


"Introduction CUDA programming"

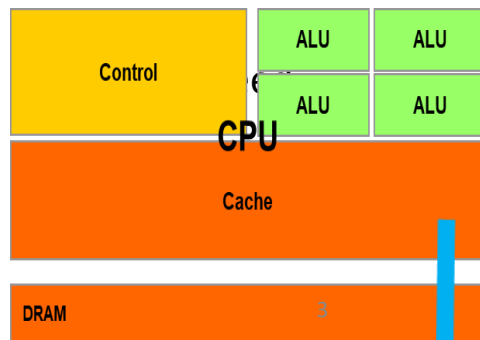
Dr.Bheemappa H
Assistant professor, IIIT,Sricity
bheemappa.h@iiit.in



- ✓ The Age of Parallel Processing
- ✓ The Rise of GPU Computing,
- ✓ History of GPUs, Early GPU computing,
- ✓ CUDA
 - ✓ A First Program
 - ✓ Querying Devices
 - ✓ Using Device Properties
 - ✓ RUNNING IN PARALLEL
- ✓ Applications of CUDA

The Age of Parallel Processing

- How to increase the performance of system?
- CPUs .1 central Processing units :
 - 1MHz to 1GHz -> 4GHz, (About 30 years)
 - clock speeds between nearly 1,000 times faster than the clock on the original personal computer
 - Upward-spiraling processor clock speeds - > power and heat restrictions
 - A single processing core to multicore processor
 - Personal computers could continue to improve in performance.
 - Without the need for continuing increases in processor clock speed.
 - 3,4,6,8,2- and 16-core CPUs
 - **hyperthreading** with two hardware threads, designed to maximize the of sequential programs
 -



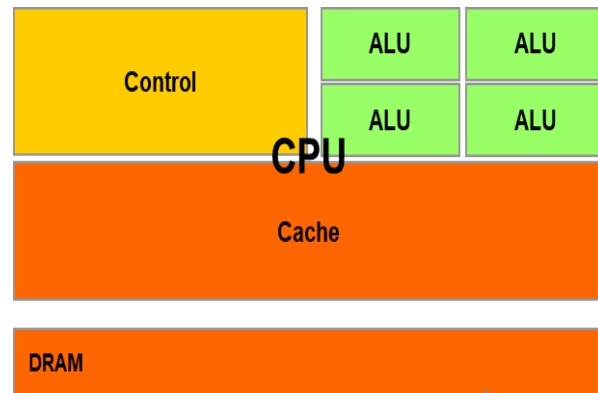
How do we use all cores of CPU for execution of applications?

Introduction

Performance?

Two Aspects:

- Data Access Rate Capability -> Bandwidth
- Data Processing Capability ->
 - How many ops per sec : FLOPS (Floating-Point Operations per Second)
- $3.4\text{Ghz} \times 8 \text{ FOPS/cycle} = 27 \text{ GFLOPS}$




Rise of GPU Computing

- Popularity of graphical OS in late 80s created a market for a new compute device
 - 2D display accelerators offered hardware-assisted bitmap operations
- Silicon Graphics popularized use of 3D graphics
 - Released OpenGL as a programming interface to its hardware
 - Popularity of first-person games in mid-90s was the final push
 - [first-person shooter](#)
- Pixel shaders were used to produce a color for a pixel on screen
 - It uses the (x,y) coordinates, input colors, texture coordinates and other attributes as inputs

Rise of GPU Computing

- NVIDIA's GeForce 3 series in 2001 implemented the DirectX 8.0 standard from Microsoft
 - Transform and lighting computations could be performed directly on the graphics processor,
 - Enhancing the potential for even more visually interesting applications
- DirectX basically helps your PC create the needed visual and audio effects needed in the games.

Early GPU Computing

- GPUs
 - OpenGL and DirectX used to interact with a GPU and Performing arbitrary computations on a GPU would still be subject to the **constraints of programming within a graphics API.**
 - Programmable pipelines
 - General-purpose computation through graphics APIs
 - Early 2000
 - Designed to produce a color for every pixel on the screen using programmable arithmetic units (Pixel shaders)
 - Arithmetic being performed on the input colors –
- Programmers could then program the pixel shaders to perform arbitrary computations on this data.
- High arithmetic throughput of GPUs,
 - Initial results from these experiments promised a **bright future for GPU computing**

Early GPU Computing

- GPUs to perform general-purpose computations.
 - How to write algorithms requiring the ability to write to arbitrary locations in memory (scatter) could not run on a GPU
 - Researchers tricked GPUs to perform non-rendering computations
 - If you wanted to use a GPU to perform general-purpose computations.
 - **Need to learn OpenGL or DirectX**
 - Programming model was very restrictive
 - Limited input colors and texture units, writes to arbitrary locations, floatingpoint computations
- This spurred the need for a highly-parallel computational device with high computational power and memory bandwidth
 - CPUs are more complex devices catering to a wider audience

PARALLEL PROGRAMMING LANGUAGES AND MODELS

- There are a variety of APIs that can be used to perform general purpose calculations on the GPU
 - Message Passing Interface (MPI) for scalable cluster computing and **OpenMPI** for shared-memory multiprocessor systems.
 - high-performance scientific computing domain.
 - Applications written in MPI have been known to run successfully on cluster computing systems
 - **CUDA**, on the other hand, provides shared memory for parallel execution in the GPU to address this difficulty.
 - CUDA achieves much higher scalability with simple, low-overhead thread management and no cache coherence hardware requirements.
 - **OpenCL** programming model defines language extensions and runtime APIs to allow programmers to manage parallelism and data delivery in massively parallel processors

