中国科学院大学
University of Chinese Academy of Sciences

Introduction to CUDA

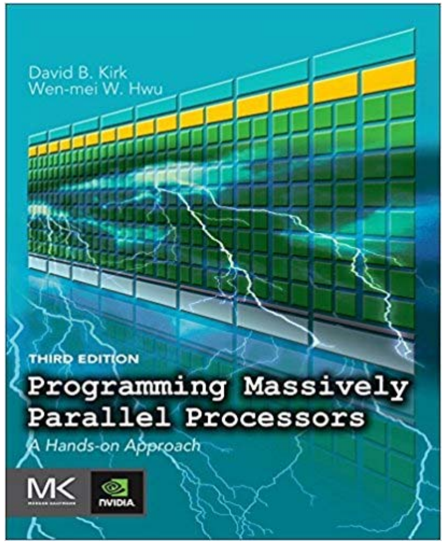
(2) Data Parallel Computing

Reference

- CUDA C Programming Guide,
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

- **Programming Massively Parallel Processors,**
  - **A Hands-on Approach**
  - **Third Edition**

  - **Chapter 2**



Content

- Data Parallelism
  - Example – color to grey conversion

- CUDA C Program Structure
  - Example – vector add

DATA PARALLELISM

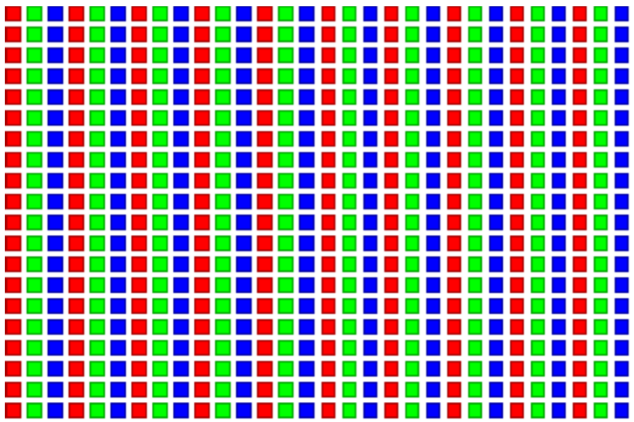
- Usually having *too much data* to be processed make software run slowly:
  - Consumer applications manipulate images or videos, with millions to trillions of pixels.
  - Scientific applications model fluid dynamics using billions of grid cells.
  - Molecular dynamics applications must simulate interactions between thousands to millions of atoms.
  - Airline scheduling deals with thousands of flights, crews, and airport gates.

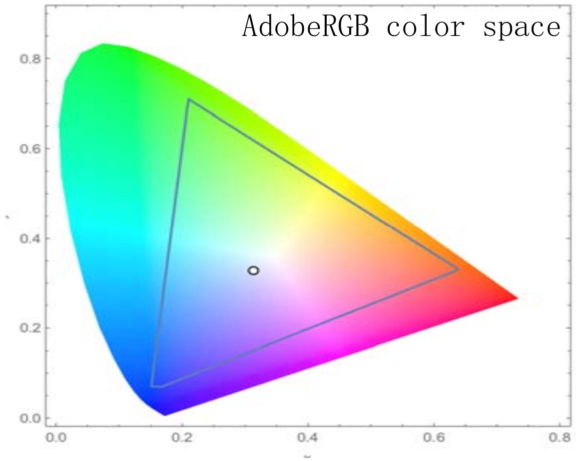
DATA PARALLELISM

- Independence
  - Importantly, most of these pixels, particles, cells, interactions, flights, and so on can be dealt with largely independently.
- Independent evaluation is the basis of *data parallelism*:
  - (re)organize the computation around the data,
  - such that we can execute the resulting independent computations in parallel

