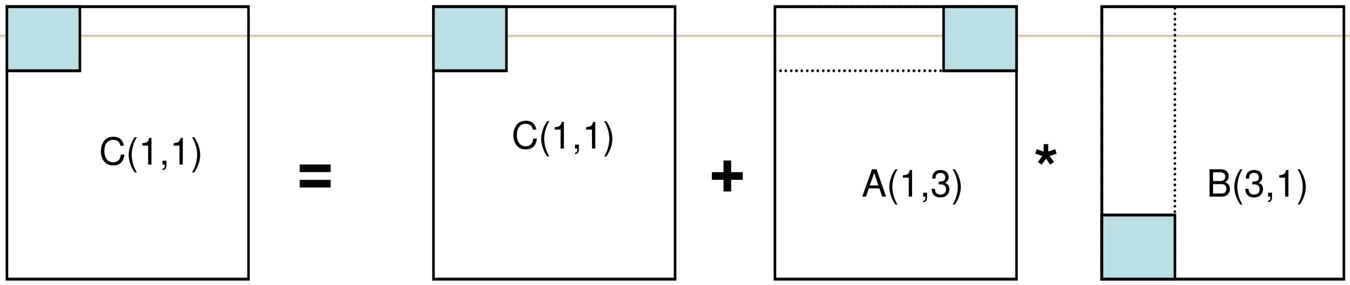
Blocked (Tiled) Matrix Multiply



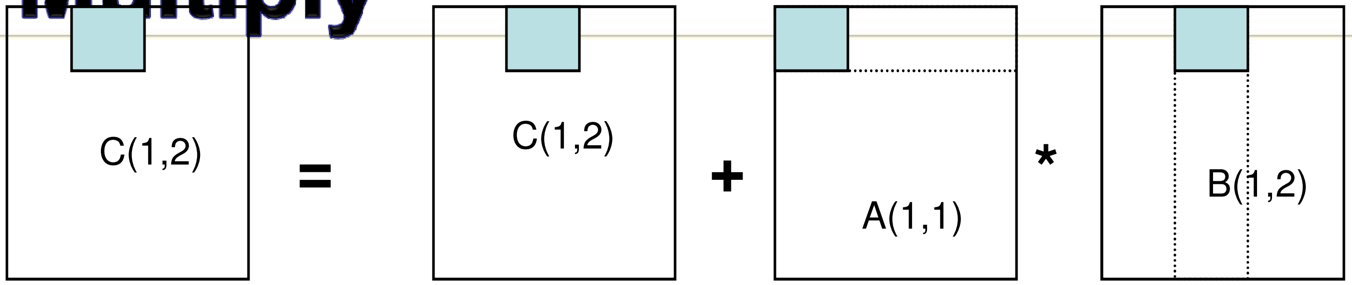
Blocked (Tiled) Matrix Multiply



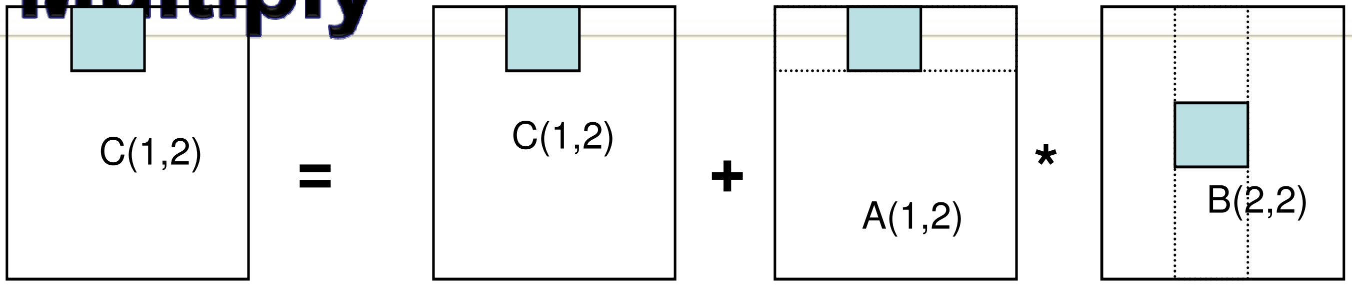
Blocked (Tiled) Matrix Multiply



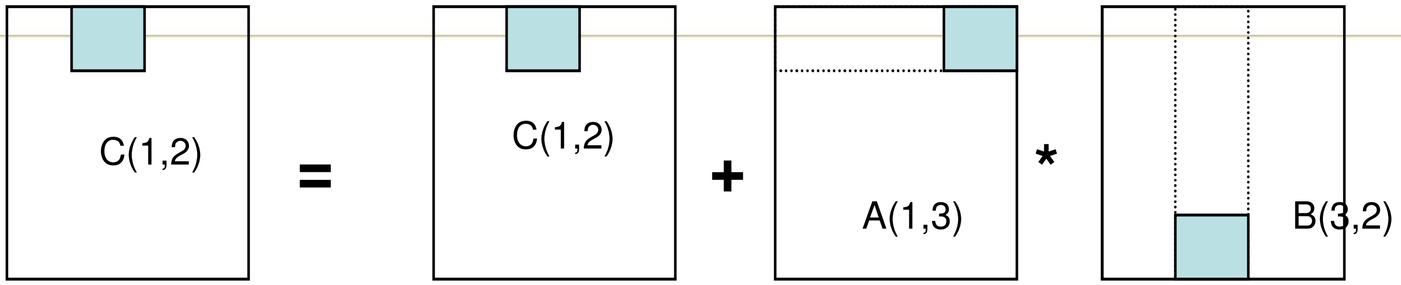
Blocked (Tiled) Matrix Multiply



Blocked (Tiled) Matrix Multiply



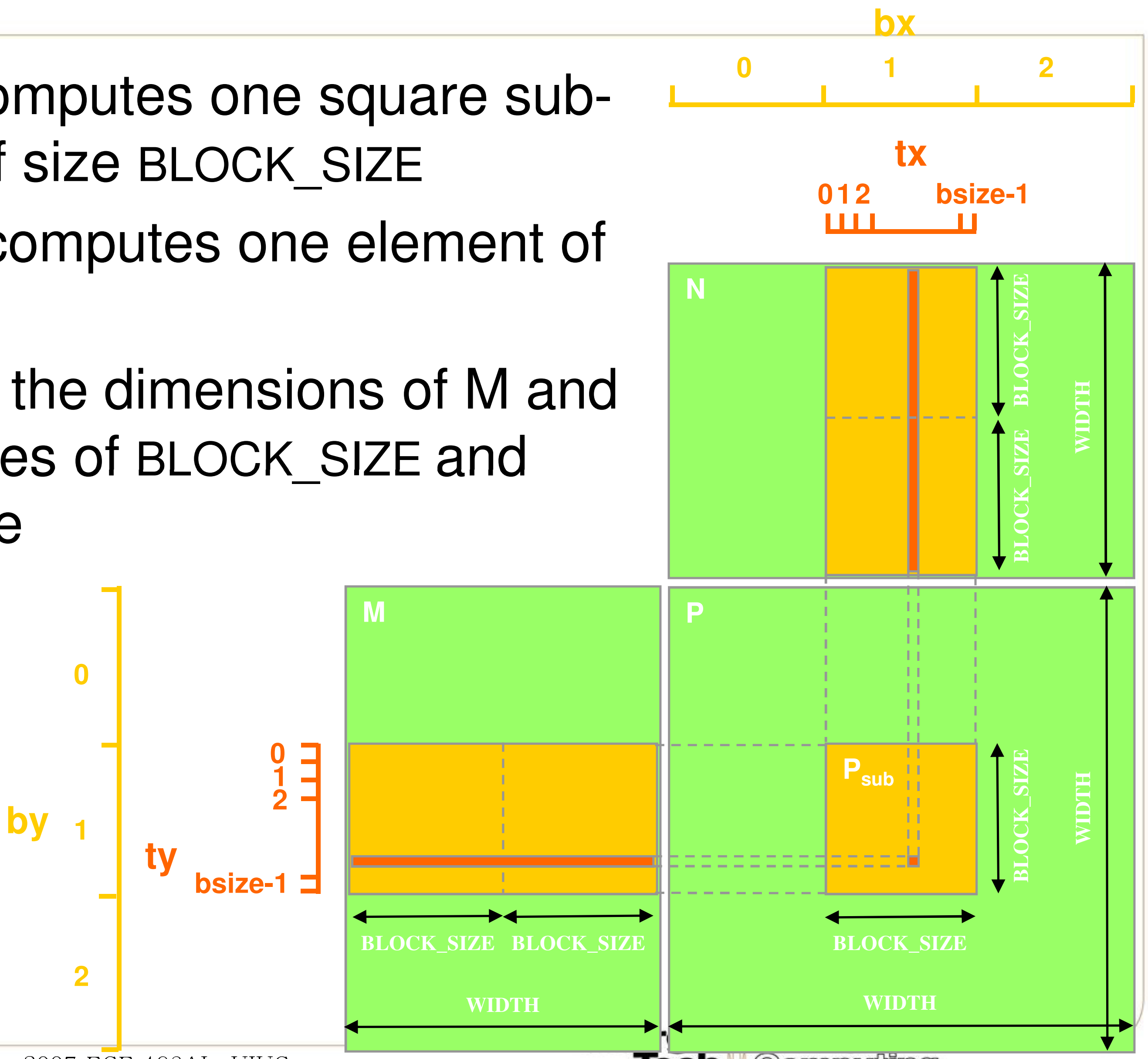
Blocked (Tiled) Matrix Multiply



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

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);
```

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

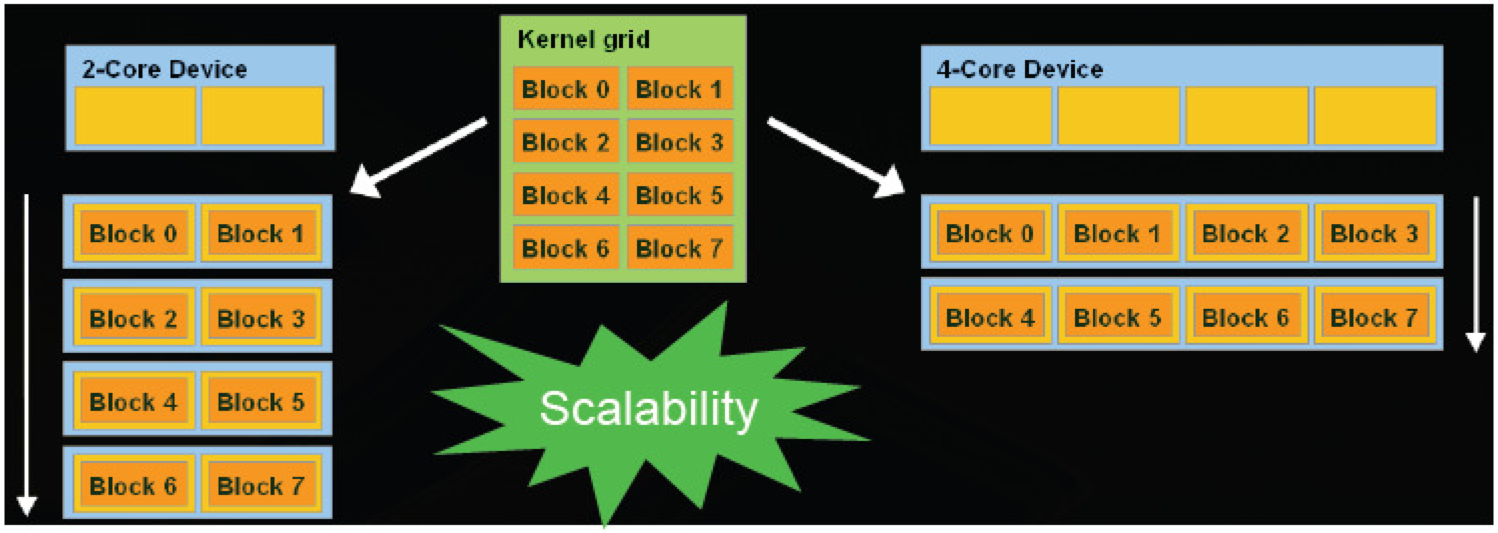
- When running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. `printf`) and vice-versa
  - Detect deadlock situations caused by improper usage of `__syncthreads`

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

Blocks must be Indepdent

- Blocks may coordinate but not synchronize
  - Shared queue pointer:OK
  - Shared block: Bad…
- Thread blocks can run in any order
  - Concurrently or sequentially
  - Facilitates scaling of the same code across many devices