- many applications use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs.
- This is why the CUDA programming model, introduced by NVIDIA in 2007, is designed to support joint CPUGPU execution of an application.¹ The demand for supporting joint CPUGPU execution is further reflected in more recent programming models such as OpenCL

Heterogeneous Parallel Computing

- A large peak-performance gap between many-threads GPUs and general-purpose multicore CPUs. (Why)
 - CPU is Latency Oriented Design
 - High clock frequency, Large caches, Sophisticated control, Powerful ALU
 - GPUs: Throughput Oriented Design
 - Small caches, To boost memory throughput, Simple control (not Branch control)
 - Energy efficient ALU, Require massive number of threads to tolerate latencies.
 - Fundamental design philosophies between the two types of processors.
 - Memory bandwidth (GPU operating at approximately six times the memory bandwidth)

GPU shaped by the fast-growing video game (massive number of floating-point calculations per video frame)

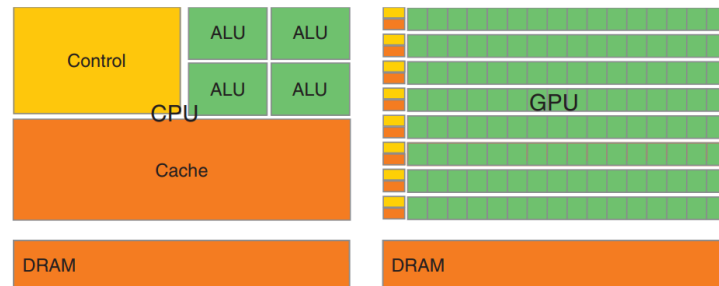


FIGURE 1.1

CPUs and GPUs have fundamentally different design philosophies.

- CPUs for sequential parts where latency matters
 - CPUs can be 10+X faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
 - GPUs can be 10+X faster than CPUs for parallel code

GPU Architecture

- GPUs consist of Streaming Multiprocessors (SMs)
- SMs contain Streaming Processors (SPs) or Processing Elements (PEs)
 - Each SM in has a number of streaming processors (SPs) that share control logic and instruction cache.
 - Each core contains one or more ALUs and FPUs
- Two SMs form a building block;

Intel CPUs

support 2 or 4 threads, depending on the machine model, er core.

The G80 chip

supports up to 768 threads per SM, which sums up to about 12,000 threads for this chip.

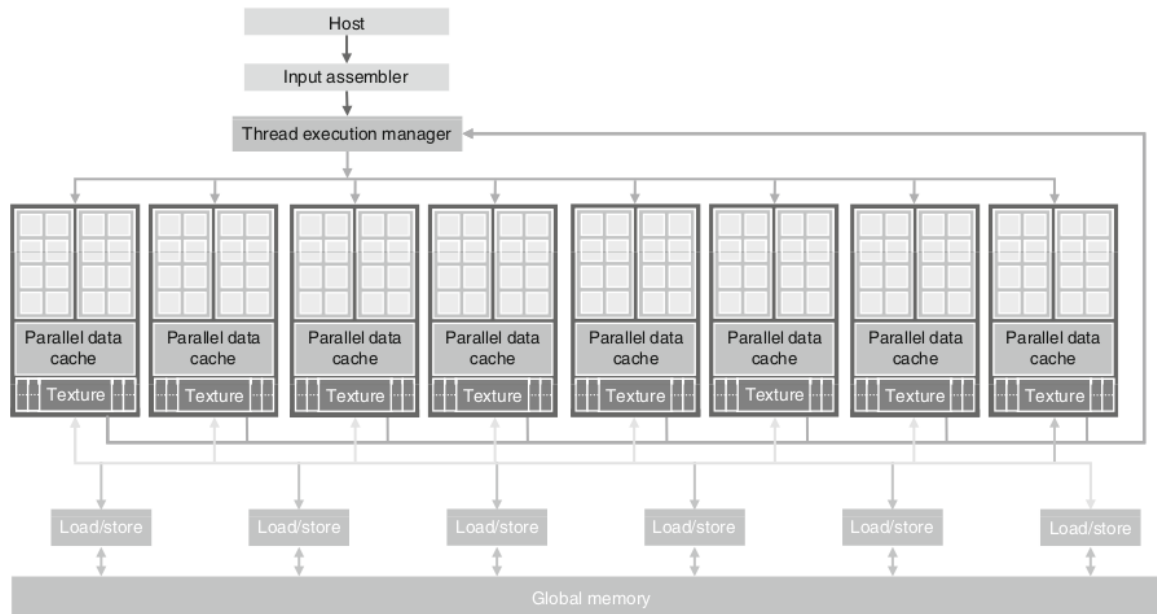
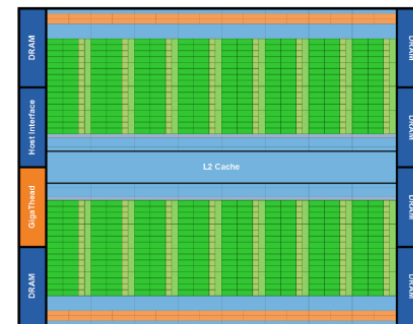


FIGURE 1.3

Architecture of a CUDA-capable GPU.

