

# GPU Programming Using CUDA

Gregory G. Howes  
Department of Physics and Astronomy  
University of Iowa

Iowa High Performance Computing Summer School  
University of Iowa  
Iowa City, Iowa  
6-8 June 2012



# Thank you



Ben Rogers

Glenn Johnson

Mary Grabe

Amir Bozorgzadeh

Preston Smith

Information Technology Services

Information Technology Services

Information Technology Services

Information Technology Services

Purdue University

and

**National Science Foundation**

Rosen Center for Advanced Computing, Purdue University

Great Lakes Consortium for Petascale Computing

# Outline

- Concepts for GPU Computing
- Programming Model for GPU Computing using CUDA C
- CUDA C Programming
- Advanced CUDA Capabilities

# GPU Computing

**Graphics Processing Units** (GPUs) have been developed in response to strong market demand for realtime, high-definition 3D graphics (**video games!**)

**GPUs** are **highly parallel**,  
**multithreaded**,  
**manycore** processors

- Tremendous computational horsepower
- Very high memory bandwidth

**We hope to access this power for scientific computing**

# GPU Programming Languages

- **CUDA** (Compute Unified Device Architecture) is the proprietary programming language for NVIDIA GPUs
- **OpenCL** (Open Computing Language) is portable language standard for general computing that can exploit capabilities of GPUs from any manufacturer.

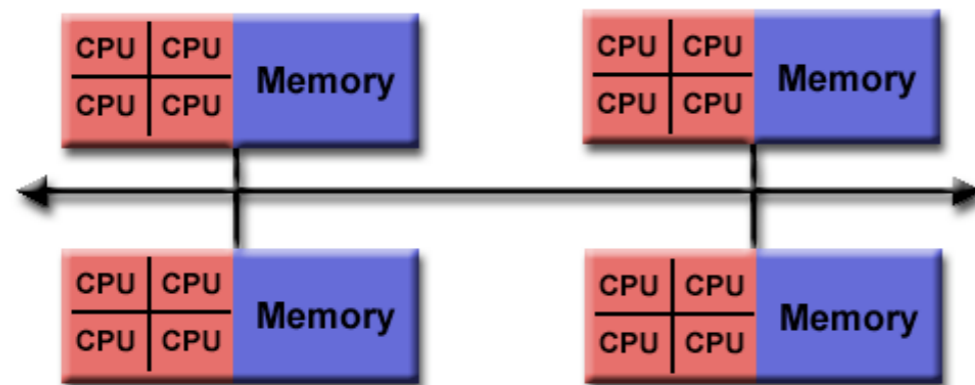
Both languages provide extensions to **C** (as well as other languages) that enable the programmers to access the powerful computing capability for **general-purpose computing** on GPUs (**GPGPU**)

Today we will focus on the basics of **CUDA C programming**

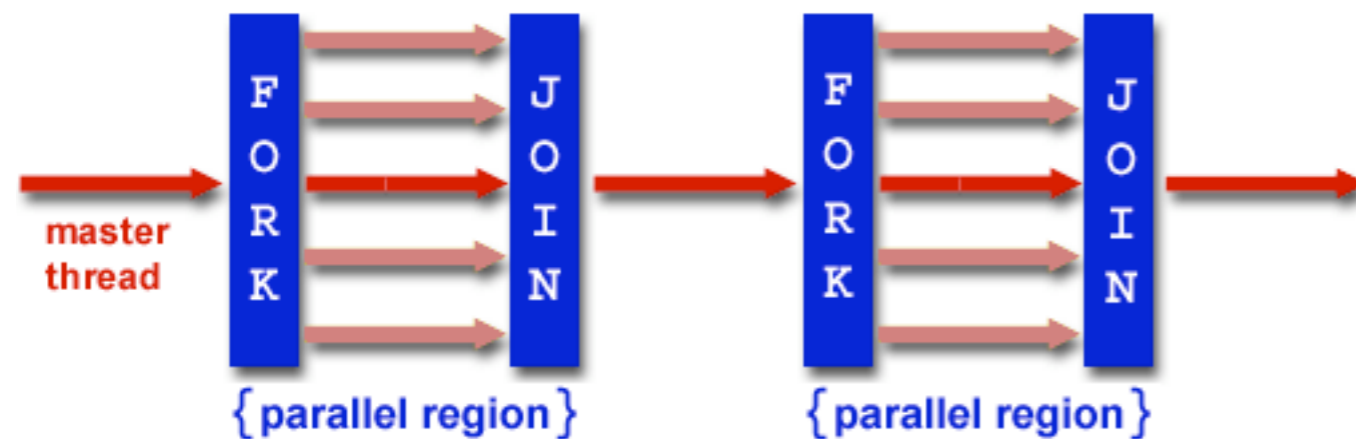
# Parallel Computing Architectures

Different computer architectures suggest three approaches to parallel computing:

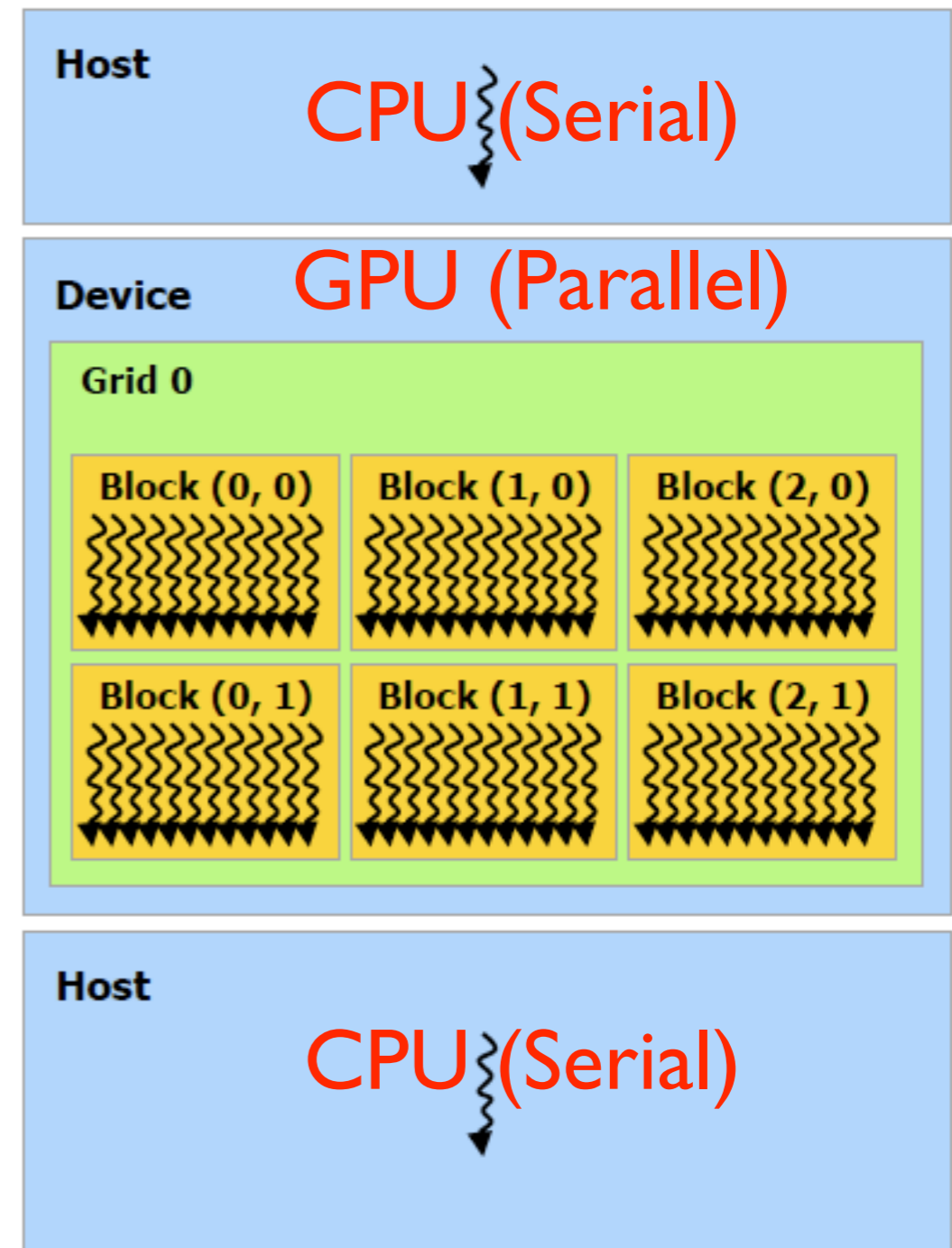
## 1) Message Passing (MPI)



