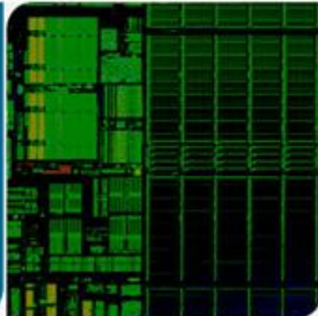
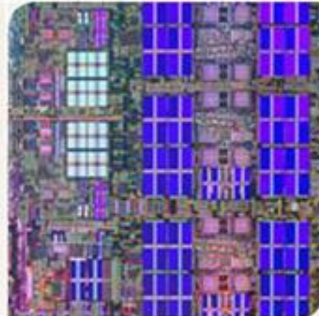
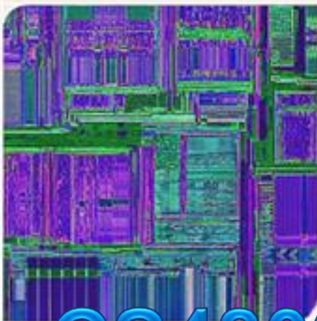




# CS4803DGC Design and Programming of Game Consoles

Spring 2011

Prof. Hyesoon Kim

Georgia Tech

College of Computing

# CUDA

- “Compute Unified Device Architecture”
- General propose computing with graphics hardware
- CUDA →Nvidia, Intel (kind of…)
- OpenCL → most of platforms
- Not so much for game applications
- Why are we studying?
  - GPU consoles
  - Parallel programming

# CUDA

Main Memory

CPU

1 Copy processing data

Instruct the processing 2

4 Copy the result

Memory for GPU

GPU (GeForce 8800)

Execute parallel in each core 3

Processing flow on CUDA

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# 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

Host

Input Assembler

Thread Execution Manager

| Parallel Data Cache | Parallel Data Cache | Parallel Data Cache | Parallel Data Cache | Parallel Data Cache | Parallel Data Cache | Parallel Data Cache | Parallel Data Cache |
|---|---|---|---|---|---|---|---|
| Texture | Texture | Texture | Texture | Texture | Texture | Texture | Texture |

Load/store Load/store Load/store Load/store Load/store Load/store

Global Memory

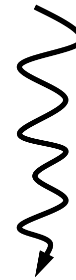
© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

Georgia Tech College of Computing

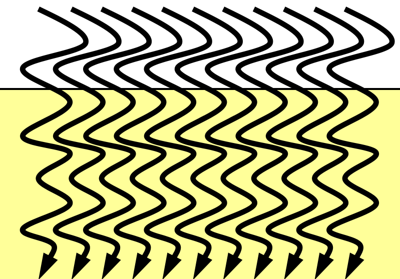
# Code Example (HelloWorld)

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

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

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

Executed at CPU



Executed at GPU
Many threads



```
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**

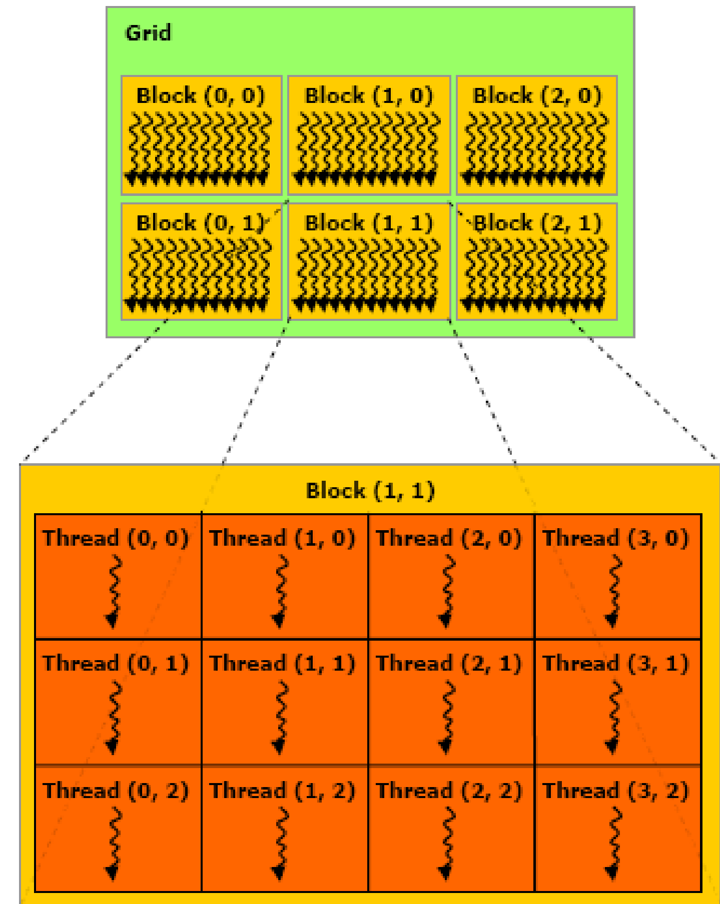
```
__device__ float filter[N];