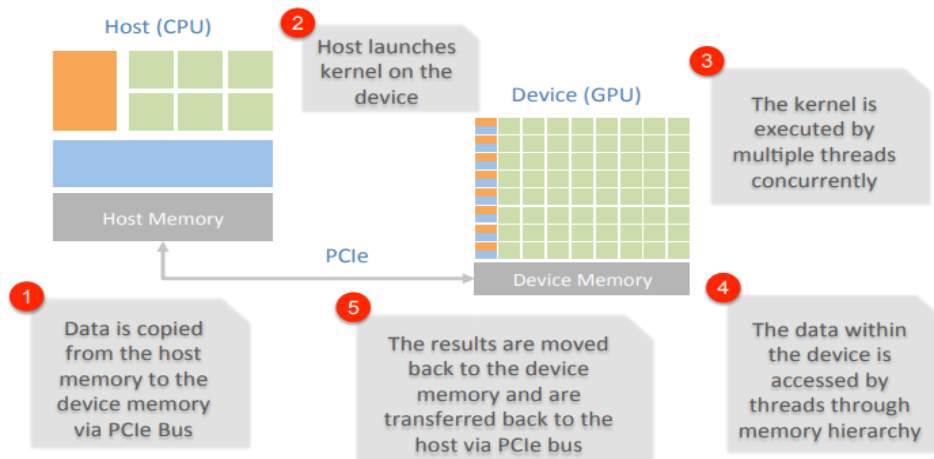


Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatcher), a green portion (execution units), and light blue portions (register file and L1 cache).

GPU Architecture

- Compute Capability
 - Architecture features
 - Number of registers and cores, cache and memory size, supported arithmetic instructions
 - Programmer should be aware of the differences among different versions of hardware
- CPU is responsible for initiating computation on the GPU and transferring data to and from the GPU

Dataflow



CPU vs GPU

CPU Limitations

CPU

- Designed for running a small number of potentially complex tasks
- Suitable to run system software like the OS and applications
- CPUs have large and broad instruction sets
- Explicit thread management

GPU

- Designed for running large number of simple tasks
- Suitable for data-parallelism
 - GPUs are best suited for repetitive and highly-parallel computing tasks
- GPU has a few highly optimized instruction sets
- Threads are managed by hardware

Limitations : Heavyweight Instruction Sets, Context Switch Latency || Less Powerful Cores, Limited APIs

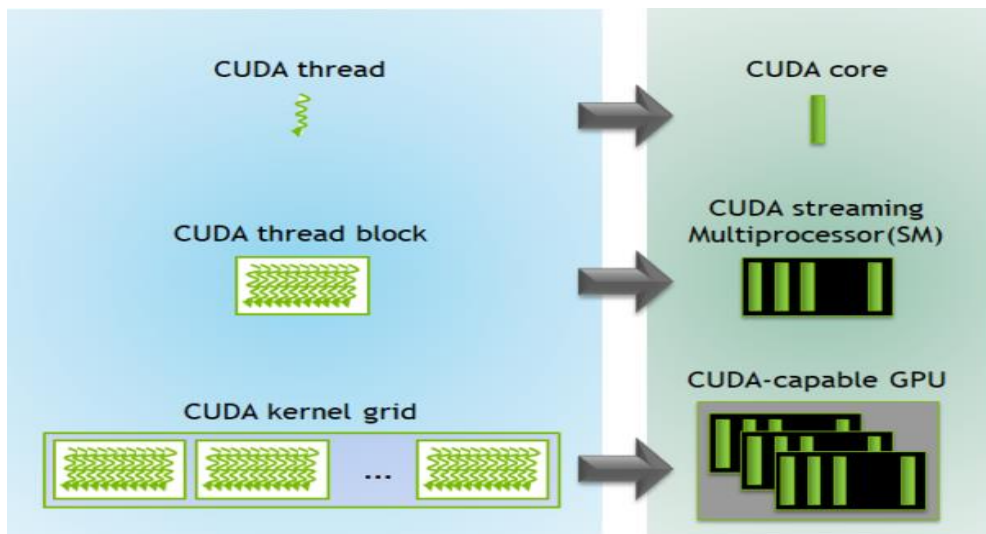
CUDA Programming

What is CUDA?

- General purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs
- Massively parallel, Compute Intensive.
- CUDA Architecture included
 - A **unified shader pipeline**, allowing each and every **arithmetic logic unit (ALU) on the chip to be marshaled** by a program intending to perform general-purpose computations.
 - **single-precision floating-point arithmetic**
- Key abstractions in CUDA
 - Hierarchy of thread groups
 - Shared memories
 - Barrier synchronization
- CUDA ->Single Instruction Multiple Thread
 - Computations can be divided into hundreds or thousands of independent units of work.

What is CUDA?

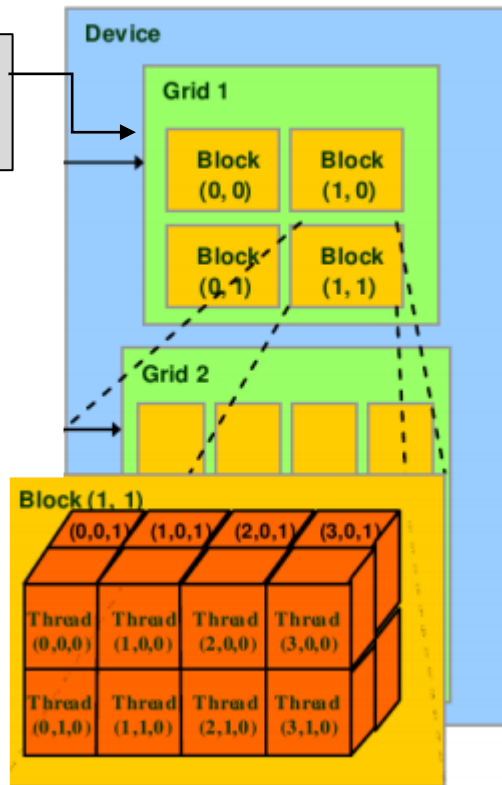
- Partition the problem => set of Sub-problems that can be solved independently(coarse)
- Assign each sub-problem => a “block” of threads to be solved in parallel
- Each sub-problem is also decomposed into finer work items to solved in parallel by all threads within the “block” .



What is CUDA?

