Practical Course on Parallel Computing

(990179)

Week 11 – Heterogeneous Parallel Programming

Fei Zhang

Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen

Georg-August-University Göttingen

21.06.2016
Spring/Summer Term 2016

Administration

Time	lecture	Exercise	Assignment sheet
Tue. 21.6, 14:15-15:45	√		√
Thu. 23.6, 14:15-15:45		√	
Tue. 28.6, 10:15-11:45		√	
Tue. 28.6, 14:15-15:45	√	√	√
Thu. 30.6, 14:15-15:45		√	
Tue. 05.7,10:15-11:45		√	

Administration

- **Testbed**
  - You may use your own CUDA hardware if you wish.
  - Access CUDA machines via SSH.
    - The access method will be given in assignment sheet.

Week 11 - Heterogeneous Parallel Programming

What will we cover in this session?

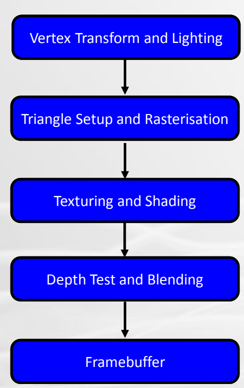
- **Basics**
  - Motivations for GPGPU
  - Heterogeneous computing
- **CUDA C**
  - Progamming model
  - Core concepts
  - Coding in CUDA C

Week 11 - Heterogeneous Parallel Programming

BASICS

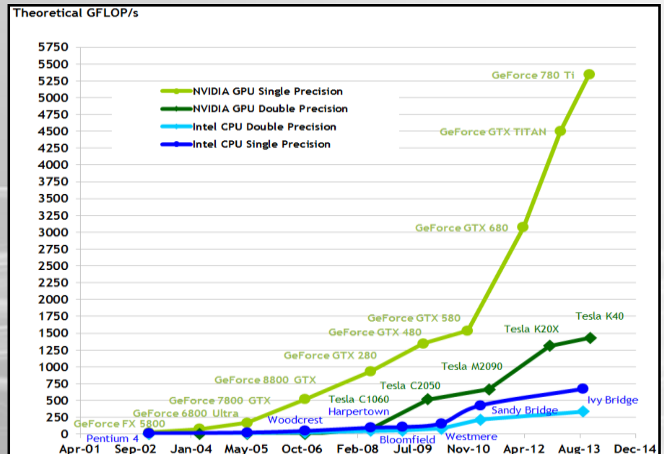
From GPU to GPGPU

- **The history of GPU**
  - Graphics pipeline have initially been implemented directly in HW.
  - Then high-level shading languages (DirectX, OpenGL and Cg) is used to exploit GPU for non-graphical applications. ➔ **GPGPU**
  - While GPGPU model demonstrated great speedups, it faced several drawbacks.
    - Require intimate knowledge of graphics APIs and GPU architecture.
    - High programming complexity.
    - Random memory read/write were not supported.
    - Lack of double precision support.
  - ➔ **CUDA**



Graphics Pipeline

Motivations for GPGPU

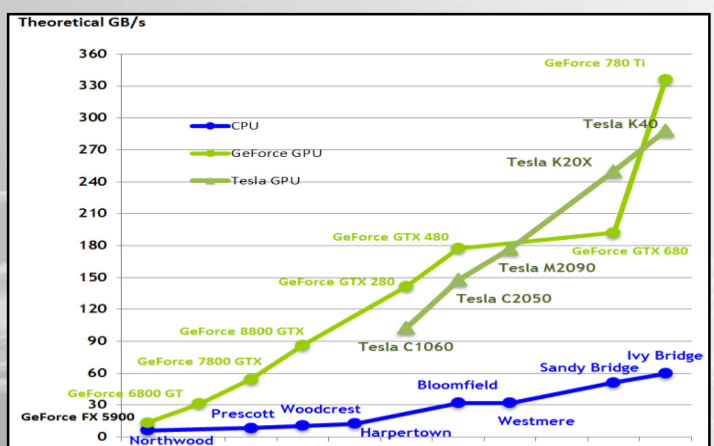


Floating-point operations per Second for the CPU and GPU

Source:http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Week 11 - Heterogeneous Parallel Programming

