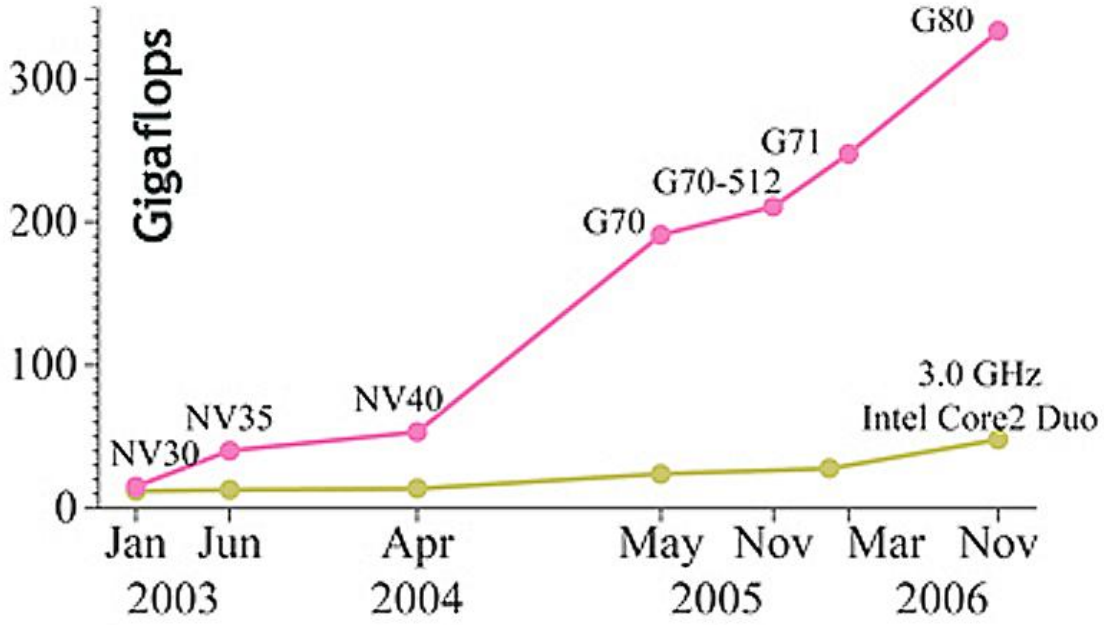
Lecture 11: Programming on GPUs (Part 1)

Overview

- GPGPU: General purpose computation using graphics processing units (GPUs) and graphics API
- GPU consists of multiprocessor element that run under the shared-memory threads model. GPUs can run hundreds or thousands of threads in parallel and has its own DRAM.
  - GPU is a dedicated, multithread, data parallel processor.
  - GPU is good at
    - Data-parallel processing: the same computation executed on many data elements in parallel
    - with high arithmetic intensity