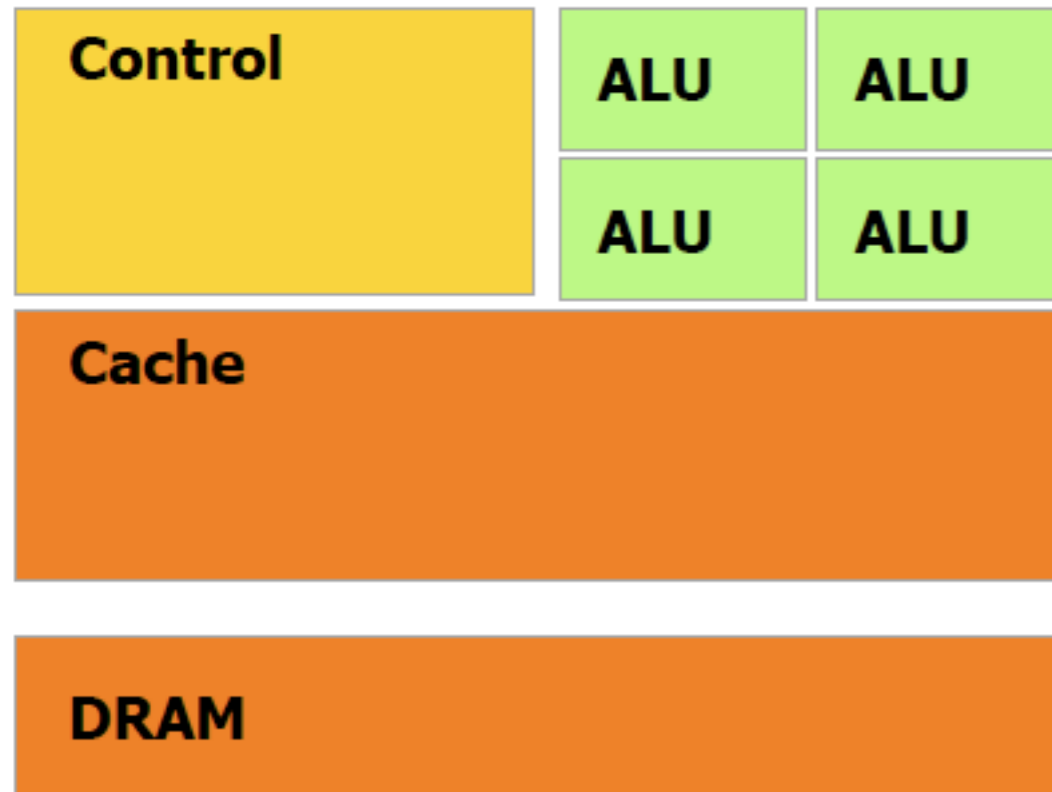
## 2) Multithreading (OpenMP)



## 3) GPU (CUDA)

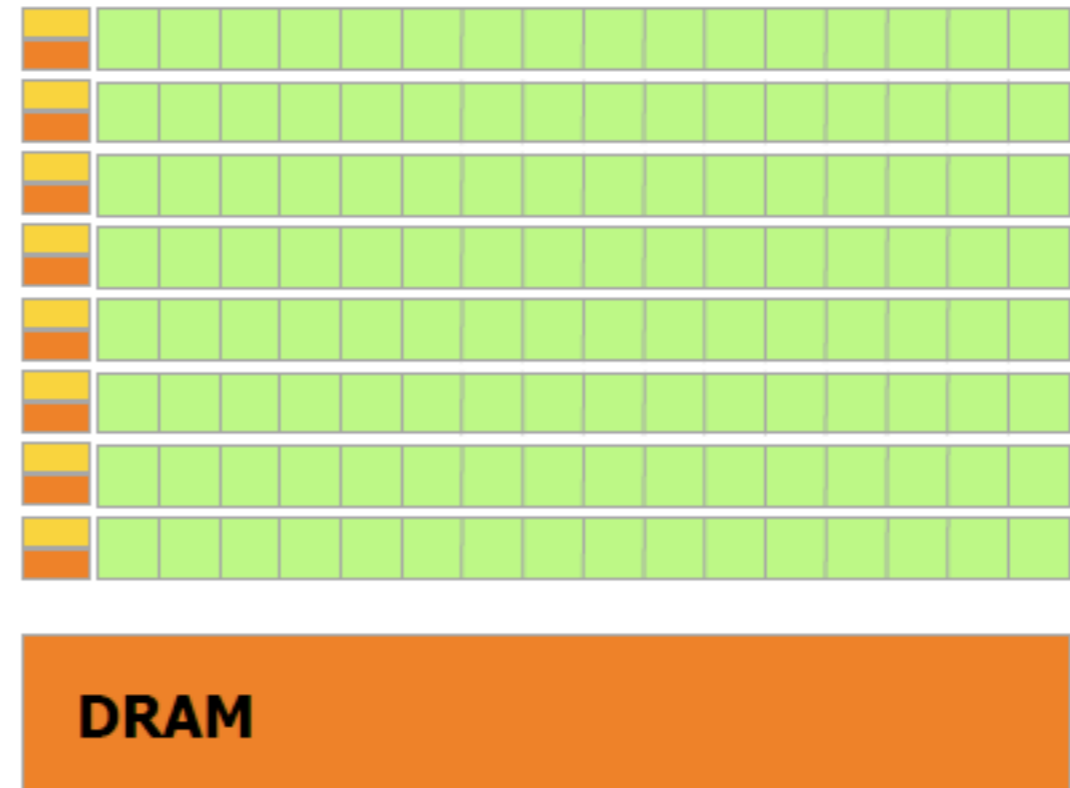


# More Transistors to Data Processing



**CPU**

CPUs devote a significant fraction of transistors to **data caching** and **flow control**