Motivations for GPGPU

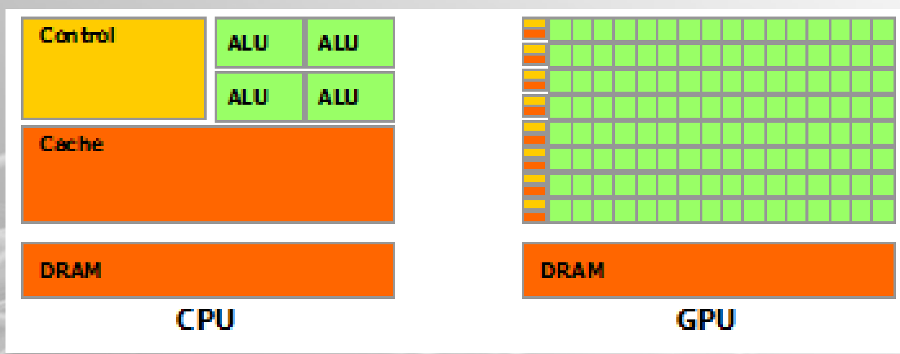


Memory bandwidth for the CPU and GPU

Source:http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Week 11 - Heterogeneous Parallel Programming

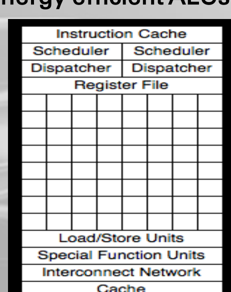
Motivations for GPGPU



GPU devotes more transistors to data processing

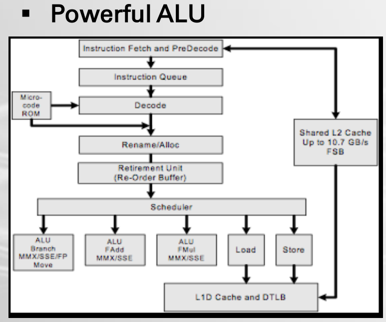
Motivations for GPGPU

- GPUs:Throughput oriented design
  - Small caches
  - Simple control
  - Energy efficient ALUs



GPU control logic

- CPUs: Latency oriented design
  - Large caches
  - Sophisticated control
  - Powerful ALU



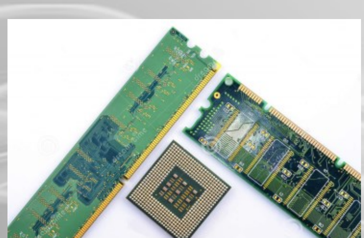
CPU control logic

Motivations for GPGPU

- GPU has evolved into a highly parallel, multithread processor with tremendous computational horsepower.
- The structure differences between CPU and GPU make GPU more suitable for compute-intensive, highly parallel computation.

Heterogeneous Computing

- Terminology:
  – ***Host :*** The CPU and its memory (host memory)
  – ***Device :*** The GPU and its memory (device memory)