- Threads, Blocks and Grids ? ?
 - Logical partitioning with:
 - Threads
 - Thread IDs
 - Blocks of threads
 - Block IDs
 - Block Dimensions
 - Threads are arranged in 1D, 2D, 3D logical fashion
 - Grid of blocks
 - Grid Dimensions
 - Blocks are arranged in 1D, 2D, 3D logical fashion
 - Each is limited with physical resources available
 - All threads follow SPMD (SPMD (single program, multiple data))
 - Same code runs on all threads with different data

• Each SM is independent and bound by threads and blocks count



What is WARP

Executes threads of one warpsize at time

WARP is 32 (consists of 32 threads)

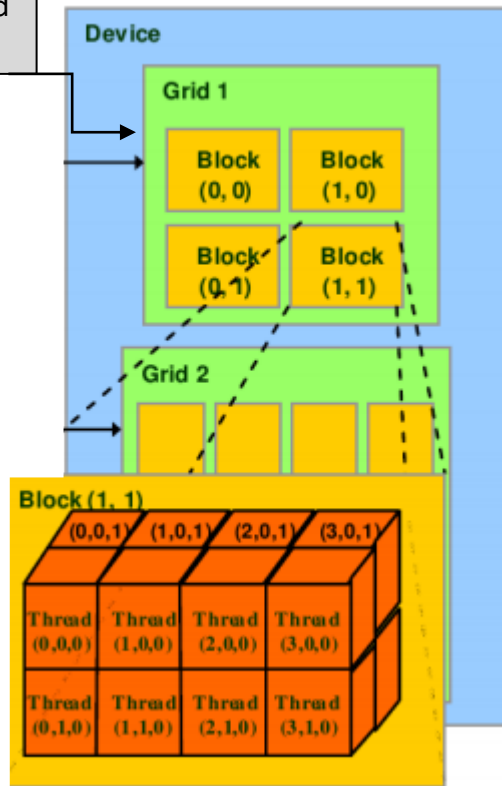
SIMD architecture to manage and execute threads in groups of 32 called warps.

GPU	GeForce GTX 680 (Kepler GK104)	GeForce GTX 980 (Maxwell GM204)
CUDA Cores	1536	2048
Base Clock	1006 MHz	1126 MHz
GPU Boost Clock	1058 MHz	1216 MHz
GFLOPs	3090	4612
Compute Capability	3.0	5.2
SIMDs	8	16
Shared Memory / SM	48KB	96KB
Register File Size / SM	256KB	256KB
Active Blocks / SM	16	32
Texture Units	128	128
Texel fill-rate	128.8 Gigatexels/s	144.1 Gigatexels/s
Memory	2048MB	4096MB
Memory Clock	6008 MHz	7010 MHz
Memory Bandwidth	192.3 GB/sec	224.3 GB/sec
ROPs	32	64
L2 Cache Size	512KB	2048KB
TDP	195 Watts	165 Watts
Transistors	3.54 billion	5.2 billion

CUDA Compilation and PTX

- Any source file containing CUDA language extensions must be compiled with **NVCC**
- Code sent from CPU to GPU is in Parallel Thread Execution (PTX)

• Each SM is independent and bound by threads and blocks count



CUDA PROGRAM STRUCTURE

- A CUDA program consists of one or more phases that are executed on either the host (CPU) or a device such as a GPU.
- The phases that exhibit little or no data parallelism are implemented in host code.
- The phases that exhibit rich amount of data parallelism are implemented in the device code.
- A CUDA program is a unified source code encompassing both host and device code.

The NVIDIA C compiler (nvcc) separates the two during the compilation process.

1. The host code is straight ANSI C code; it is further compiled with the host's standard C compilers and runs as an ordinary CPU process.
 2. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called **kernels**
 3. The device code is typically further compiled by the nvcc and executed on a GPU device
- The **kernel functions** (or, simply, kernels) typically generate a large number of threads to exploit data parallelism.
 - **Ex:** Entire matrix multiplication computation can be implemented as a kernel where each thread is used to compute one element of output matrix

- Number of threads used by the kernel is a function of the matrix dimension
- For a 1000 x 1000 matrix multiplication, the kernel that uses one thread to compute one P element would generate 1,000,000 threads when it is invoked.
- **CUDA threads are of much lighter weight than the CPU threads**
 - CUDA requires lower cycles to generate and schedule compared to CPU.

C

```
void c_hello(){  
    printf("Hello World!\n");  
}  
  
int main() {  
    c_hello();  
    return 0;  
}
```

CUDA

```
__global__ void cuda_hello(){  
    printf("Hello World from GPU!\n");  
}  
  
int main() {  
    cuda_hello<<<1,1>>>();  
    return 0;  
}
```

`__global__` : specifies
r indicates a function
that runs on device
(GPU)

host =>cpu

The major difference between C and CUDA implementation is
`__global__` specifier and `<<<...>>>` syntax.

function can be called through host code, e.g. the `main()` function in the example

Compiling CUDA programs=> **nvcc** hello.cu -o hello

ert Design Transitions Animations Slide Show Review V

bheemappah@iiits001gpu: ~

```
#include<stdio.h>

__global__ void cuda_hello(){

}

int main() {
    cuda_hello<<<1,1>>>();
    printf("Hello world");
    return 0;
}

~
~
~
~
~
~
~
~
~
~
```

```
bheemappah@iiits001gpu:~$ vim hello.cu
bheemappah@iiits001gpu:~$ nvcc hello.cu -o hello
bheemappah@iiits001gpu:~$ ./hello
Hello worldbheemappah@iiits001gpu:~$
```

```
#include <iostream>

__global__ void kernel( void ) {
}

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

- A call to the empty function, embellished with <<<1,1>>>
- CUDA C adds the __global__ qualifier to standard C.
- Passing parameters

RUNNING IN PARALLEL

- Vector Addition on the Device

The syntax of kernel execution configuration is as follows