- Performance history: GPUs are much faster than CPUs



AMD FireStream 9250: 1Tflops

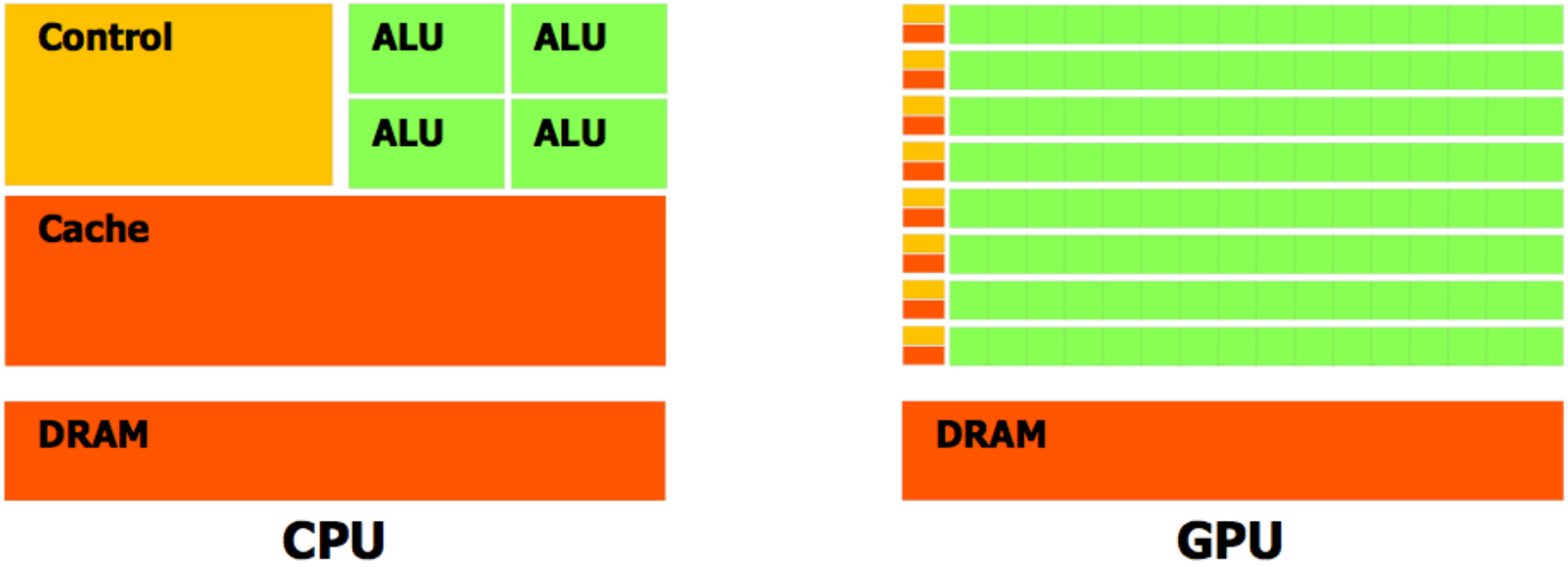