Host

Device

Heterogeneous Computing

- An example

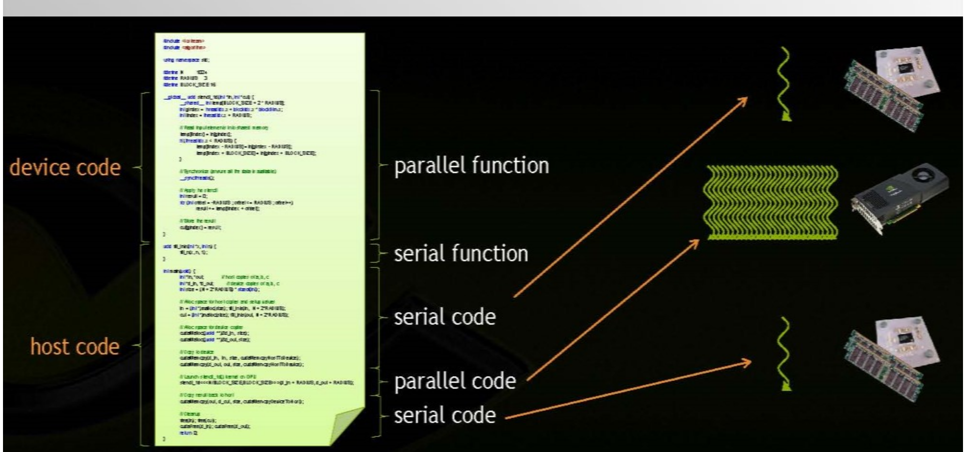
```
#include <iostream>
#include <algorithm>
using namespace std;
#define N 1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__ void mykernel(int *in, int *out) {
        ......
}

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}
```

Call device function

```
int main(void) {
        int *in, *out; // host copies of a, b, c
        int *d_in, *d_out; // device copies of a, b, c
        .......

        cudaMalloc((void **)&d_in, size);
        cudaMalloc((void **)&d_out, size);
        // Copy to device
        cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        mykernel<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in, d_out );

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

Week 11 - Heterogeneous Parallel Programming

Heterogeneous Computing



Week 11 - Heterogeneous Parallel Programming

CUDA C

Week 11 - Heterogeneous Parallel Programming

PROGRAMMING MODEL

Week 11 - Heterogeneous Parallel Programming

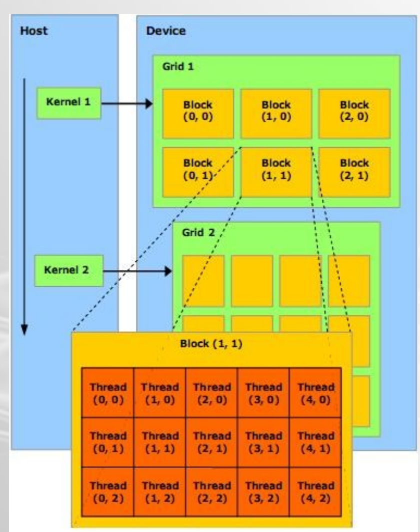
Kernels

- C functions are executed N times by N different CUDA threads, as opposed to only once like regular C functions.
- A kernel is defined by using the *__global__* declaration specifier.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
   int i = threadIdx.x;
   C[i] = A[i] + B[i];
}

int main() {
   ...
   // Kernel invocation with N threads
   VecAdd<<<1, N>>>(A, B, C);
   ...
}
```

Thread Hierarchy

- Kernel launches a grid of thread blocks.
- A single kernel is executed by several threads.
- Threads are grouped into blocks.



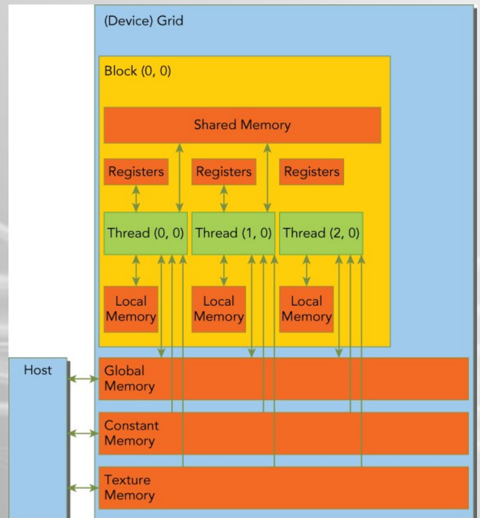
Thread Hierarchy

- **Dimensionality**
  - Threads and blocks can be 1, 2 or 3 dimensional.
    - The number of threads per block and the number of blocks per grid are specified in the <<<......>>> syntax.
  - Grids can be 1 or 2 dimensional.