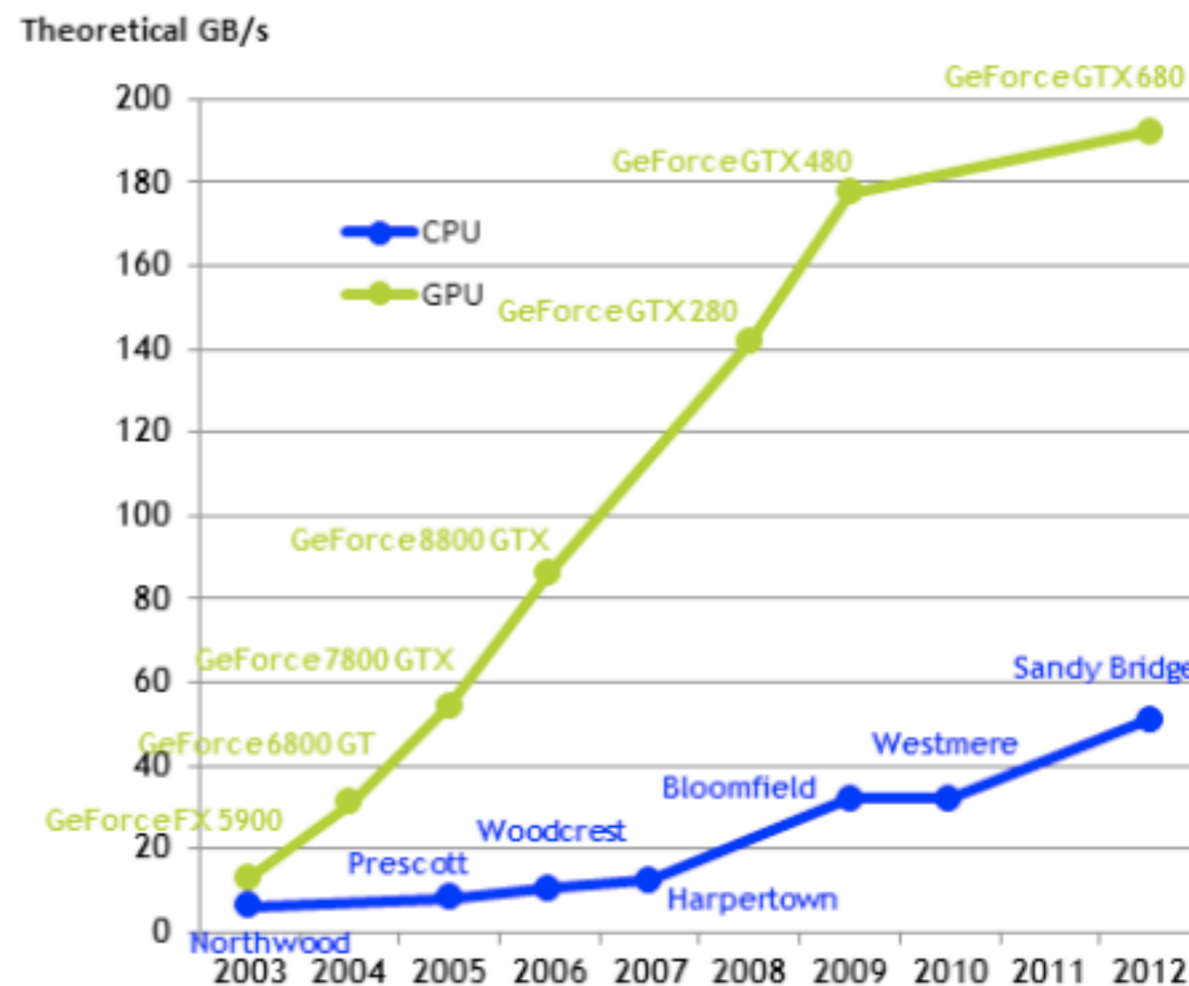
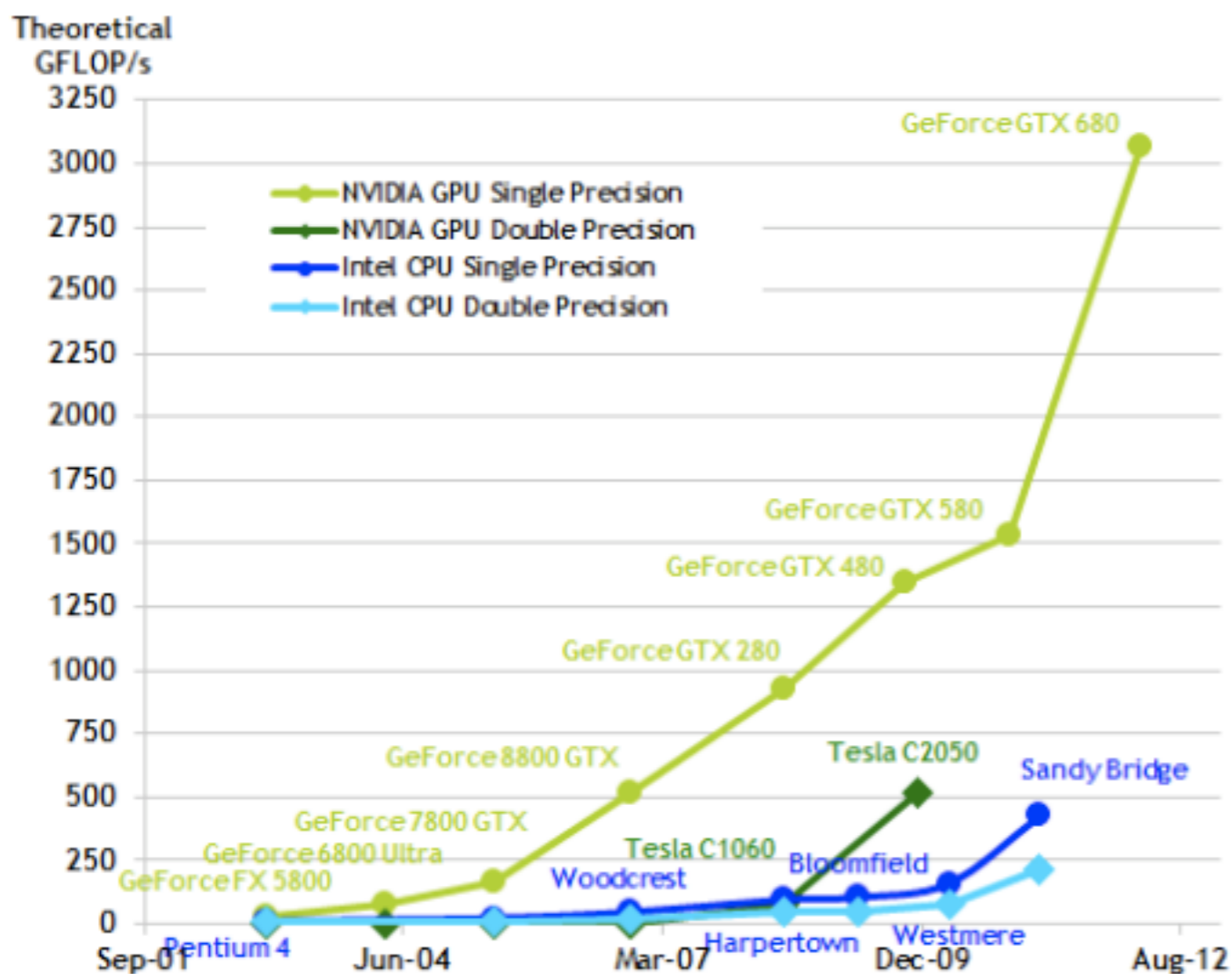
**GPU**