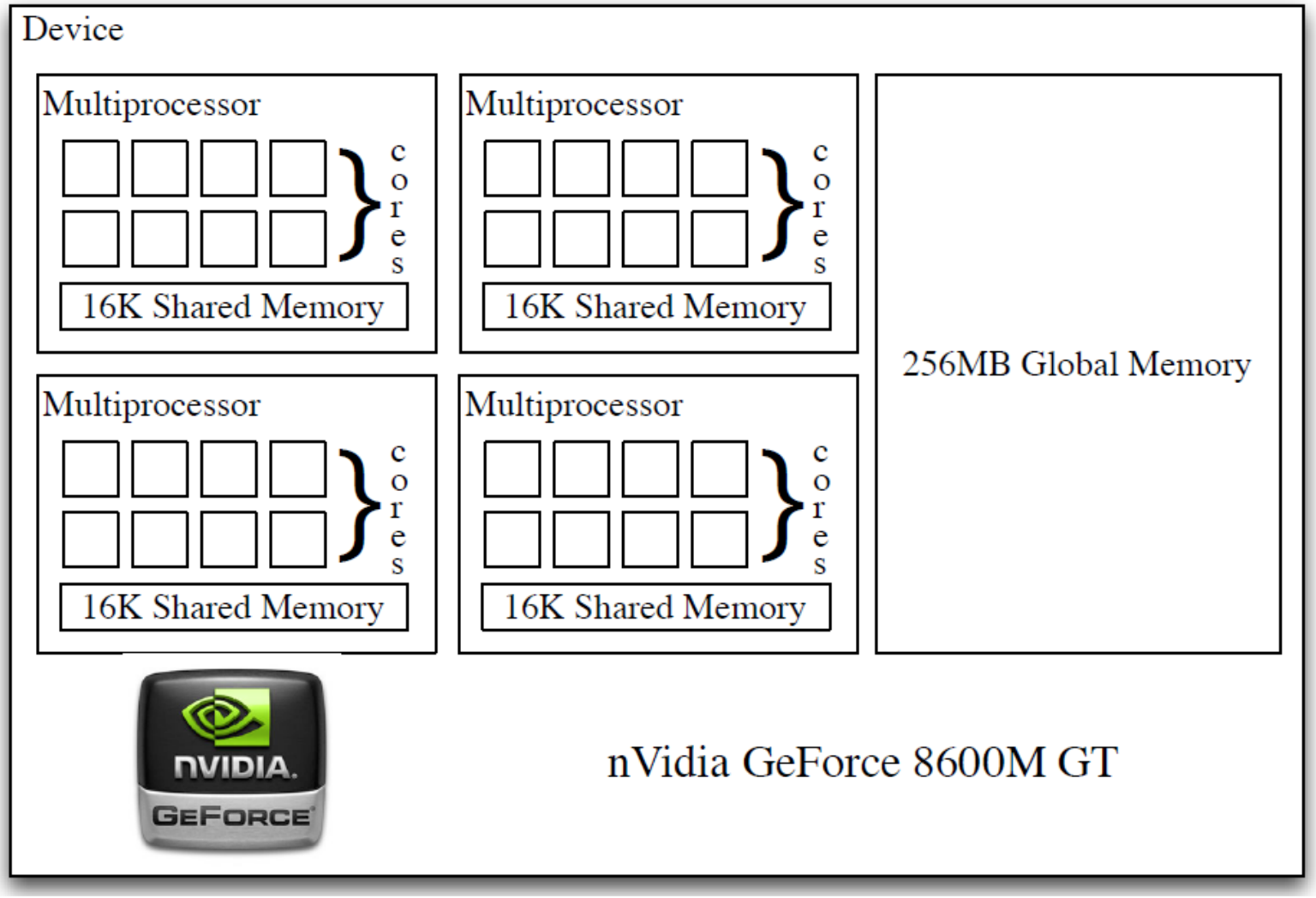
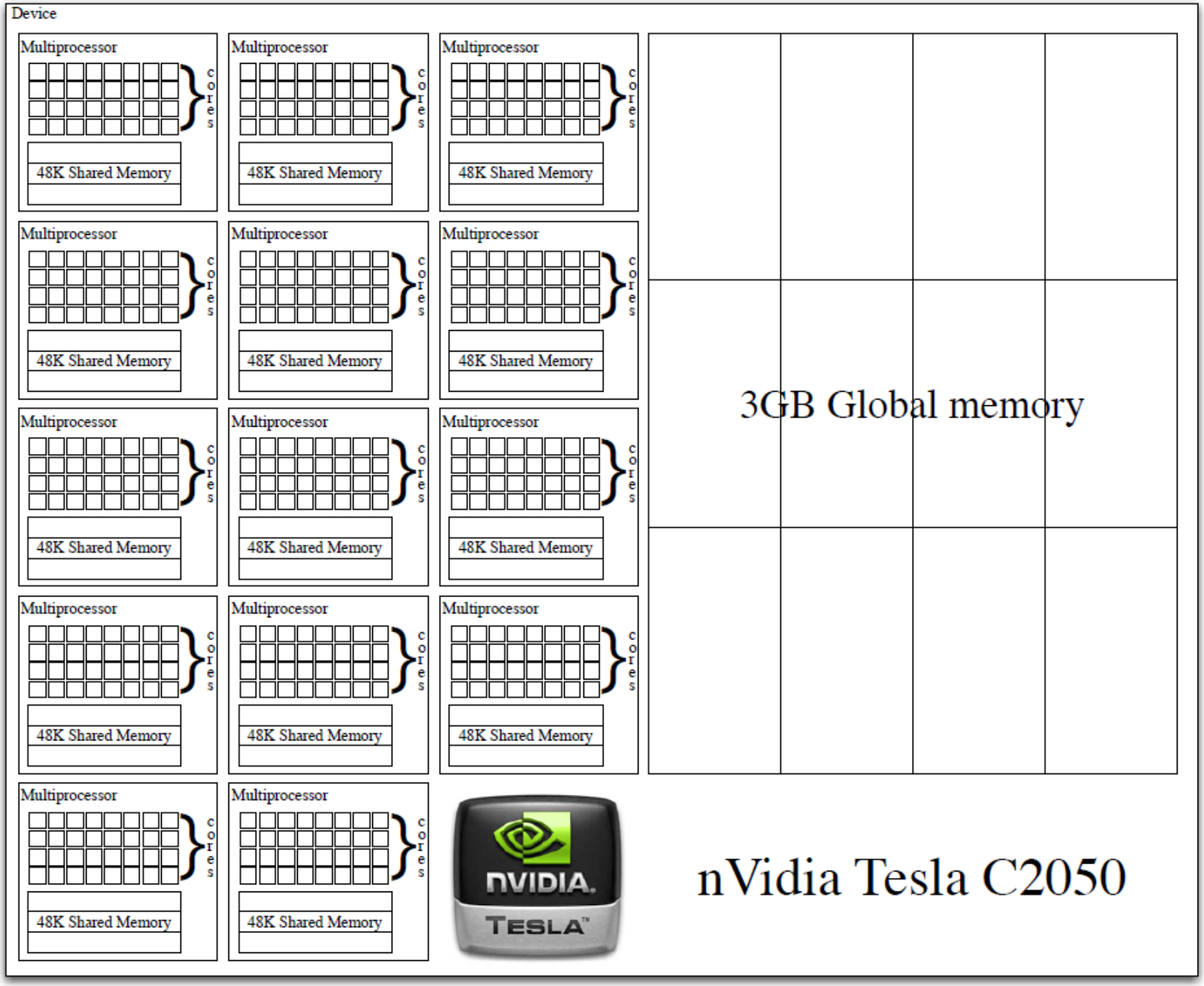