__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);
```

Georgia Tech College of Computing

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# 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

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

**C Program Sequential Execution**

`Serial code`

Host

`Parallel kernel`

`Kernel0<<<>>>()`

Device

Grid 0

| Block (0, 0) | Block (1, 0) | Block (2, 0) |
| --- | --- | --- |
| Block (0, 1) | Block (1, 1) | Block (2, 1) |

`Serial code`

Host

`Parallel kernel`

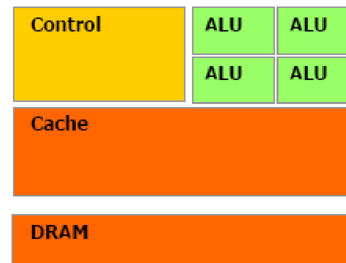
`Kernel1<<<>>>()`

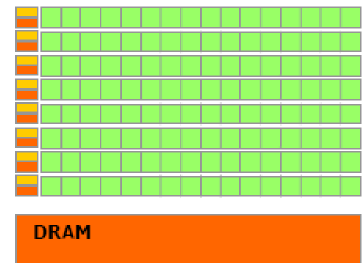
Device

Grid 1

| Block (0, 0) | Block (1, 0) |
| --- | --- |
| Block (0, 1) | Block (1, 1) |
| Block (0, 2) | Block (1, 2) |

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

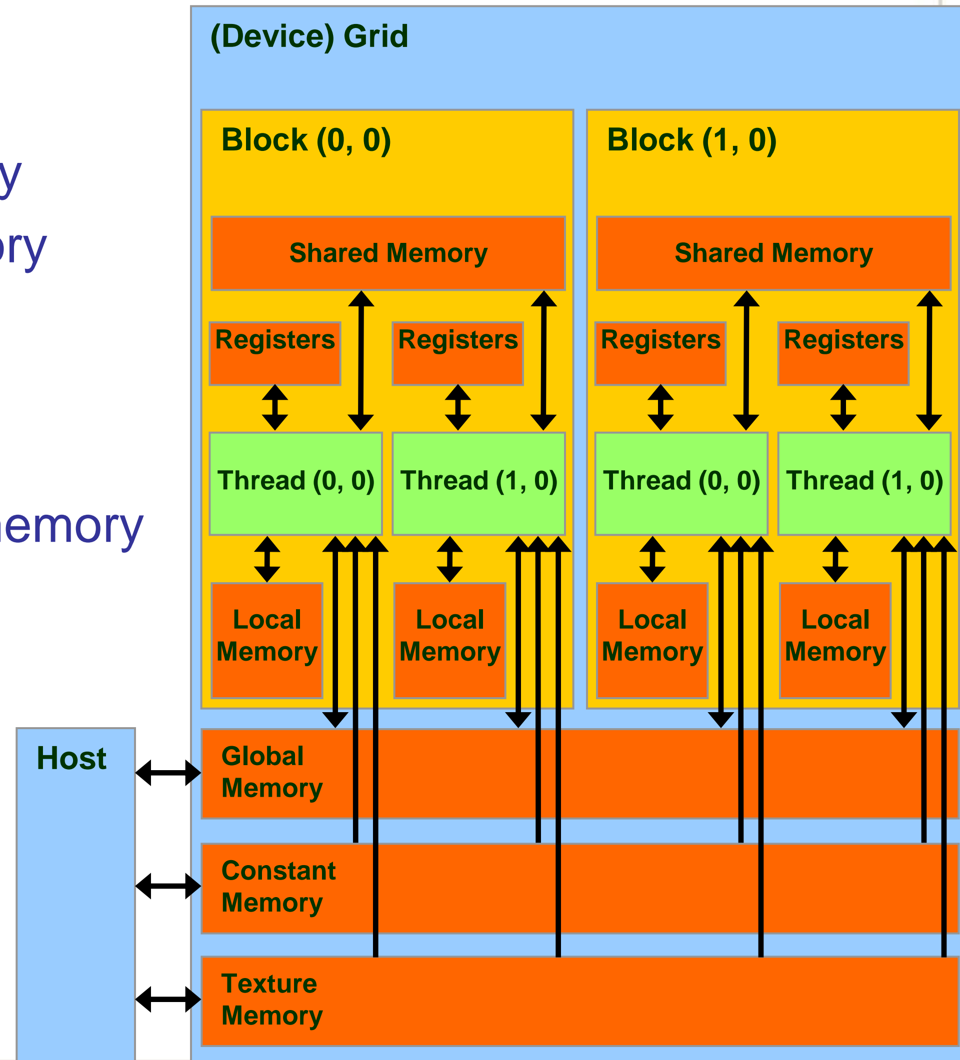


# CUDA Device Memory Space Overview

- 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

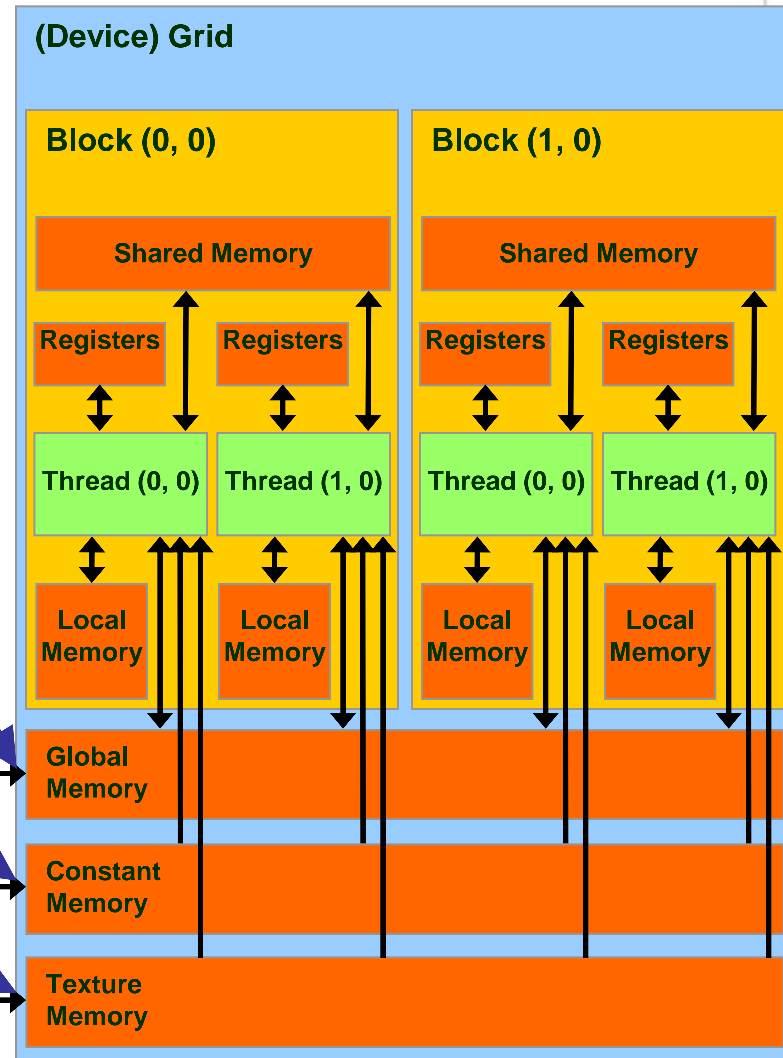
- The host can R/W global, constant, and texture memories




© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

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

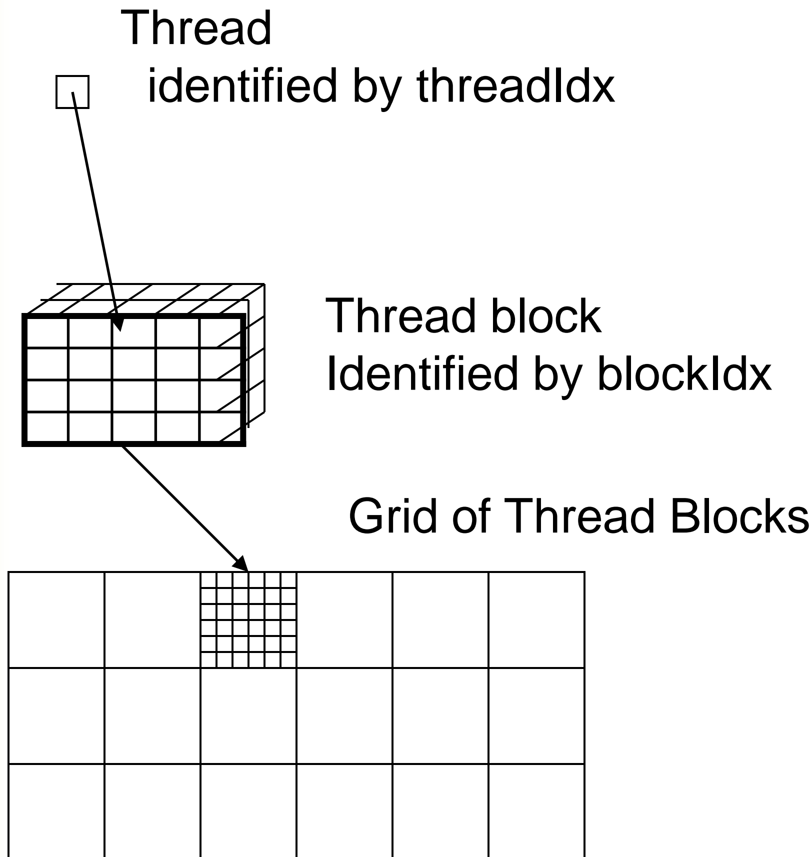
- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