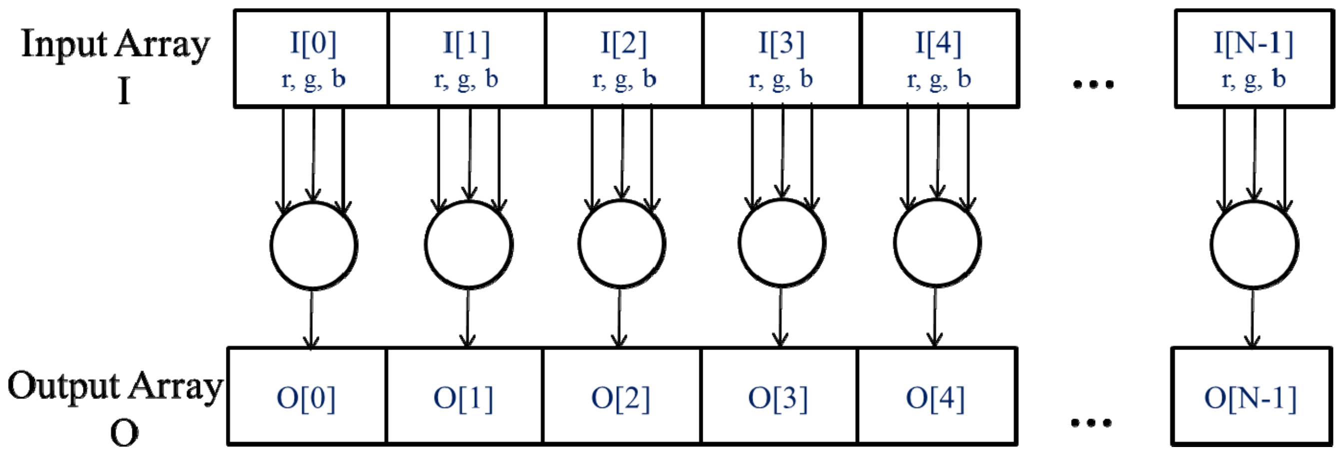
Example of data parallel

Conversion of a color image to grey–scale image





The pixels can be calculated independently of each other



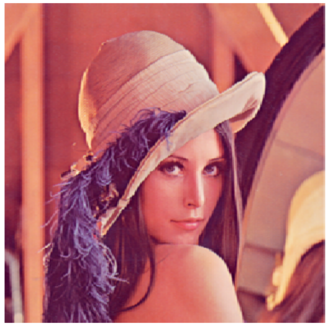
The pixels can be calculated independently of each other during colour to greyscale conversion.

colorToGreyscaleConversion ‘Preview’

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConvertion(unsigned char * Pout,  unsigned char * Pin,
                    int width, int height) {

 int Col =   threadIdx.x + blockIdx.x * blockDim.x;
 int Row = threadIdx.y + blockIdx.y * blockDim.y;