Figure 1-2. The GPU Devotes More Transistors to Data Processing

nVidia GPU Architecture



- Many processors are striped together
- Small, fast shared memory


Device
Multiprocessor
cores
48K Shared Memory
3GB Global memory
nVidia Tesla C2050

Hardware Overview

- Basic building block is a “streaming multiprocessor” (SM) with:
  - 32 cores, each with 1024 registers
  - up to 48 threads per core
  - 64KB of shared memory / L1 cache
  - 8KB cache for constants held in device memory
- C2050: 14 SMs, 3/6 GB memory

GPU Computing at CRC

- http://wiki.crc.nd.edu/wiki/index.php/Developmental_Systems
- gpu1.crc.nd.edu
- gpu2.crc.nd.edu
- gpu3.crc.nd.edu
- gpu4.crc.nd.edu
- gpu5.crc.nd.edu
- CUDA compiler is nvcc
- To compile and run GPU code:
  - module load cuda
  - module show cuda
  - nvcc hello.cu

CUDA Concepts and Terminology

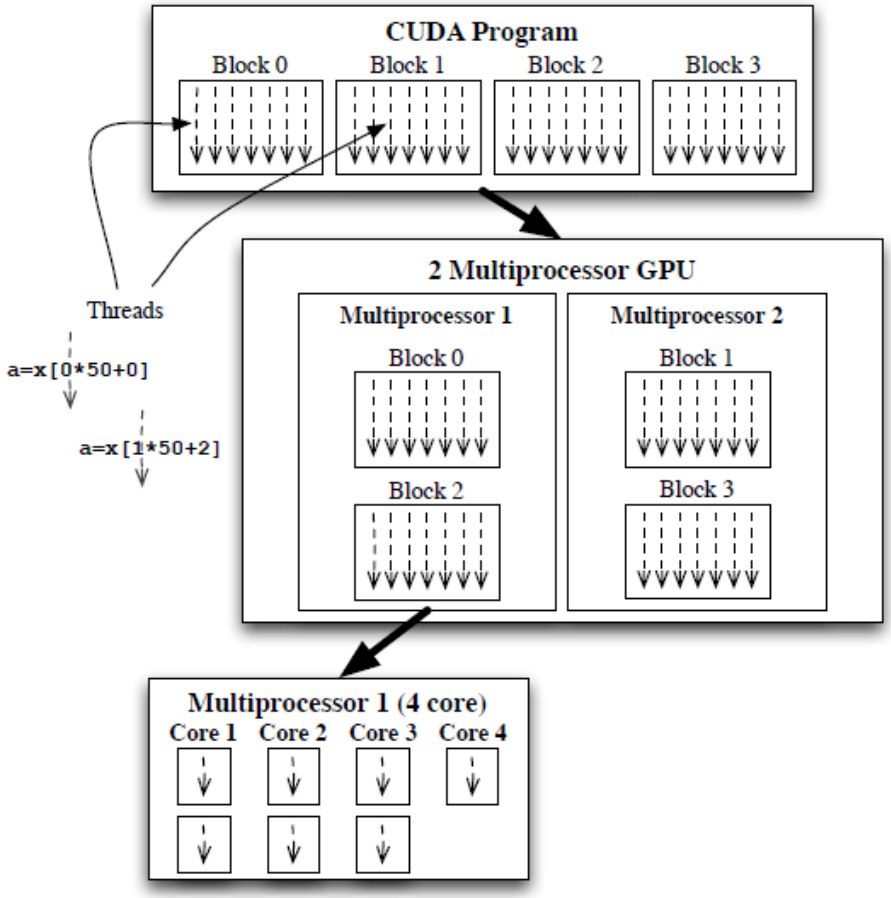
- Kernel: a C function which is flagged to be run on a GPU.
- A kernel is executed on the core of a multiprocessor inside a **thread**. Loosely speaking, a thread can be thought of as just an index $j \in N$, an index of cores in multiprocessors
- At any given time, a **block** of threads is executed on a multiprocessor. A lock can be thought of as just an index $i \in N$, an index of multiprocessors in devices
- Together, (i, j) corresponds to one kernel running on a core of a single multiprocessor.
- Simplistically speaking, parallelizing a problem is to split it into identical chunks indexed by a pair $(i, j) \in N \times N$