- Texture and Constant Memories
  - Constants initialized by host
  - Contents visible to all threads




Courtesy: NDVIA

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# 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()

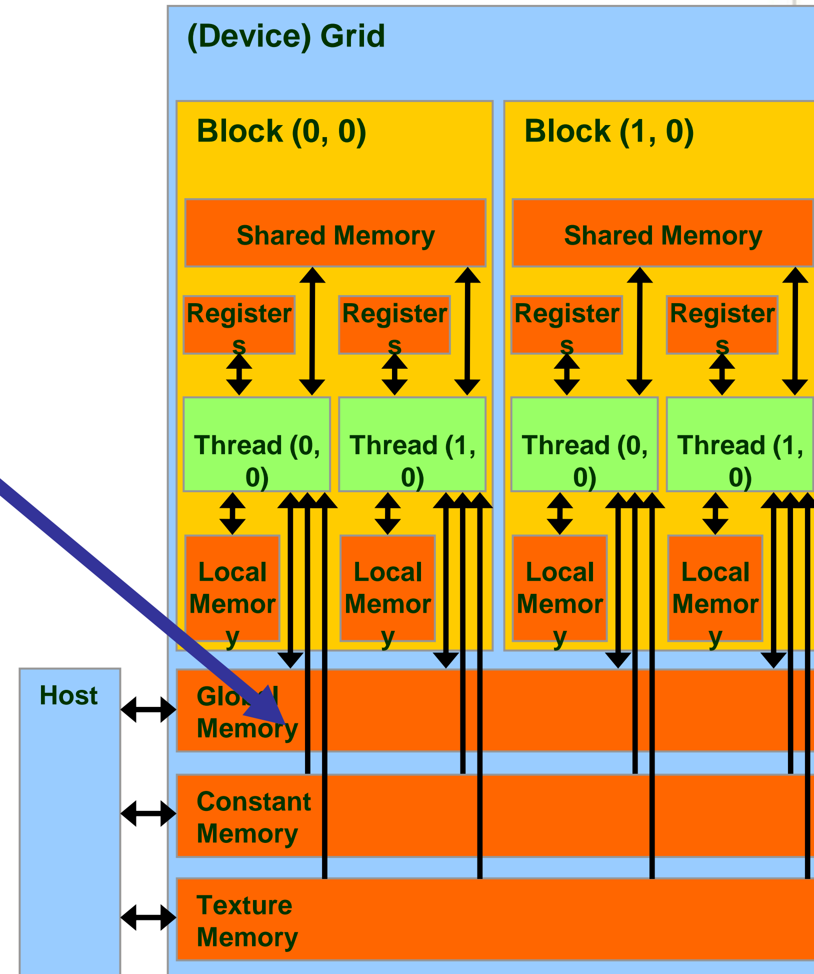
-Grid of thread blocks

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

- CUDA – API

# CUDA Device Memory Allocation

- cudaMalloc()
  - Allocates object in the device Global Memory
  - 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

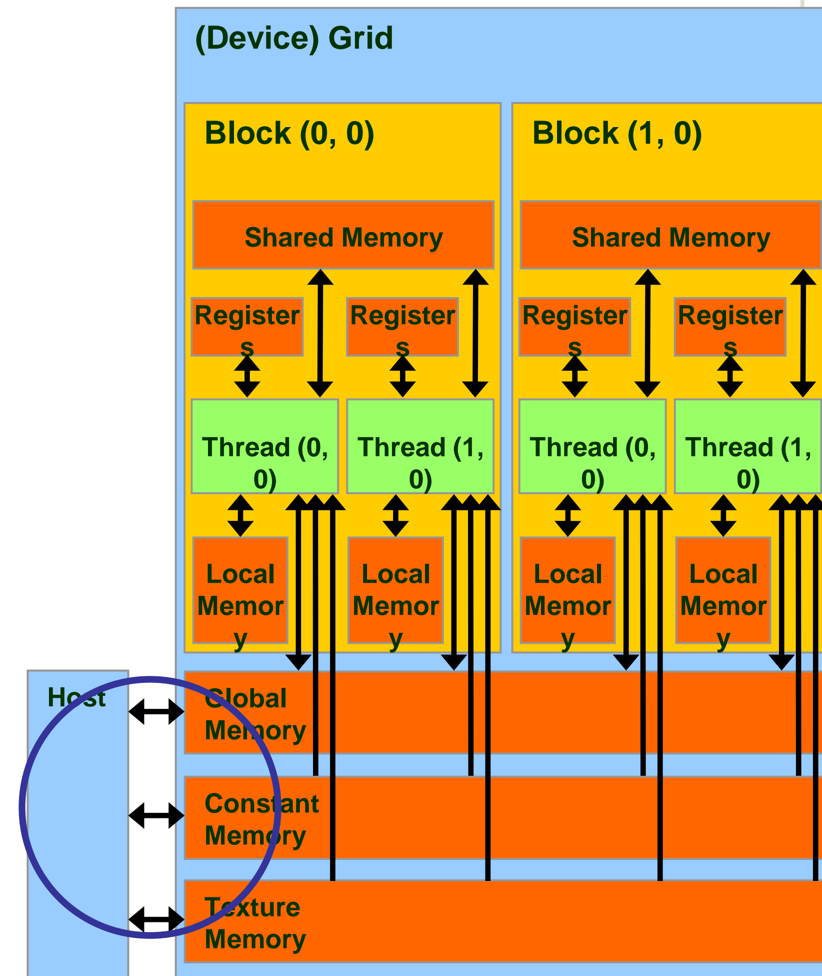



Georgia Tech College of Computing