Thread Hierarchy

- **Dimensionality**
  - The index of a thread and its thread ID related to each other.
    - For one-dimensional block, they are same.
    - For a two-dimensional block of size (Dx, Dy), the thread ID of a thread of index (x,y) is x+y*Dx.
    - For a three-dimensional block of size (Dx, Dy, Dz), the thread ID of a thread of index (x,y,z) is x+y*Dx+z*Dx*Dy.

Memory Hierarchy

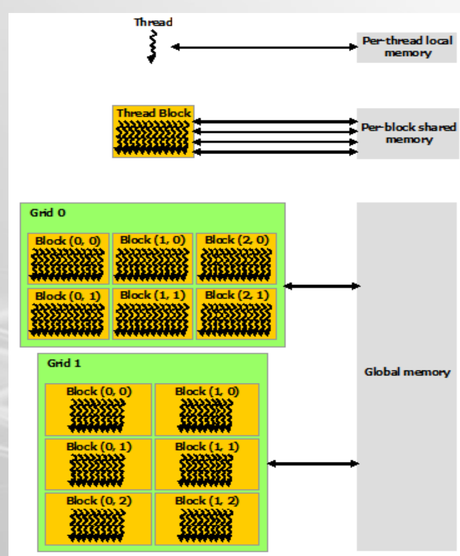


Source: https://www.safaribooksonline.com/library/view/professional-cuda-c/9781118739310/c04.xhtml#c4

Week 11 - Heterogeneous Parallel Programming

Memory Hierarchy

- Each thread has private local memory.
- Each thread block has shared memory visible to all threads of the block.
- All threads have access to same global memory, constant memory.



Week 11 - Heterogeneous Parallel Programming

Memory Hierarchy

Memory	Scope	Lifetime
register	Thread	Thread
local	Thread	Thread
shared	Block	Block
global	Grid	Application
constant	Grid	Application

CORE CONCEPTS

CUDA

- A general purpose parallel computing platform and programming model.
- Supports different languages, application programming interfaces, or directives-based approaches, such as FORTRAN, DirectCompute, OpenACC.
- CUDA C
  - Based on industry-standard C/C++.
  - Small set of extensions to enable heterogeneous computing.

CUDA C

- Consists of a minimal set of C language extensions and a runtime library.
- Defines kernels as C functions.
- Uses some new syntax to specify the grid and block dimension each time the function is called.

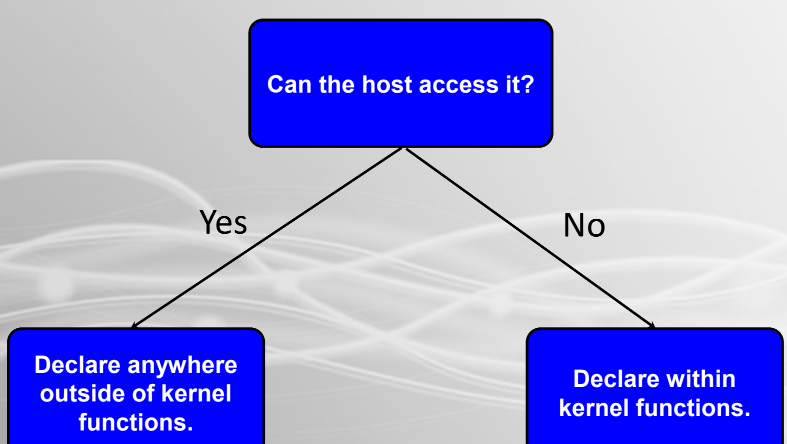
Variable type qualifiers

- Specify the memory location on the device of a variable

Qualifier	Where does the variable reside	Lifetime
__device__	Global memory space	Application
__constant__	Constant memory space	Application
__shared__	Shared memory space of a thread block.	Block

- Shared memory is faster than global memory. Try to use shared memory as much as possible!