```
<<< M , T >>>
```

Going parallel

```
add<<< 1, 1 >>>();  
      ↓  
add<<< N, 1 >>>();
```

Instead of executing add() once, execute N times in parallel

kernel execution configuration <<<...>>>

Informs how many threads to launch on GPU during CUDA runtime .

Passing parameters,

- We can pass parameters to a kernel as we would with
- any C function
 - We need to allocate memory to do anything useful on a device, such as return values to the host.

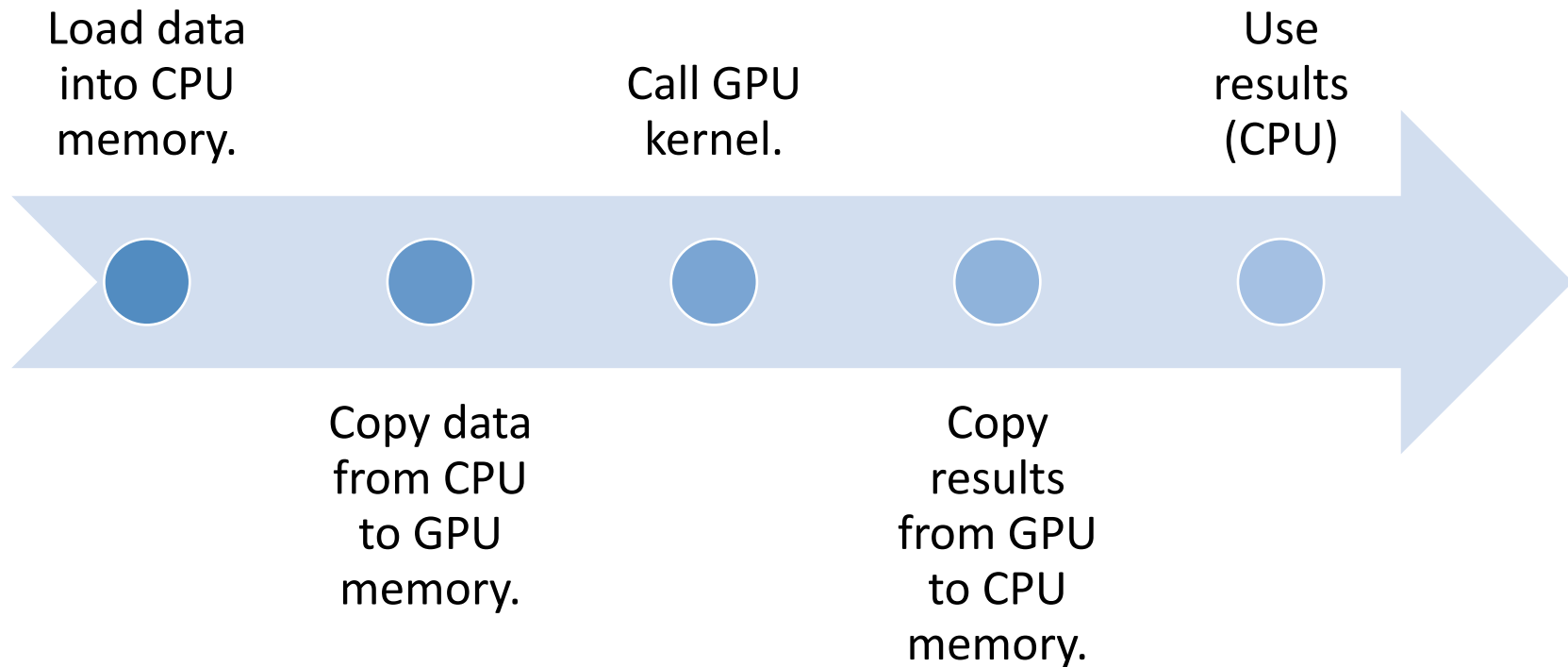
```
#include <stdio.h>
#include <cuda.h>
const char *msg = "Hello World.\n";
__global__ void dkernel() {
    printf(msg);
}
int main() {
    dkernel<<<1, 32>>>();
}
```



error: identifier "msg" is undefined in device code

- A variable in CPU memory cannot be accessed directly in a GPU kernel
- A programmer needs to maintain copies of variables.
- It is programmer's responsibility to maintain sync.

CUDA Program Flow



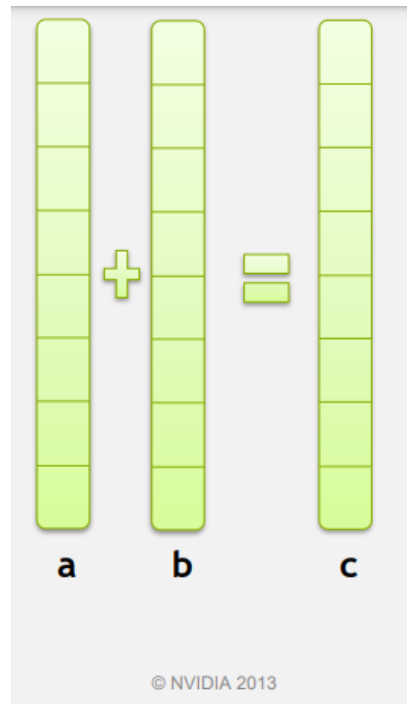
Addition on the Device

- A simple kernel to add two integers

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

add() will execute on the device
add() will be called from the host

- **a, b and c** must point to device memory as add() runs on the device
- **How do we allocate device memory ?**
- Memory Management ?
 - Device pointers
 - Host pointers



Memory Management

- `cudaMalloc()`,
- `cudaFree()`,
- `cudaMemcpy()`

- Device pointers point to GPU memory
May be passed to/from host code
May not be dereferenced in host code
- Host pointers point to CPU memory
May be passed to/from device code
May not be dereferenced in device code

Restrictions on the usage of device pointer as follows:

- You *can* pass pointers allocated with `cudaMalloc()` to functions that execute on the device.
- You *can* use pointers allocated with `cudaMalloc()` to read or write memory from code that executes on the device.
- You *can* pass pointers allocated with `cudaMalloc()` to functions that execute on the host.
- You *cannot* use pointers allocated with `cudaMalloc()` to read or write memory from code that executes on the host.

cudaMemcpy()

- Host pointers can access memory from host code,
- Device pointers can access memory from device code.
- How to access the memory on device from host code?
 - Using cudaMemcpy()
 - An additional parameter to specify which of the source and destination pointers point to device memory.
 - cudaMemcpyDeviceToHost: Source pointer is a device pointer and the destination pointer is a host pointer.
 - cudaMemcpyHostToDevice: source data is on the host and the destination is an address on the device.

Addition on the Device: main()