To parallelize a for loop:

for(int i=0; i < 1000; i++) {a=x[i];}

- In block/thread, we would like to have a single block/1000 thread ($i = 0, j = 0, \dots, 999$) kernels containing: a = x[thread_index];
- In real implementation, the exact same kernel is called blocks × threads times with the block and thread indices changing.
  - To use more than one multiprocessor, say $i = 0, .., 19, j = 0, \dots, 49$ and kernel:

  a = x[block_index+thread_index];



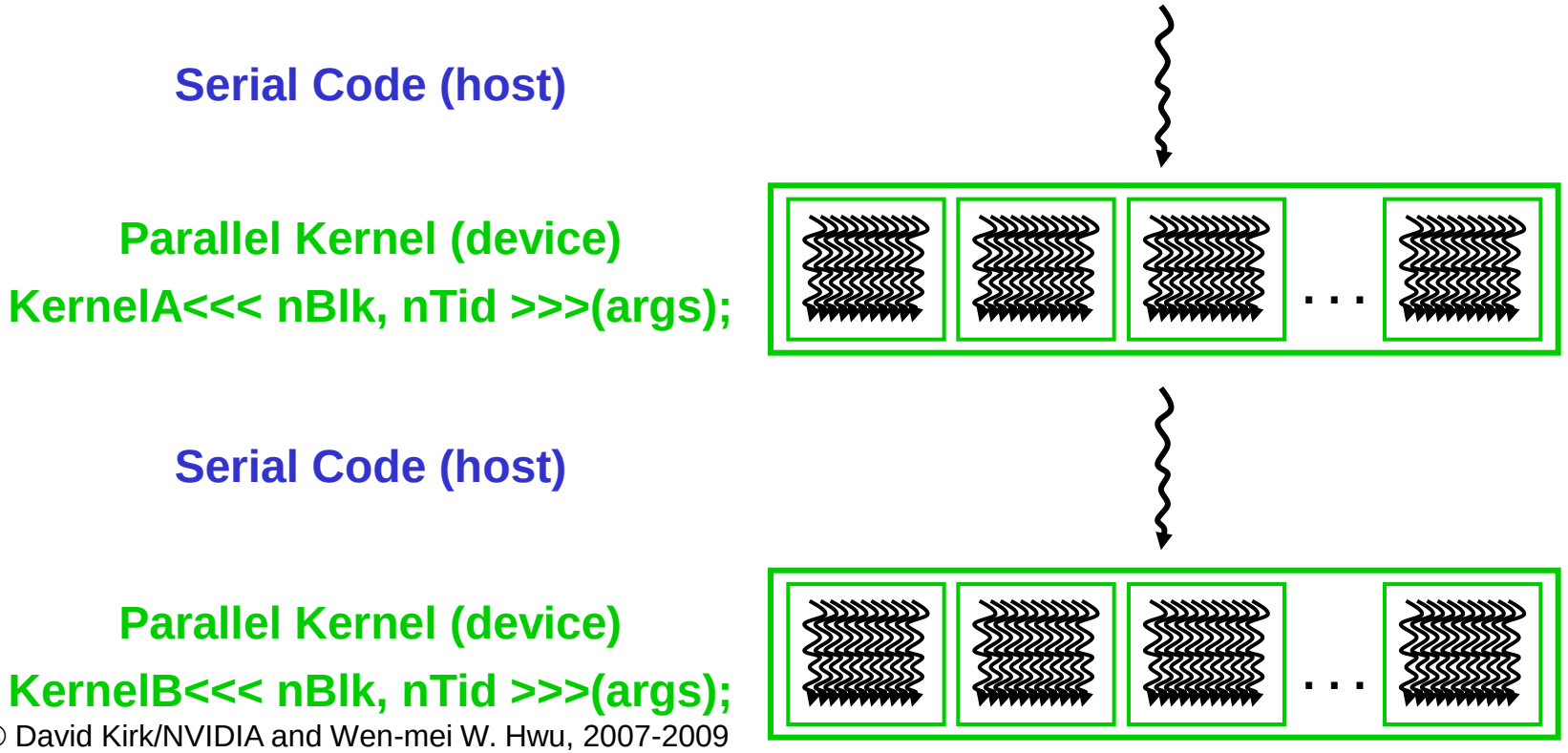
- We can not assume threads will complete in the order they are indexed.
- We can not assume blocks will complete in the order they are labeled.
- To deal with data/task dependency:
  - Use synchronization: __syncthreads();
  - Split into kernels and call consecutively from C
- Shared memory model: do not write to same memory location from different threads

- CUDA: Compute unified device architecture
  - A new hardware and software architecture for issuing and managing computations on the GPU
- CUDA C is a programming language developed by NVIDIA for programming on their GPUs. It is a extension of C.

CUDA Programming Model

- A CUDA program consists of code to be run on the **host**, i.e. the CPU, and the code to be run on the **device,** i.e. the GPU.
  - Device has its own DRAM
  - Device runs many threads in parallel
- A function that is called by the host to execute on the device is called a **kernel**.
  - Kernels run on many threads which realize data parallel portion of an application
- Threads in an application are grouped into **blocks**. The entirety of blocks is called the **grid** of that application.

- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SIMD kernel C code



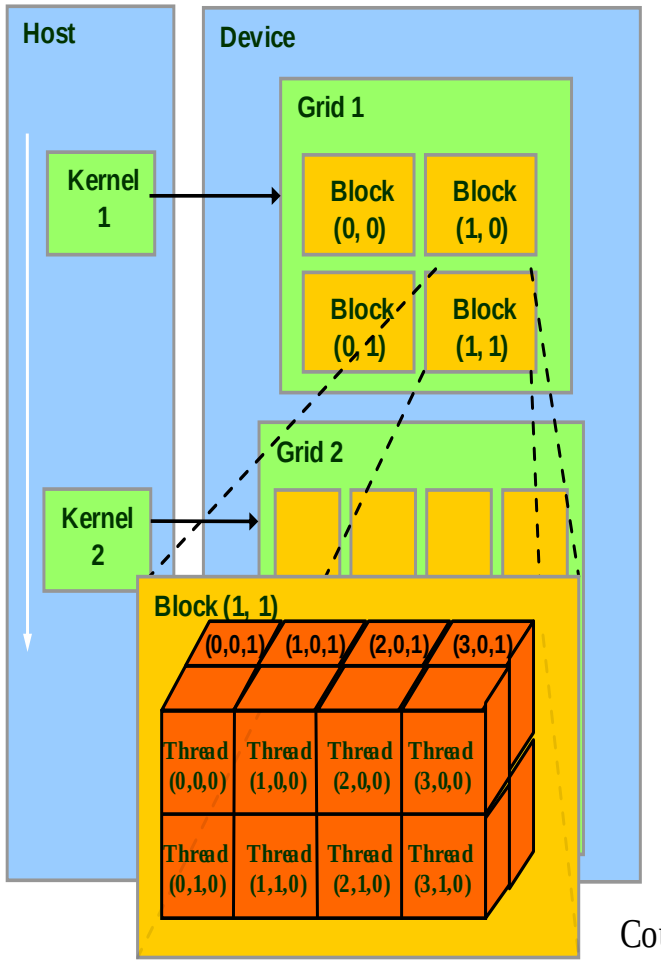
Extended C

- **Type Qualifiers**
  - **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);
```

Thread Batching

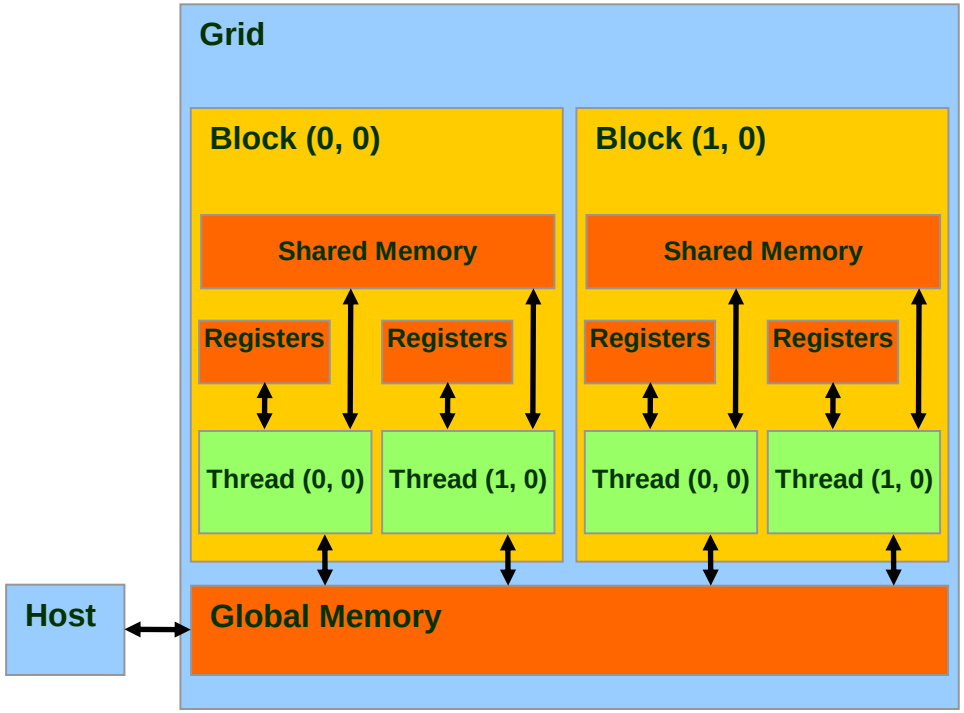
- A kernel is executed as a **grid of thread blocks**
- A **thread block** is a batch of threads that can cooperate.
- Each thread uses **ID** to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D or 3D
- Threads within a block coordinate by shared memory, atomic operations and barrier synchronization.
- Threads in different blocks can not cooperate.
- Convenient for solving PDEs on grid cells.



Courtesy: NDVIA

CUDA Memory Model

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access



Device Memory Allocation

- cudaMalloc()
  - Allocate space in device Global Memory
- cudaFree()
  - Free allocated space in device Global Memory
- Example. Allocate 64 by 64 single precision float array. Attached the allocated storage to *Md.