Declaring Variables



Built-in vector types

- char, short, int, long, longlong, float, double
  - Vector types derived from the basic integer and floating-point types.
  - Built with a constructor function of the form make_<type name>
    - int2 make_int2(int x, int y); //creates a vector of type int2 with value (x,y).
  - Use x, y, z and w to access 1^{st}, 2^{nd} , 3^{rd} and 4^{th} components.
- dim3
  - An integer vector type based on uint3.
  - Used to specify dimensions.
  - Any component left unspecified is initialized to 1.
    - dim3 blocks2D(5,5); = dim3 blocks2D(5,5,1);

Built-in variables

Variable	Type	Description
gridDim	dim3	Contains the dimensions of the grid
blockIdx	uint3	Contains the block index within the grid
blockDim	dim3	Contains the dimensions of the block
threadIdx	uint3	Contains the thread index within the block

Function type qualifiers

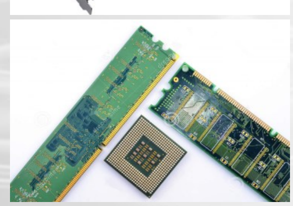
- Specify whether a function executes on the host or on the device and whether it is callable from the host or from the device.

Qualifier	Where does the function execute	Callability
__device__	Device	Device only
__global__	Device (must has void return type, and specify execution configuration)	• Callable from the host, • Callable from the devices (compute capability = 3.x)
__host__	host	Host only

Week 11 - Heterogeneous Parallel Programming

Device memory operations

- Host and device memory are separate entities.
  - *Device* pointers point to GPU memory.
  - *Host* pointers point to CPU memory



Week 11 - Heterogeneous Parallel Programming

Device memory operations

- Allocate memory space
  - *cudaMalloc(void** devPtr, size_t size)*
    - *devPtr* – Pointer to allocated device memory.
    - *size* – Requested allocated size in bytes.
- Free memory space
  - *cudaFree(void* devPtr)*
    - *devPtr* – Device pointer to memory to free.

Week 11 - Heterogeneous Parallel Programming

Device memory operations

- Copy data between host and device
  - *cudaMemcpy(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind)*
    - *dst* – Destination memory address.
    - *src* – Source memory address.
    - *count* – Size in bytes to copy.
    - *kind* –Type of transfer. Types can be:
      - *cudaMemcpyHostToHost*
      - *cudaMemcpyHostToDevice*
      - *cudaMemcpyDeviceToHost*
      - *cudaMemcpyDeviceToDevice.*

Week 11 - Heterogeneous Parallel Programming

Execution configuration

- Defines the dimension of the grid and blocks.
- **Syntax:** *<<<Dg, Db, Ns, S>>>*
  - *Dg* : dim3 type, the dimension and size of the grid.
  - *Db* : dim3 type, the dimension and size of each block.
  - *Ns* : optional, default is 0.
  - *S* : optional, defaut is 0.
- The function call will fail if *Dg* or *Db* are bigger than the maximum sizes allowed for the device.

Week 11 - Heterogeneous Parallel Programming

Execution configuration

- Example

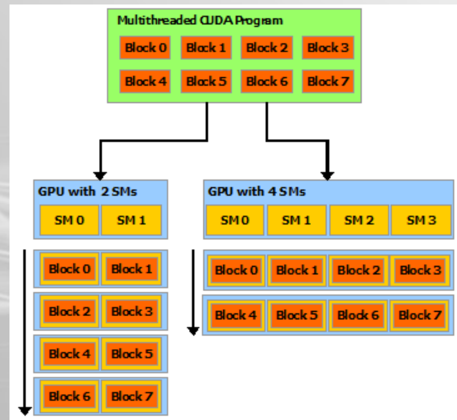
```
__global__ kernel (int *a, int *b, int *c)
{
    ...
}

void main(){
    ......

    dim3 blocks(nx, ny, nz);
    dim3 threadsPerBlock (mx, my, mz);

    kernel <<<blocks, threadsPerBlock>>> (ah, bh, ch);
}
```