```
#include <iostream>
#include "book.h"

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

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

    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) ) );

    add<<<1,1>>>( 2, 7, dev_c );

    HANDLE_ERROR( cudaMemcpy( &c,
                              dev_c,
                              sizeof(int),
                              cudaMemcpyDeviceToHost ) );

    printf( "2 + 7 = %d\n", c );
    cudaFree( dev_c );

    return 0;
}
```

host copy of C

???

memory on the device
Allocate space for device copies

Copy result back to host

Querying Devices

- How to determine which devices (if any) are present and what are capabilities each device?

1. To get the count of CUDA devices

```
cudaGetDeviceCount()
```

```
int count;  
HANDLE_ERROR( cudaGetDeviceCount( &count ) );
```

2. cudaDeviceProp structure

```
struct cudaDeviceProp {  
    char name[256];  
    size_t totalGlobalMem;  
    size_t sharedMemPerBlock;  
    int regsPerBlock;  
    int warpSize;  
    size_t memPitch;  
    int maxThreadsPerBlock;  
    int maxThreadsDim[3];  
    int maxGridSize[3];  
    size_t totalConstMem;  
    int major;  
  
    int minor;  
    int clockRate;  
    size_t textureAlignment;  
    int deviceOverlap;  
    int multiProcessorCount;  
    int kernelExecTimeoutEnabled;  
    int integrated;  
    int canMapHostMemory;  
    int computeMode;  
    int maxTexture1D;  
    int maxTexture2D[2];  
    int maxTexture3D[3];  
    int maxTexture2DArray[3];  
    int concurrentKernels;  
}
```

Querying Devices

```
#include "../common/book.h"
```

```
int main( void ) {
```

```
    cudaDeviceProp prop;
```

```
    int count;
```

```
    HANDLE_ERROR( cudaGetDeviceCount( &count ) );
```

```
    for (int i=0; i< count; i++) {
```

```
        HANDLE_ERROR( cudaGetDeviceProperties( &prop, i ) );
```

```
        //Do something with our device's properties
```

```
    }
```

```
}
```

```
    HANDLE_ERROR( cudaGetDeviceProperties( &prop, i ) );
```

```
    printf( "    --- General Information for device %d ---\n", i );
```

```
    printf( "Name:   %s\n", prop.name );
```

```
    printf( "Compute capability:  %d.%d\n", prop.major, prop.minor );
```

```
    printf( "Clock rate:  %d\n", prop.clockRate );
```

```
    printf( "Device copy overlap:  " );
```

```
    if (prop.deviceOverlap)
```

```
        printf( "Enabled\n" );
```

```
    else
```

```
        printf( "Disabled\n" );
```

```
    printf( "Kernel execution timeout :  " );
```

Querying Devices

- cudaSetDevice().
- cudaChooseDevice ()

```
#include "../common/book.h"

int main( void ) {
    cudaDeviceProp prop;
    int dev;

    HANDLE_ERROR( cudaGetDevice( &dev ) );
    printf( "ID of current CUDA device: %d\n", dev );

    memset( &prop, 0, sizeof( cudaDeviceProp ) );
    prop.major = 1;
    prop.minor = 3;
    HANDLE_ERROR( cudaChooseDevice( &dev, &prop ) );
    printf( "ID of CUDA device closest to revision 1.3: %d\n", dev );
    HANDLE_ERROR( cudaSetDevice( dev ) );
}
```

RUNNING IN PARALLEL

- Vector Addition on the Device

The syntax of kernel execution configuration is as follows

```
<<< M , T >>>
```

Going parallel

```
add<<< 1, 1 >>>();  
      ↓  
add<<< N, 1 >>>();
```

Instead of executing add() once, execute N times in parallel

kernel execution configuration <<<...>>>

Informs how many threads to launch on GPU during CUDA runtime .

Vector sums

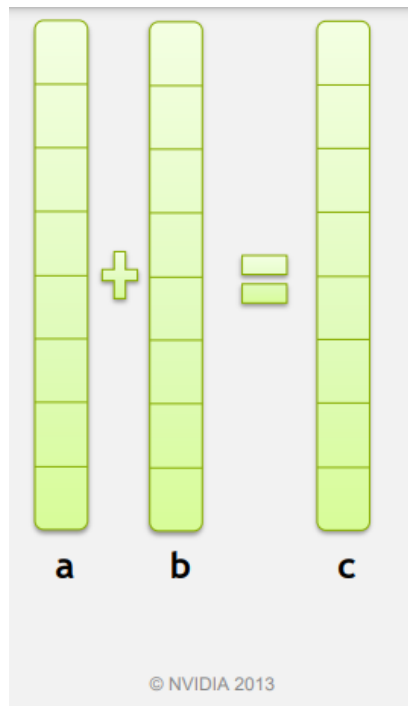
```
void add( int *a, int *b, int *c ) {  
    int tid = 0;    // this is CPU zero, so we start at zero  
    while (tid < N) {  
        c[tid] = a[tid] + b[tid];  
        tid += 1;    // we have one CPU, so we increment by one  
    }  
}
```

CPU CORE 1