Linking

- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (`cudart`)
  - The CUDA core library (`cuda`)

- Some Additional API Features

Language Extensions: Built-in Variables

- `dim3 gridDim;`
  - Dimensions of the grid in blocks (`gridDim.z` unused)
- `dim3 blockDim;`
  - Dimensions of the block in threads
- `dim3 blockIdx;`
  - Block index within the grid
- `dim3 threadIdx;`
  - Thread index within the block

Common Runtime Component

- Provides:
  - Built-in vector types
  - A subset of the C runtime library supported in both host and device codes

Common Runtime Component: Built-in Vector Types

- `[u]char[1..4], [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4]`
  - Structures accessed with `x`, `y`, `z`, `w` fields:

```
        uint4 param;
        int y = param.y;
```

- `dim3`
  - Based on `uint3`
  - Used to specify dimensions

Common Runtime Component: Mathematical Functions

- `pow`, `sqrt`, `cbrt`, `hypot`
- `exp`, `exp2`, `expm1`
- `log`, `log2`, `log10`, `log1p`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
- `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- `ceil`, `floor`, `trunc`, `round`
- Etc.
  - When executed on the host, a given function uses the C runtime implementation if available
  - These functions are only supported for scalar types, not vector types

Host Runtime Component

- Provides functions to deal with:
  - Device management (including multi-device systems)
  - Memory management
  - Error handling
- Initializes the first time a runtime function is called
- A host thread can invoke device code on only one device
  - Multiple host threads required to run on multiple devices

Host Runtime Component: Memory Management

- Device memory allocation
  - `cudaMalloc(), cudaFree()`
- Memory copy from host to device, device to host, device to device
  - `cudaMemcpy(), cudaMemcpy2D(), cudaMemcpyToSymbol(), cudaMemcpyFromSymbol()`
- Memory addressing
  - `cudaGetSymbolAddress()`

Device Runtime Component: Mathematical Functions

- Some mathematical functions (e.g. `sin(x)`) have a less accurate, but faster device-only version (e.g. `__sin(x)`)
  - `__pow`
  - `__log`, `__log2`, `__log10`
  - `__exp`
  - `__sin`, `__cos`, `__tan`
- SFU