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# 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
- Fast memcpy (later lectures)
  - cudaMemcpyAsync ()
  - Page-Locked Host Memory




© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

Georgia Tech College of Computing

# 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;
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# 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);
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# 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);
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# 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`

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# CUDA Function Declarations (cont.)

- `__device__` functions cannot have their address taken
- For functions executed on the device:
  - No recursion → Cuda 3.0 (Fermi) supports recursion
  - No static variable declarations inside the function
  - No variable number of arguments

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# Review: Calling a Kernel Function – Thread Creation

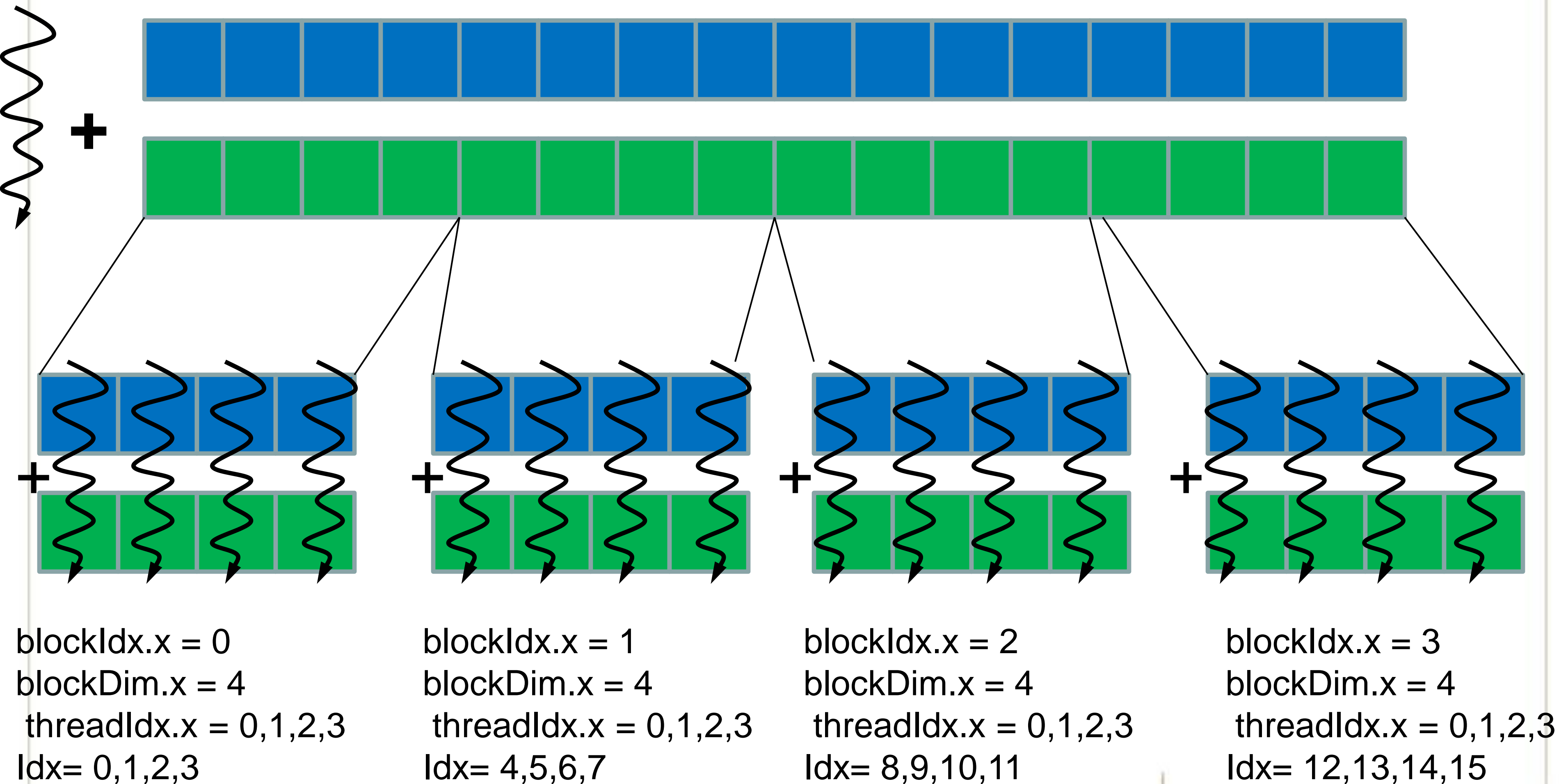
- A kernel function must be called with an execution configuration:

```
__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
   >>>(...);
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# Elementwise Matrix Addition

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




Georgia Tech College of Computing

# 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);
}
```

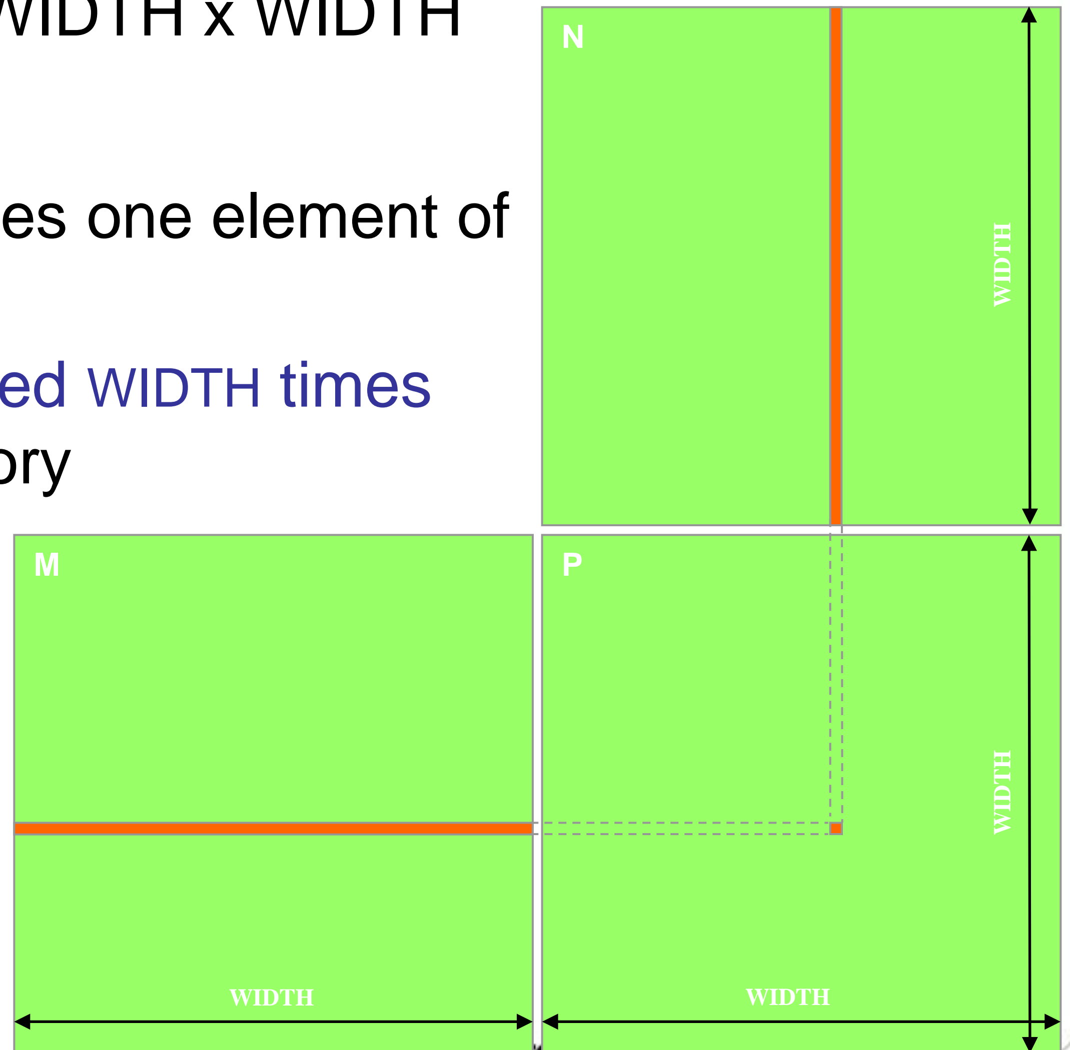
# A Simple Running Example Matrix Multiplication

- 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

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# 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




Georgia Tech College of Computing

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# 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);
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

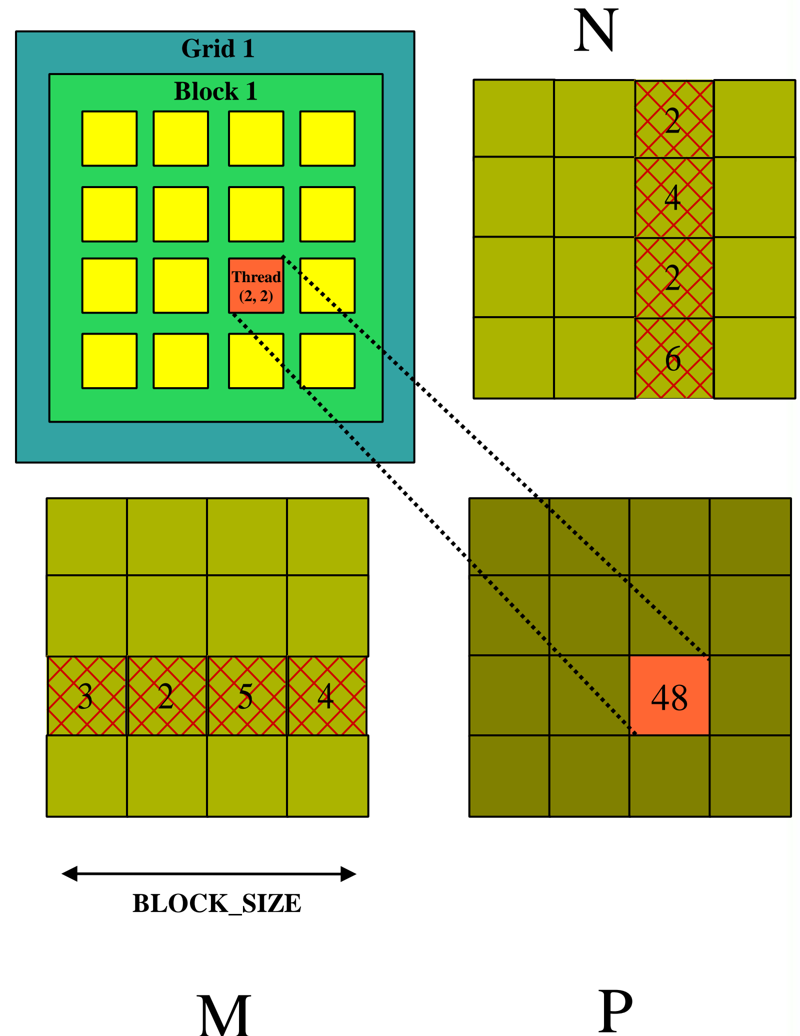
# Step 2: Matrix Multiplication A Simple Host Code in C

```
// 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;
        }
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# 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




Georgia Tech College of Computing

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# 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;
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# 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
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# 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);
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

Georgia Tech College of Computing

# 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;
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# 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;
}
```



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# 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);
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# 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);
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007 ECE 498AL, UIUC

# Lab 0.1

- Set your environment to run CUDA
  - Use Braid Lab
  - Linux or windows
- Run your own matrix multiplication code
- Compare the performance between CPU and GPU
- 10x10, 100x100  matrix size.

- Student's Information Sheet (1/28)
- Make-up class schedule

  Wed: 1/26, 6-7 pm

  location: TBD