```
void add( int *a, int *b, int *c )  
{  
    int tid = 0;  
    while (tid < N) {  
        c[tid] = a[tid] + b[tid];  
        tid += 2;  
    }  
}
```

CPU CORE 2

```
void add( int *a, int *b, int *c )  
{  
    int tid = 1;  
    while (tid < N) {  
        c[tid] = a[tid] + b[tid];  
        tid += 2;  
    }  
}
```



GPU vector sums

<<< N , 1 >>>

- `kernel<<<2,1>>>` => Creating two copies of the kernel and running them in parallel.
- With `kernel<<<256,1>>>()` => ???
- GPU runs N copies of our kernel code

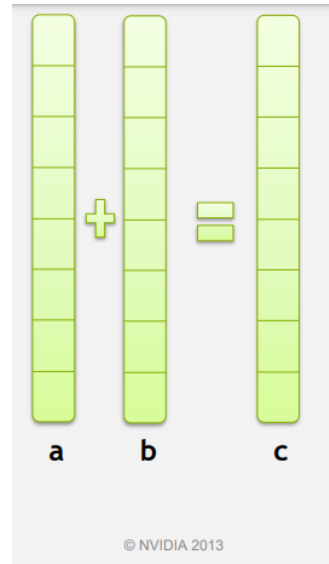
How can we tell from within the code which block is currently running??



Use of variable `blockIdx.x` (the kernel code itself)

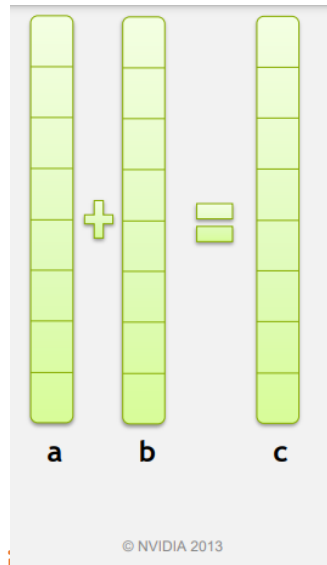
$N \leq 65,535$ – a hardware-imposed limit ($N \leq 2^{31} - 1$ from version 3.x and above)

```
__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];  
}
```



Built-in variable “blockIdx”

- blockIdx; this is one of the built-in variables that the CUDA.
- It contains the value of the block index for whichever block is currently running the device code
- Why `blockIdx.x`?:
 - CUDA C allows you to define a group of blocks in two dimensions. : `.x` and `.y`
 - matrix math or image processing
- collection of parallel blocks a *grid*.



By using `blockIdx.x` to index into the array, each block handles a different index

BLOCK 1

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 0;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 2

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 1;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 3

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 2;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 4

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 3;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

four blocks, all running through the same copy of the device code but having different values for the variable blockIdx.x.

```

#include "../common/book.h"

#define N 10

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

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

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

```

```

    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
                              cudaMemcpyHostToDevice ) );

    HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
                              cudaMemcpyHostToDevice ) );

    add<<<N,1>>>( dev_a, dev_b, dev_c );

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

    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    // free the memory allocated on the GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );

    return 0;
}

```

CUDA Threads

- • A block can be split into parallel threads

Using blocks:

```
__global__ void add(int *a, int *b, int *c){  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

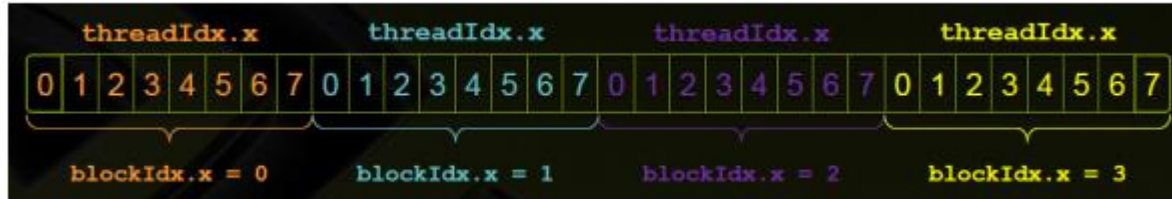
Using threads all in one block:

```
__global__ void add(int *a, int *b, int *c){  
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];  
}
```

```
main(){  
    ...  
    add<<<1, 100>>>>(dev_a, dev_b, dev_c);  
}
```

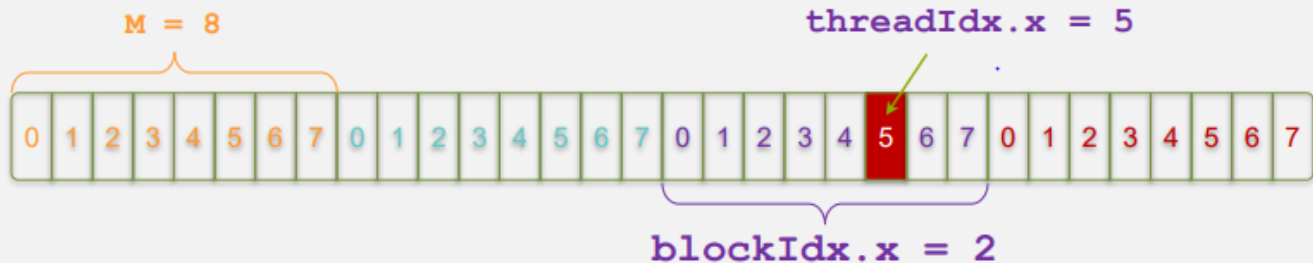
- Hardware limits the number of blocks in a single launch to 65,535.
- Hardware also limits the number of threads per block with which we can launch a kernel. –
For many GPUs,
maxThreadsPerBlock = 512 (or 1024, version 2.x above).