GPUs devote more transistors to **data processing** (arithmetic and logic units, ALU)

# GPU vs. CPU Peak Performance

**F**loating point **O**perations **P**er **S**econd  
(FLOPS)

Memory Access Bandwidth  
(GB/s)





# Many Languages for GPU Computing

## NVIDIA's diverse offerings for GPU Computing

### GPU Computing Applications

#### Libraries and Middleware

cuFFT  
cuBLAS  
cuRAND  
cuSPARSE

LAPACK  
CULA  
MAGMA

Thrust  
NPP  
cuDPP

VSIP  
SVM  
OpenCL

PhysX  
OptiX

iray  
RealityServer

MATLAB  
Mathematica

C

C++

Fortran

Java  
Python  
Wrappers

Direct  
Compute

OpenCL<sup>™</sup>

Directives  
(e.g. OpenACC)



NVIDIA GPU

with the CUDA Parallel Computing Architecture

**Fermi Architecture**  
(compute capabilities 2.x)

GeForce 400 Series

Quadro Fermi Series

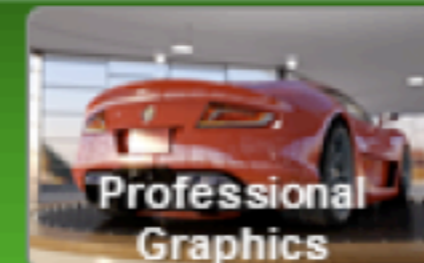
Tesla 20 Series

**Tesla Architecture**  
(compute capabilities 1.x)

GeForce 200 Series  
GeForce 9 Series  
GeForce 8 Series

Quadro FX Series  
Quadro Plex Series  
Quadro NVS Series

Tesla 10 Series



# Outline

- Concepts for GPU Computing
- Programming Model for GPU Computing using CUDA C
- CUDA C Programming
- Advanced CUDA Capabilities

# CUDA C Programming

## CUDA C Programming Language

- Minimal set of extensions to the C programming language
- Core concepts:
  - Hierarchy of thread groups
  - Shared memory
  - Barrier synchronization

### 1) Kernel:

- C function executed  $N$  times in parallel by  $N$  CUDA threads
- Called by the Host (CPU) but executed on the Device (GPU)

### 2) Thread Hierarchy:

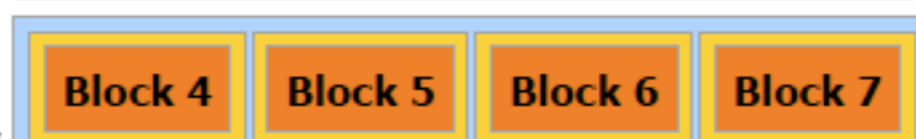
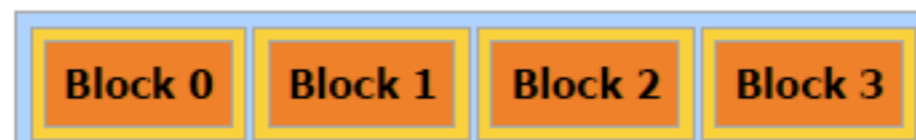
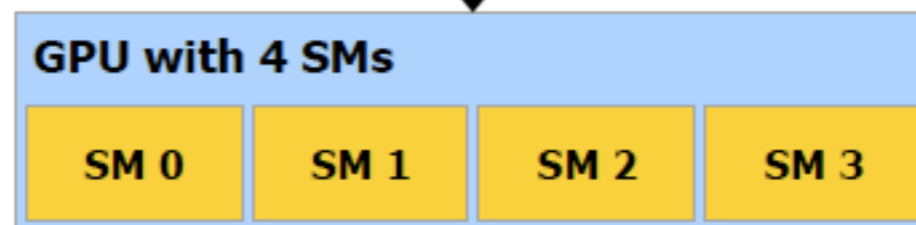
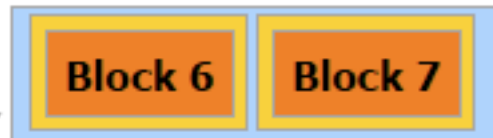
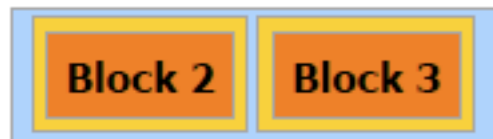
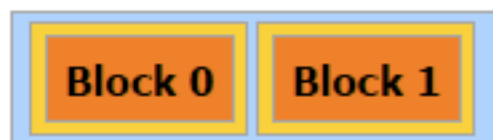
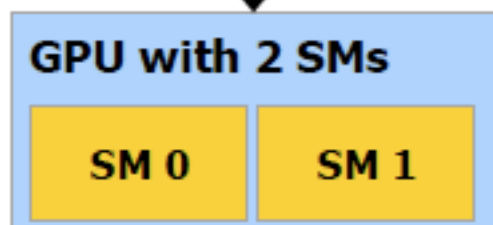
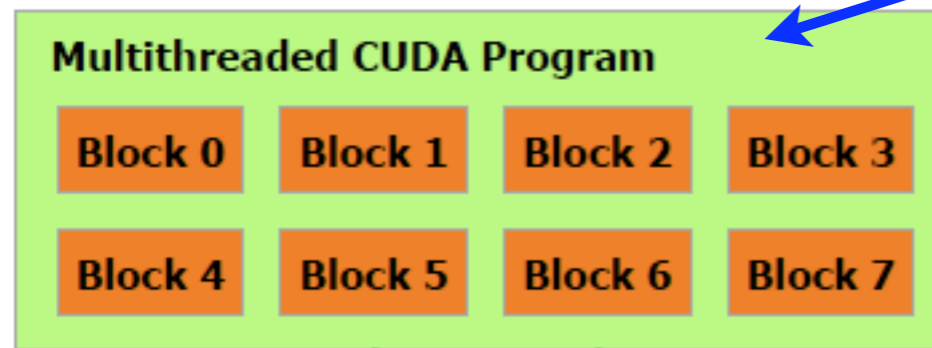
- Grid: Contains many blocks that can be solved independently in parallel
- Block: Contains many threads that can be solve cooperatively in parallel