Week 11 - Heterogeneous Parallel Programming

Scalability

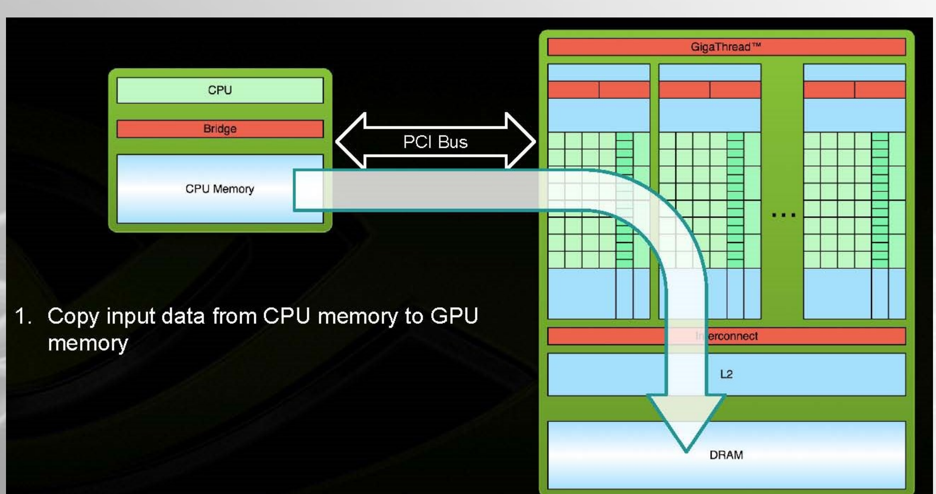
- A compiled CUDA program can execute on any number of multiprocessors.