- Vector addition to use both blocks and threads
 - Consider indexing an array with one element per thread
 - We also use 8 threads per block.
 - With “M” threads/block a unique index for each thread is given by:
 - $\text{int index} = \text{threadIdx.x} + \text{blockIdx.x} * M;$



```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * M;  
          =          5      +          2      * 8;  
          = 21;
```



```
int nblocks=3, nthreads=4, nsize=3*4;
```

```
h_x = (float *)malloc(nsize*sizeof(float));  
cudaMalloc((void **)&d_x,nsize*sizeof(float));  
my_first_kernel<<<nblocks,nthreads>>>(d_x);  
cudaMemcpy(h_x,d_x,nsize*sizeof(float),  
cudaMemcpyDeviceToHost);
```

Block 0	Thread 0	Thread 1	Thread 2	Thread 3
Block 1	Thread 0	Thread 1	Thread 2	Thread 3
Block 2	Thread 0	Thread 1	Thread 2	Thread 3

```
__global__ void kernel( int *a )
```

```
{
```

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

```
    a[idx] = 7;
```

```
}
```

Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

```
__global__ void kernel( int *a )
```

```
{
```

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

```
    a[idx] = blockIdx.x;
```

```
}
```

Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

```
__global__ void kernel( int *a )
```

```
{
```

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

```
    a[idx] = threadIdx.x;
```

```
}
```

Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

Applications of CUDA

- Medical imaging
 - Two or more images need to be taken, and the film needs to be developed and read by a skilled doctor to identify potential tumor.
 - TechniScan: three- dimensional ultrasound imaging method.
 - The TechniScan Svava system relies on two NVIDIA Tesla C1060 processors in order to process the 35GB of data generated by a 15-minute scan.
 - within 20 minutes the doctor can manipulate a highly detailed, three-dimensional image

Passing Parameters & Data Transfer

- 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.

```
// File name: add.cu
#include <stdio.h>

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

int main(void){
    int c;
    int *device_c;
    cudaMalloc((void**)&device_c, sizeof(int));
    add<<<1, 1>>>(2, 7, device_c);
    cudaMemcpy(&c, device_c, sizeof(int), cudaMemcpyDeviceToHost);
    printf("2+7 = %d\n", c);
    cudaFree(device_c);
    return 0;
}
```

- `<<<N,1>>>();`
 - –The number “N” represents the number of parallel blocks (of threads) in which we would like the GPU to execute our kernel
 - `add<<< 256, 1>>>()` can be thought as that the runtime creates 256 copies of the kernel and runs them in parallel
- Built-in variable “`blockIdx`”

```
__global__ void add(int *a, int *b, int *c){  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

Block 0

`c[0] = a[0] + b[0];`

Block 1

`c[1] = a[1] + b[1];`

Block 2

`c[2] = a[2] + b[2];`

- Using blocks:

```
__global__ void add(int *a, int *b, int *c){  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- Using threads all in one block:

```
__global__ void add(int *a, int *b, int *c){  
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];  
}
```

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`

- **dim3** is an integer vector type that can be used in **CUDA** code.
 - Its most common application is to pass the grid and block dimensions in a kernel invocation.
 - It can also be used in any user code for holding values of 3 dimensions.


```

#include <stdio.h>
#include <cuda.h>
__global__ void dkernel() {
    if (threadIdx.x == 0 && blockIdx.x == 0 &&
        threadIdx.y == 0 && blockIdx.y == 0 &&
        threadIdx.z == 0 && blockIdx.z == 0) {
        printf("%d %d %d %d %d %d.\n", gridDim.x, gridDim.y, gridDim.z,
            blockDim.x, blockDim.y, blockDim.z);
    }
}

int main() {
    dim3 grid(2, 3, 4);
    dim3 block(5, 6, 7);
    dkernel<<<grid, block>>>();
    cudaThreadSynchronize();
    return 0;
}

```

How many times the kernel printf gets executed when the if condition is changed to *if (threadIdx.x == 0) ?*

Number of threads launched = $2 * 3 * 4 * 5 * 6 * 7$.
 Number of threads in a thread-block = $5 * 6 * 7$.
 Number of thread-blocks in the grid = $2 * 3 * 4$.

ThreadId in x dimension is in [0..5).
 BlockId in y dimension is in [0..3).

GPU Vector Sums

```
#include <stdio.h>
#include <cuda.h>
#include <cuda_runtime.h>
#include <curand_kernel.h>
#define N 512
__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_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);
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;
}
```

Review: 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

Acknowledgments

1. https://www.int.washington.edu/PROGRAMS/12-2c/week3/clark_01.pdf
2. <https://cuda-utorial.readthedocs.io/en/latest/tutorials/tutorial01/>
3. <https://developer.nvidia.com/blog/even-easier-introduction-cuda/>

- COMPUTATIONAL FLUID DYNAMICS

- Accurate simulations prove far too computationally expensive to be realistic,
- NVIDIA's CUDA Architecture to accelerate computational fluid dynamics unprecedented levels.

- Environmental science

- Molecular simulation of surfactant interactions with dirt, water, and other materials.
- Use of GPU-accelerated Highly Optimized object-oriented Many-particle Dynamics (HOOMD) simulation software = >16 times the performance of previous platforms.
- Simulation reduced from several weeks to a few hours

- Thank you .