### 3) Memory Functions:

- Allocate and Free memory space on Device (GPU),
- Copy data from Host (CPU) to Device (GPU), and vice versa

# Thread Hierarchy: Grid of Blocks

**Grid:** Full problem to be solved by GPU is broken into **blocks**

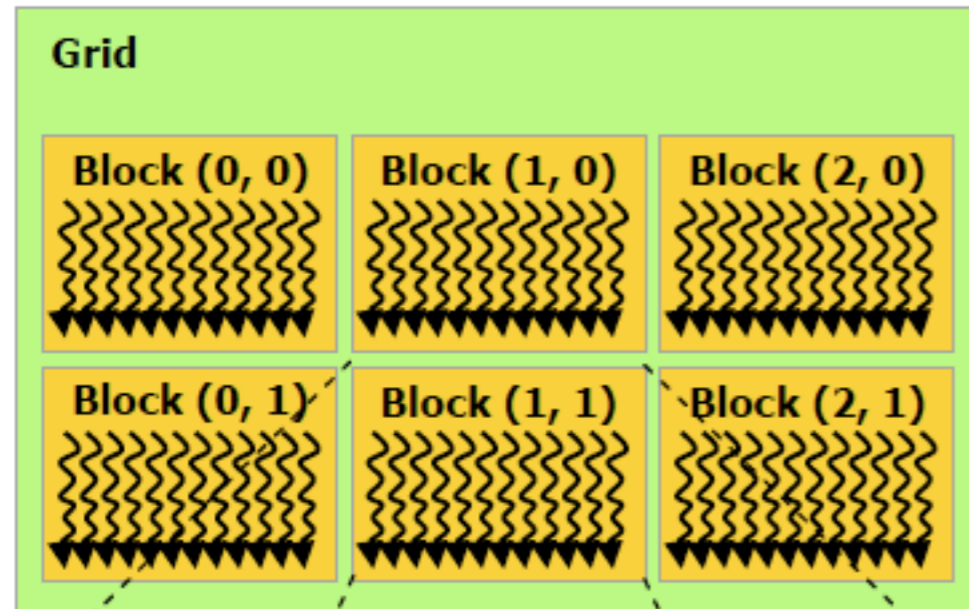


Each **block** is **independent**.  
The blocks can be executed, **in any order**, **concurrently** or **sequentially**

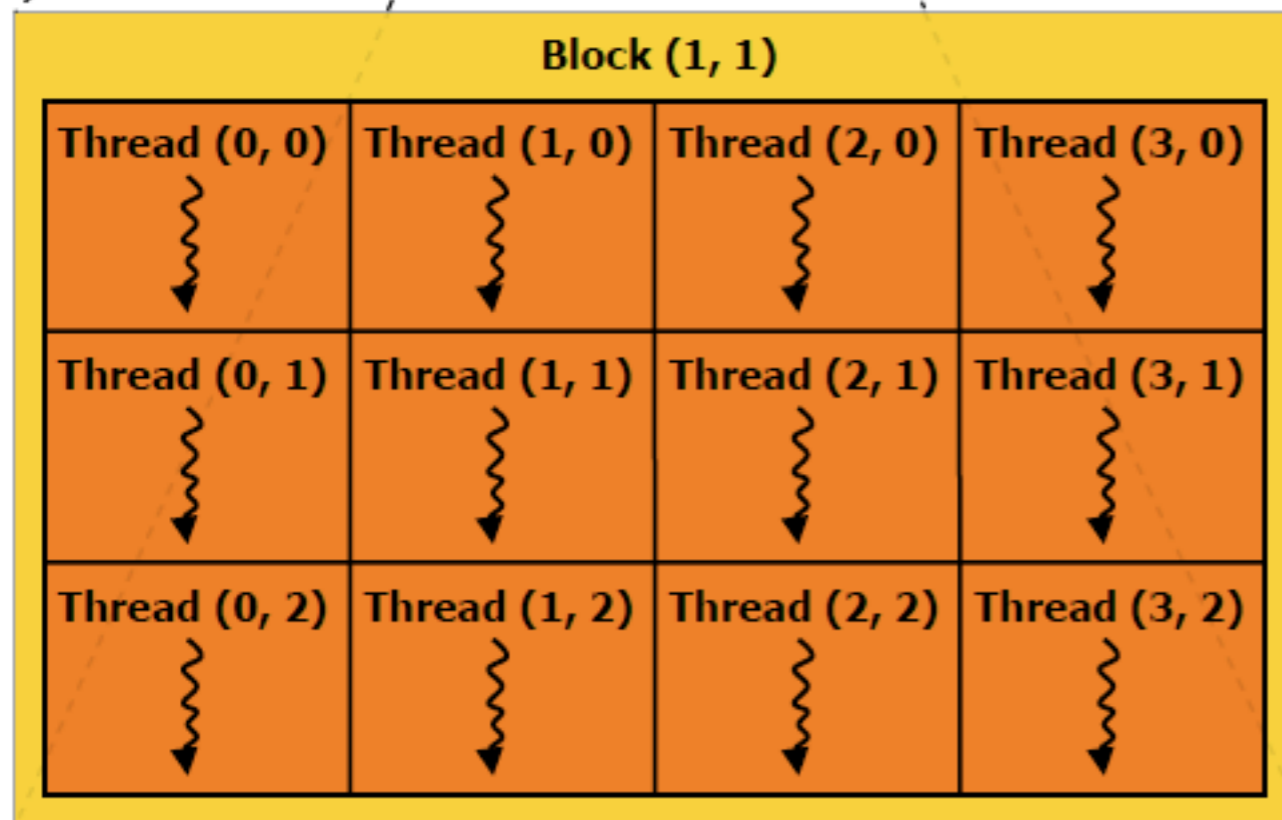
The **Streaming Multiprocessors (SMs)** control the execution of each block

This model enables excellent **scalability** for a varying number of cores per GPU