Automatic scalability



Week 11 - Heterogeneous Parallel Programming

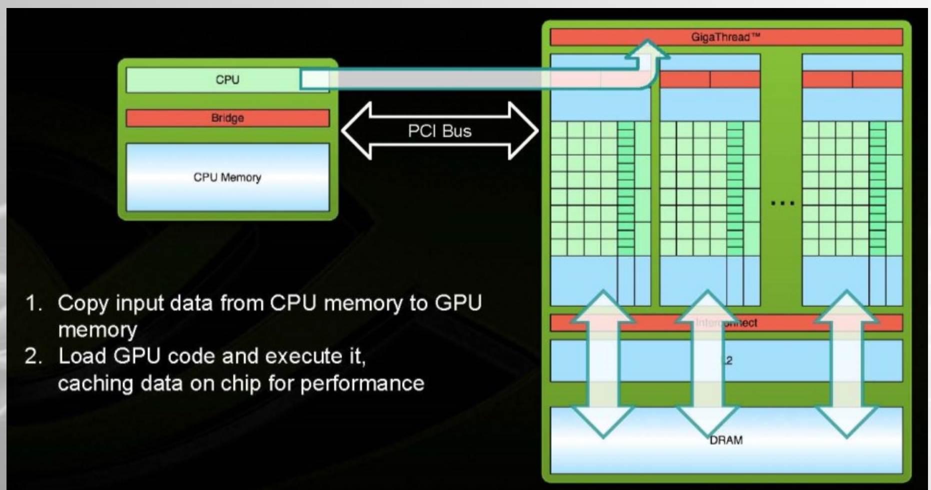
Program structure



Source: https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf

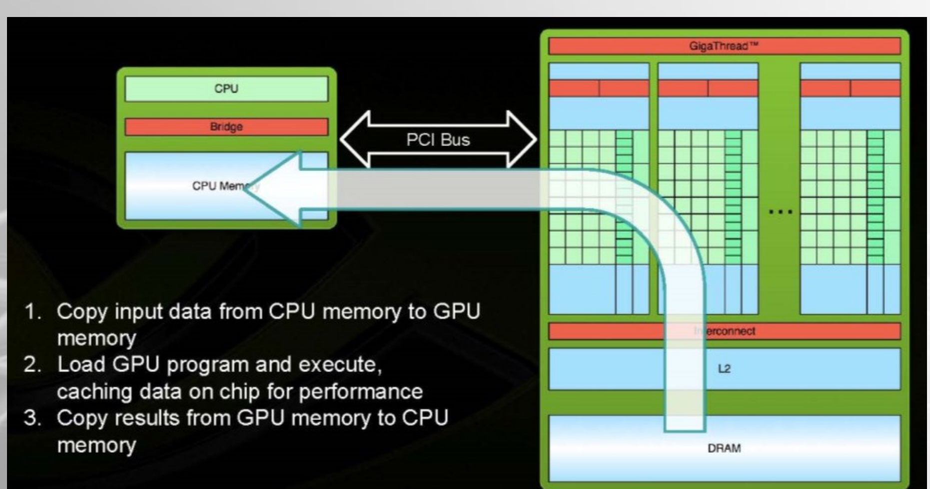
Week 11 - Heterogeneous Parallel Programming

Program structure



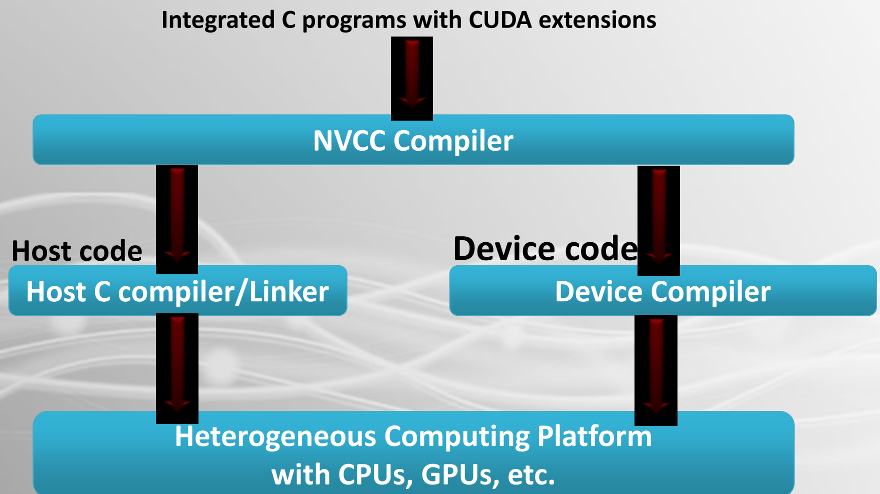
Week 11 - Heterogeneous Parallel Programming

Program structure



Week 11 - Heterogeneous Parallel Programming

Compiling CUDA program



Week 11 - Heterogeneous Parallel Programming

Restrictions for device code

- No recursion.
- No static variable.
- No function pointer.
- Can only access device memory.
- All threads in a grid execute the same kernel function.

Week 11 - Heterogeneous Parallel Programming

CODING IN CUDA

Hello World!

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

- Standard C that runs on the host.
- NVIDIA compiler (nvcc) can be used to compile programs with no device code.

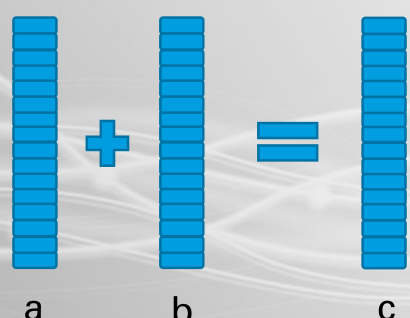
Hello World! with device code

```
__global__ void mykernel() {
}

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