```
TILE_WIDTH = 64;
Float* Md
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);

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

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
- Asynchronous transfer
- 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, M, size, cudaMemcpyHostToDevice);

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

CUDA Function Declarations

- **__global__** defines a kernel function
  - Must return **void**
  - **Example: __global__ void KernelFunc()**
  - **Executed on the device, only callable from the host**
- **__device__** defines a function called by kernels.
  - Example: **__device__ float DeviceFunc()**
  - **Executed on the device, only callable from the device**
- **__host__** defines a function running on the host
  - **Example: __host__ float HostFunc()**
  - **Executed on the host, only callable from the host**

- `__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

Thread Creation

- Threads are created when program calls kernel functions.
- 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
   >>>(...);
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

Kernel Call – Hello World

```
// File name: hello.cu

#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <curand_kernel.h>


__global__ void kernel(void){
}

int main()
{
   kernel <<<1, 1>>> ();
   printf("Hello world\n");
   return 0;
}
```

Compile: nvcc hello.cu

```
// file name: add_num.cu
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <curand_kernel.h>

__global__ void add(int a, int b, int *c){
   *c = a + b;
}

int main()
{
   int c;
   int *dev_c;

   cudaMalloc((void**)&dev_c, sizeof(int));
   add <<<1, 1>>>(3, 7, dev_c);
   cudaMemcpy(&c, dev_c, sizeof(int), cudaMemcpyDeviceToHost);
   printf("result = %d\n", c);
   cudaFree(dev_c);
   return 0;
}
```

- Can pass parameters to a kernel as with C function
- Need to allocate memory to do anything useful on a device, such as return values to the host.

- Do not deference the pointer returned by cudaMalloc() from code that executes on the host. Host code may pass this pointer around, perform arithmetic on it. But we can not use it to read or write from memory.
- We can access memory on a device through calls to cudaMemcpy() from host code.

Querying Device

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include <string.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <curand_kernel.h>

int main(int argc, char** argv)
{
   int gpuDevice;
   int devNum = 0;
   int c, count;
   int cudareturn;

   cudaGetDeviceCount(&count);
   while ((c = getopt (argc, argv, "d:")) != -1)
   {
      switch (c)
      {
      case 'd':
         devNum = atoi(optarg);
      break;
      case '?':
         if (isprint (optopt))
            fprintf (stderr, "Unknown option `-%c'.\n", optopt);
         else
            fprintf (stderr,
               "Unknown option character `\\x%x'.\n",
               optopt);
         return 1;
      default:
         printf("GPU device not specified using device 0 ");
     }
   }
   cudareturn = cudaSetDevice( devNum );
   printf("device count = %d\n", count);
   if (cudareturn == 11)
   {
     printf("cudaSetDevice returned  11, invalid device number ");
     exit(-1);
   }
   cudaGetDevice(&gpuDevice);
   return 0;
}
```

GPU Vector Sums

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <curand_kernel.h>

#define N 50

__global__ void add(int *a, int *b, int *c){
    int tid = blockIdx.x;  // handle the data at this index

    if(tid < N)
        c[tid] = a[tid] + b[tid];
}

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

    cudaMalloc((void**)&dev_c, N*sizeof(int));
    cudaMalloc((void**)&dev_b, N*sizeof(int));
    cudaMalloc((void**)&dev_a, N*sizeof(int));
    for(i=0; i < N; i++)
    {
        a[i] = -i;
        b[i] = i*i*i;
    }
    cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);

    add <<<N, 1>>>(dev_a, dev_b, dev_c);
    cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);
    for(i=0; i < N; i++)
        printf("%d + %d = %d\n", a[i], b[i], c[i]);

    cudaFree(dev_c);
    cudaFree(dev_b);
    cudaFree(dev_a);
    return 0;
}
```

- CUDA built-in variable: blockIdx
  - CUDA runtime defines this variable.
  - It contains the value of the block index for whichever block is currently running the device code.
  - CUDA C allows to define a group of blocks in two-dimensions.
- N – specified as the number of parallel blocks
  - A collection of parallel blocks is called a **grid**.
  - This example specifies to the runtime system to allocate a one-dimensional grid of N blocks.
  - Threads will have different values for blockIdx.x, from 0 to $N - 1$.
  - $N \leq 65{,}535$ – a hardware-imposed limit.
- if(tid< N)
  - Avoid potential bugs – what if # threads requested is greater than N?