# Thread Hierarchy: Block of Threads



Each **block** contains many **threads**



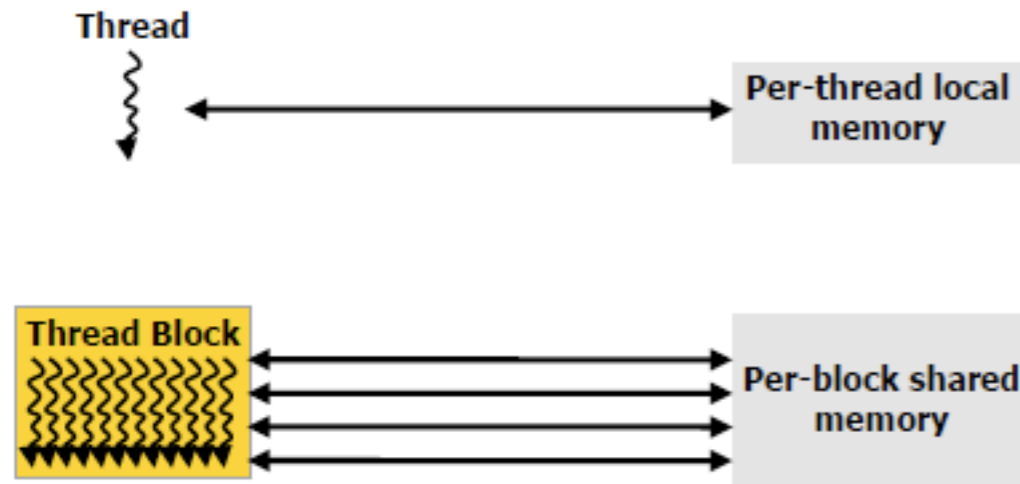
**Threads** within a **block** are executed in parallel, either **cooperatively** or **independently**

# Memory Hierarchy

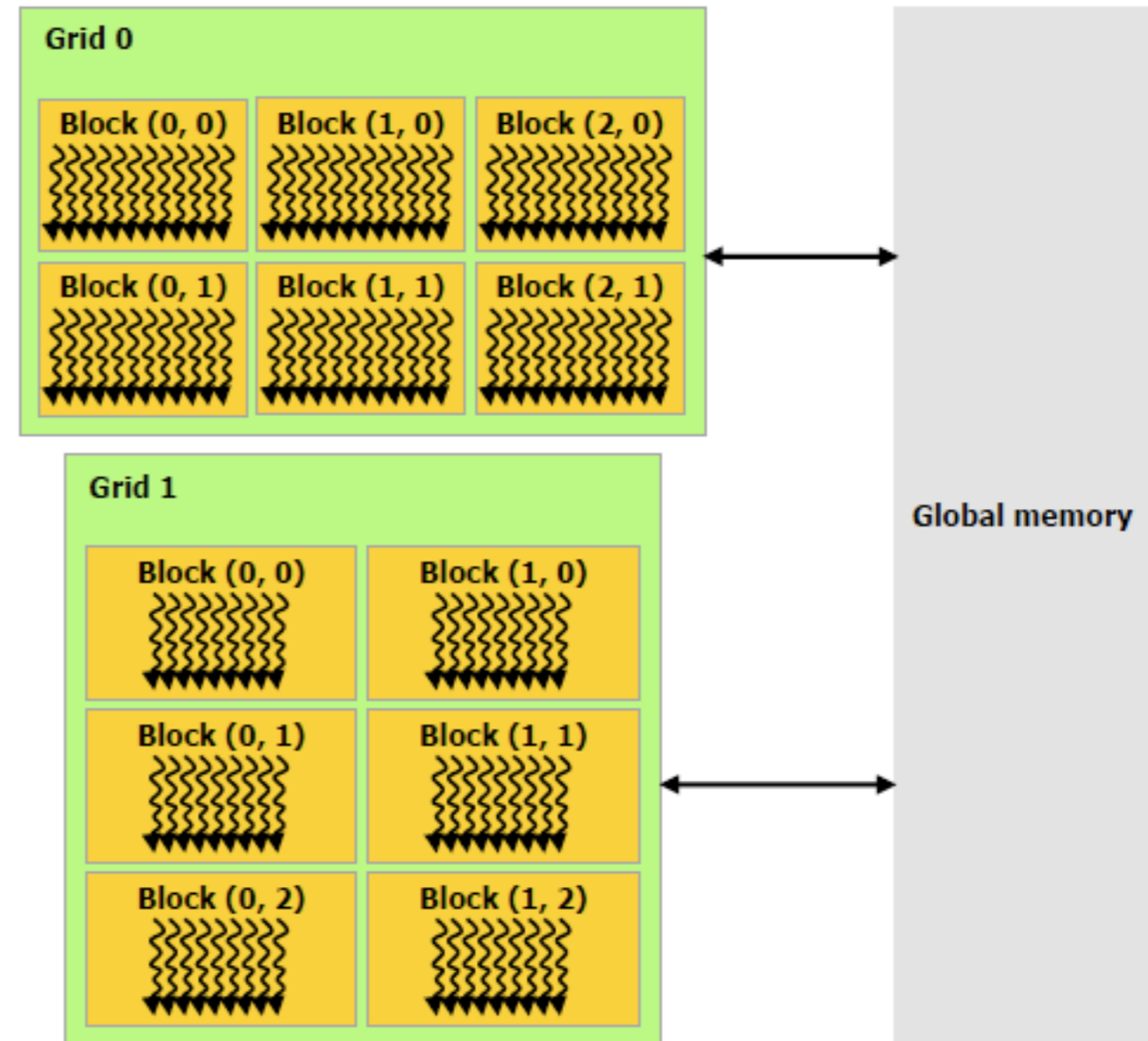
Very high memory bandwidth can be achieved using a hierarchy of memory

All threads have slower access to global memory

Each thread has private local memory



Each thread block has fast access to shared memory



All threads can also access read-only **Constant** and **Texture** memory, optimized for different memory usages