```
$ nvcc –o helloworld helloworld.cu
$ ./helloworld
Output: Hello World!
```

- CUDA file with extension *.cu, which contain mix of host and device code.
- CUDA C/C++ keyword *__global__* indicates a function that:
  - Runs on the device
  - Is called from the host

- But wait… GPU computing is about massive parallelism!



Coding in CUDA

- How do we run code in parallel on the device?

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

- Instead of executing once, execute N times in parallel.

Parallel with Blocks

- A kernel

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

- add() will be called from the host, and executed on the device
- Using blockIdx.x to index into the array, each block handles a different element of the array.

Parallel with Blocks

```
#define N 512
int main(void) {

    int *a, *b, *c; // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Week 11 - Heterogeneous Parallel Programming

Parallel with Blocks

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Week 11 - Heterogeneous Parallel Programming

Parallel with Blocks

- On the device, each block can execute in parallel:

Block 0

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

Block 1

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

Block 2

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

Block 3

c[3]=a[3]+b[3]

⋮

Block 510

c[510]=a[510]+b[510]

Block 511

c[511]=a[511]+b[511]

Parallel with Blocks: Review

- Using *__global__* to declare a function as device code.
- CPU and GPU have separated memory.
- Pass and copy parameters from host to device.
  - cudaMalloc();
  - cudaMemcpy();
  - cudaFree();
- Launch parallel kernels.
  - Launch N copies of kernel with <<<N,1>>>;
  - Use blockIdx.x to access block index.

Parallel with Threads

- A block can be split into parallel threads.

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

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

- Use threadIdx.x instead of blockIdx.x.

Parallel with Threads

```
#define N 512
int main(void) {

    int *a, *b, *c; // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

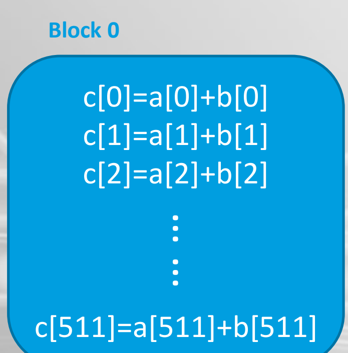
Parallel with Threads

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N threads
    add<<<1,N>>>(d_a, d_b, d_c);   //change from <<<N,1>>>

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Week 11 - Heterogeneous Parallel Programming

Parallel with Threads

- On the device, threads execute in parallel:

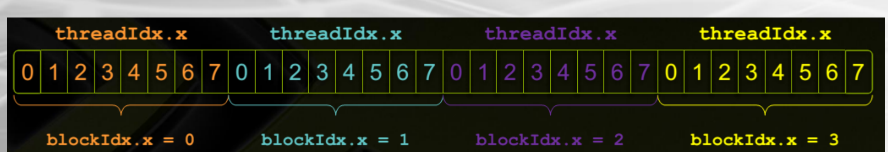


Week 11 - Heterogeneous Parallel Programming

Combine Blocks with Threads

- Indexing arrays with blocks and threads
  - No longer as simple as using blockIdx.x and threadIdx.x.
  - With M threads per block, a unique index for each thread is given by:

    int index = threadIdx.x + blockIdx.x * blockDim.x



Combine Blocks and Threads

- A block can be split into parallel threads.

```
__global__ void add(int *a, int *b, int *c) {
    //compute the element index
    int index = threadIdx.x + blockIdx.x * blockDim.x

    c[index] = a[index] + b[index];
}
```

- What changes need to be made in main()?

Combine Blocks and Threads

```
#define N 512
int main(void) {

    int *a, *b, *c; // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```


Combine Blocks and Threads

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with threads and blocks
    int threadsPerBlock = 64;
    int blocks = N/threadsPerBlock;
    add<<<blocks, threadsPerBlock>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Combine Blocks and Threads

- On the device, threads execute in parallel:

Block 0

c[0]=a[0]+b[0]
c[1]=a[1]+b[1]
c[2]=a[2]+b[2]
⋮
⋮
c[63]=a[63]+b[63]

Block 1

c[64]=a[64]+b[64]
c[65]=a[65]+b[65]
c[66]=a[66]+b[66]
⋮
⋮
c[127]=a[127]+b[127]

… …

Questions