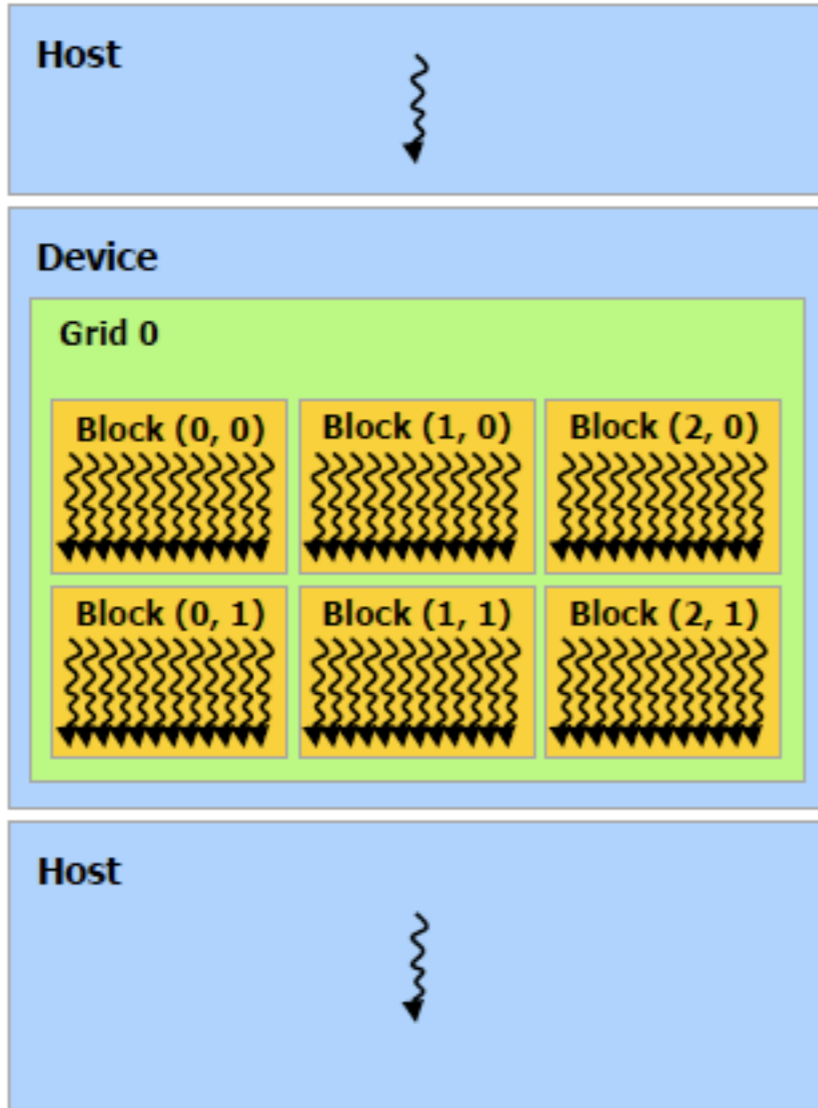
# General Flow of CUDA C Program

C Program  
Sequential  
Execution

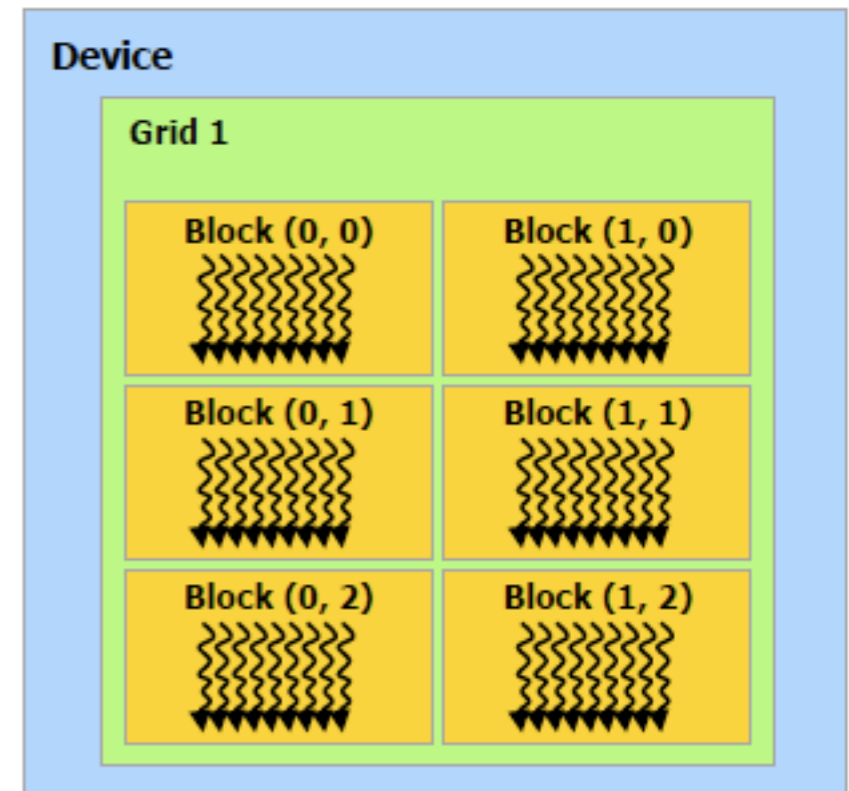
Serial code

Parallel kernel  
Kernel0<<<>>>()

Serial code



Parallel kernel  
Kernel1<<<>>>()



# Outline

- Concepts for GPU Computing
- Programming Model for GPU Computing using CUDA C
- **CUDA C Programming**
- Advanced CUDA Capabilities



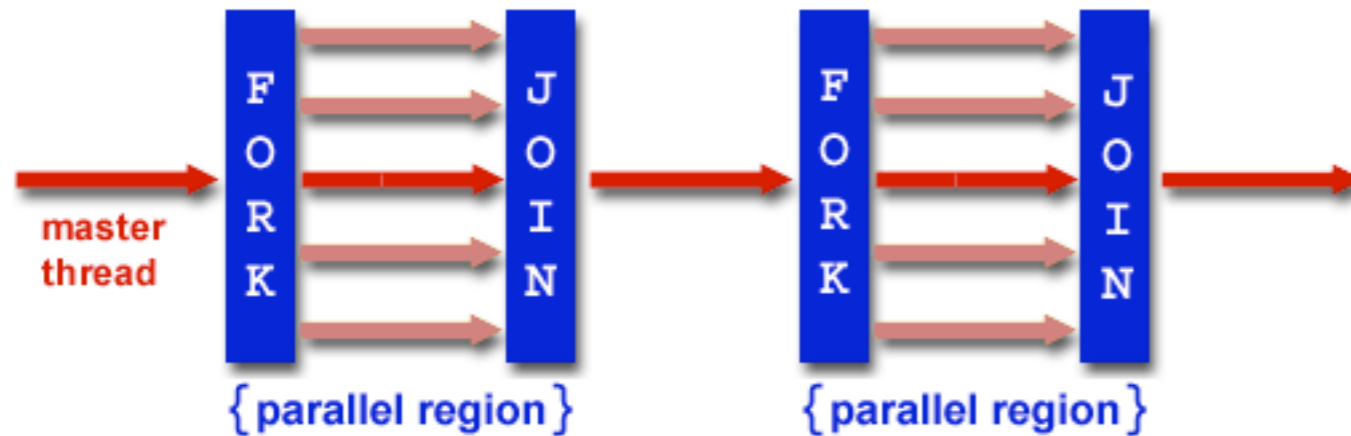
# Programming in CUDA C

- Comparison of Multithreading and GPU Computing
- The Kernel
  - Thread Hierarchy
- Memory Functions
- Examples

# Comparison to OpenMP Multithreading

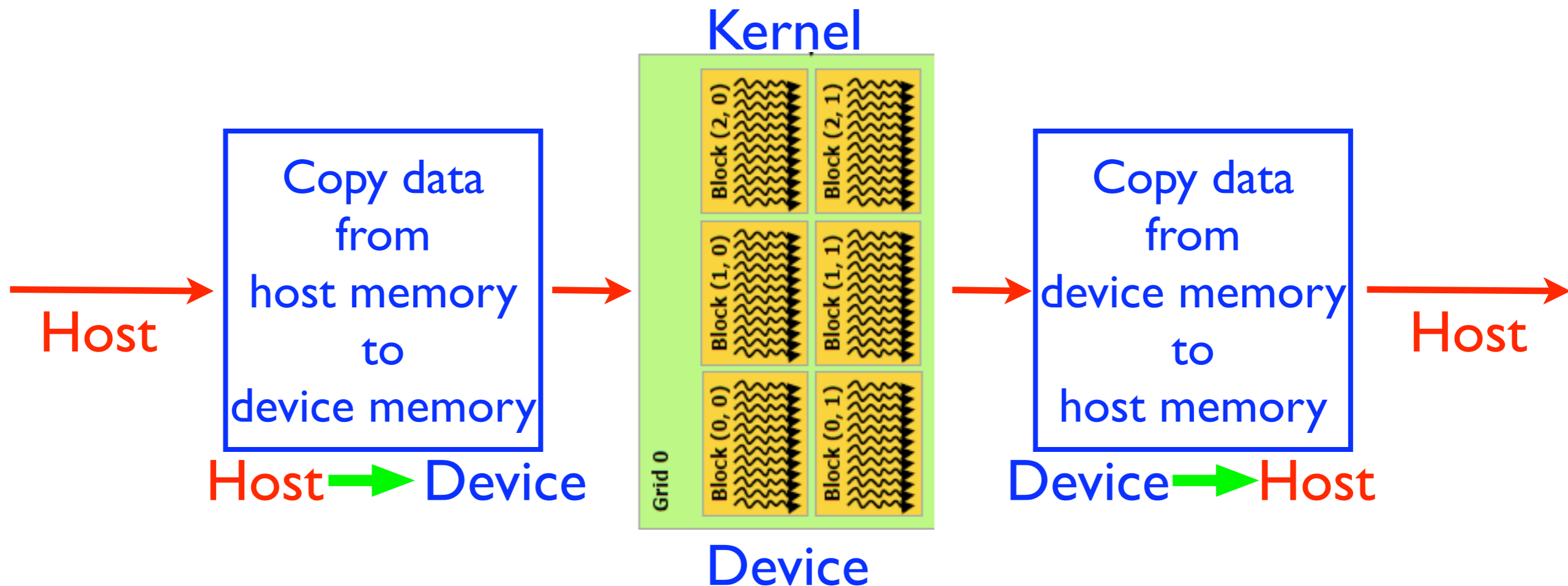
## General Implementation:

- Multithreading using OpenMP



All of this executes on the multicore host CPU with access to shared memory

- Parallel Computation on GPU device



# The Kernel in CUDA C

## Kernels:

- CUDA C enables the programmer to define C functions, called **kernels**, that are executed N times in parallel on the GPU device
- Declaration specifier: Kernels are defined using **\_\_global\_\_**

```
__global__ void VecAdd(int *a, int *b, int *c)
```

- Kernels are called from the host (CPU) using a new **execution configuration** syntax,

```
<<<blocksPerGrid, threadsperBlock>>>
```

```
VecAdd<<<blocksPerGrid, threadsperBlock>>> (d_a, d_b, d_c) ;
```

**NOTE:** Kernel functions are called from the **host**,  
but executed on the **device!**

# Blocks and Threads

## Thread Hierarchy:

- The execution configuration specifies:  
**blocksPerGrid**  
**threadsperBlock**
- These variables are 3-component integer vectors of type **dim3**
- For example,  
**dim3 threadsperBlock (N,N) ;**  
defines 2D thread blocks of size  $N$  by  $N$
- In the kernel function, the thread ID is accessed through the variable **threadIdx**, where the two dimensions are given by **threadIdx.x** and **threadIdx.y**

# Blocks and Threads

## Blocks:

- The block ID and block dimensions are similarly accessed through **blockIdx** giving **blockIdx.x** and **blockIdx.y**  
**blockDim** giving **blockDim.x** and **blockDim.y**
- A general formula for computing the appropriate index based on multiple blocks is

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

## Limitations:

- The maximum number of threads per block is the number of GPU cores, 512 for the GeForce GTX580.
- The number of blocks per grid is unlimited, and is determined by the number of blocks required to do the entire calculation.
- For a 2D computation of size  $N$  by  $N$

```
dim3 numBlocks (N/threadsPerBlock.x, N/threadsPerBlock.y) ;
```

# CUDA Kernel

## Standard serial C function

```
/* Function to compute y=a*x+y */
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i=0; i < n; i++)
        y[i]= a*x[i] + y[i];
}

/* Call Function from main() */
saxpy_serial(4096*256, 2.0, x, y);
```

## Parallel CUDA C kernel

```
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i=blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i]= a*x[i] + y[i];
}

/* Call Function from main() */
saxpy_parallel<<<4096,256>>>(4096*256, 2.0, x, y);
```

# CUDA Kernel

## General Comments:

- The kernel contains only the commands within the loop
- The kernel call is asynchronous
  - After the kernel is called, the host can continue processing before the GPU has completed the kernel computation

	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

- The computations in the kernel can only access data in device memory

Therefore, a critical part of CUDA programming is handling the transfer of data from host memory to device memory and back!

# CUDA Memory Functions

- Device memory is allocated and freed using  
`cudaMalloc()`  
`cudaFree()`

Example:

```
size_t size = N * sizeof(float);  
float* d_A;  
cudaMalloc(&d_A, size);
```

- Data is transferred using  
`cudaMemcpy()`

Example:

```
/* Allocate array in host memory */  
float* h_A = (float*)malloc(size);  
/* Copy array from host memory to device memory */  
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
```



# Example CUDA code for Vector Addition

```
#include<stdio.h>
#include<cuda.h>
#define N 100 /* Size of vectors */

/* Define CUDA kernel */
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x*blockDim.x+threadIdx.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

# Example CUDA code for Vector Addition

```
int main() {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    /* allocate the memory on the GPU */
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );

    /* Copy the arrays 'a' and 'b' from CPU host to GPU device*/
    cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice );

    int threadsPerBlock=512;
    int blocksPerGrid=(N+threadsPerBlock-1)/threadsPerBlock;
    add<<<blocksPerGrid, threadsPerBlock>>>( dev_a, dev_b, dev_c );

    /* Copy the array 'c' back from GPU device to CPU host*/
    cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost );

    /* Free the memory allocated on the GPU device*/
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
}
```

# Outline

- Concepts for GPU Computing
- Programming Model for GPU Computing using CUDA C
- CUDA C Programming
- **Advanced CUDA Capabilities**

# Advanced CUDA Capabilities

- Shared Memory
- Concurrent memory copy and kernel execution
- Asynchronous concurrent execution
- Lower-level CUDA driver API
- Multiple devices on host system with peer-to-peer memory access
- Texture and surface memory
- Graphics functions with OpenGL and Direct3D Application Programming Interfaces (APIs)