 if (Col < width && Row < height) {
   // get 1D coordinate for the grayscale image
   int greyOffset = Row*width + Col;
   // one can think of the RGB image having
   // CHANNEL times columns of the gray scale image
   int rgbOffset = greyOffset*CHANNELS;
   unsigned char r =  rgbImage[rgbOffset      ]; // red value for pixel
   unsigned char g = rgbImage[rgbOffset + 1]; // green value for pixel
   unsigned char b = rgbImage[rgbOffset + 2]; // blue value for pixel
   // perform the rescaling and store it
   // We multiply by floating point constants
   grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
 }
}
```

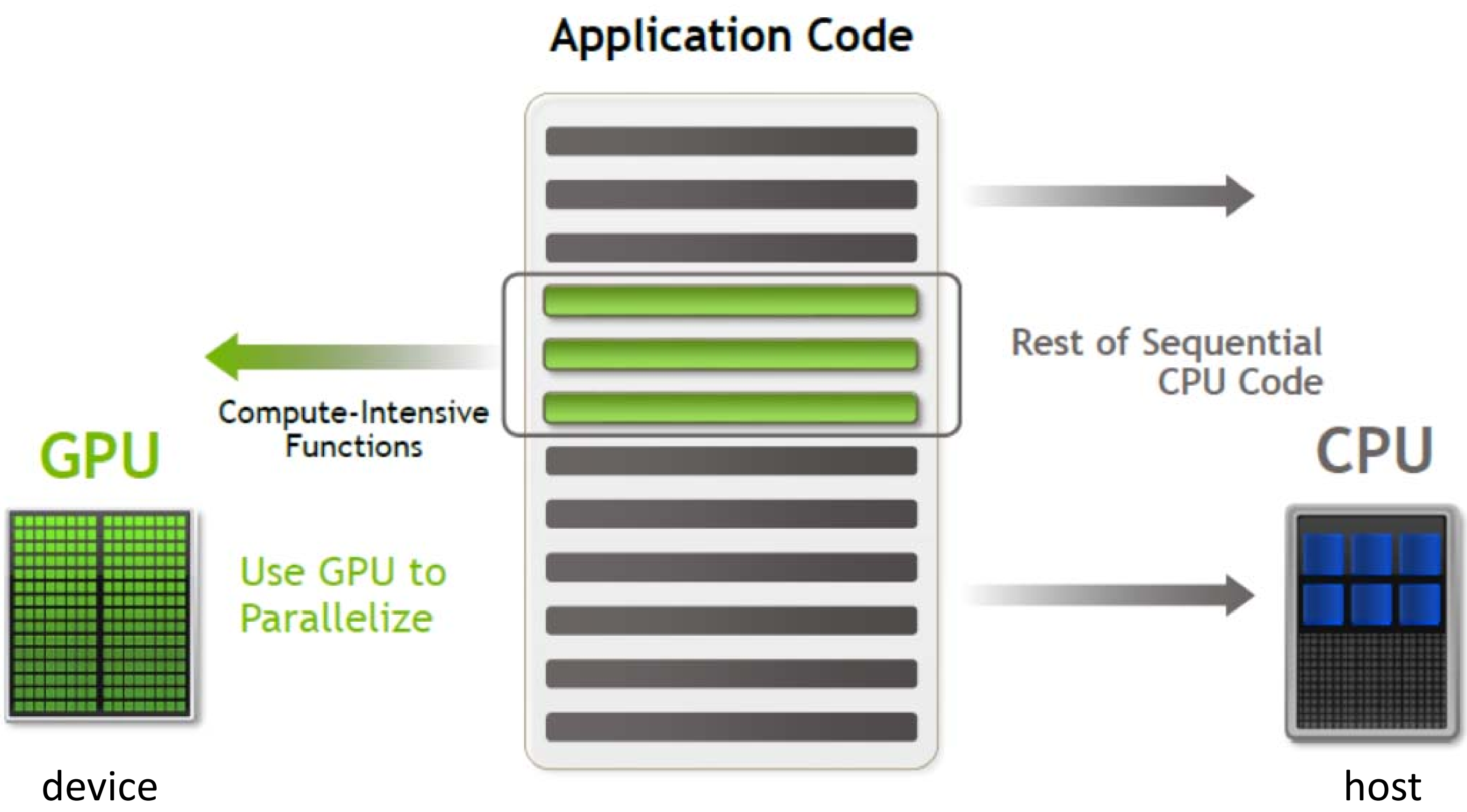


Content

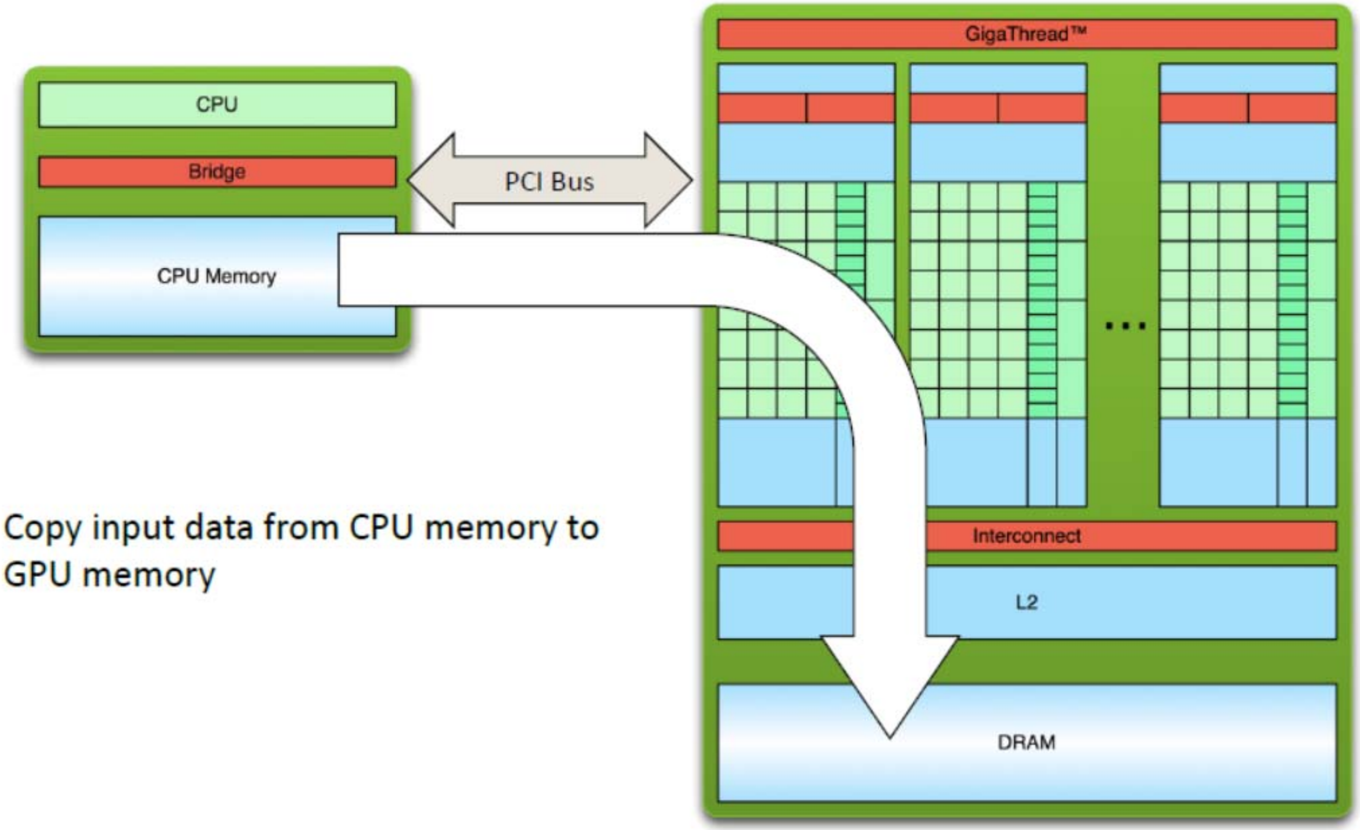
- Data Parallelism
  - Example – color to grey conversion

- CUDA Programming model
  - CUDA C Structure
  - Example – vector add
  - Memory and Data Transfer
  - Kernel Function
  - Kernel Launch

Heterogeneous parallel computing

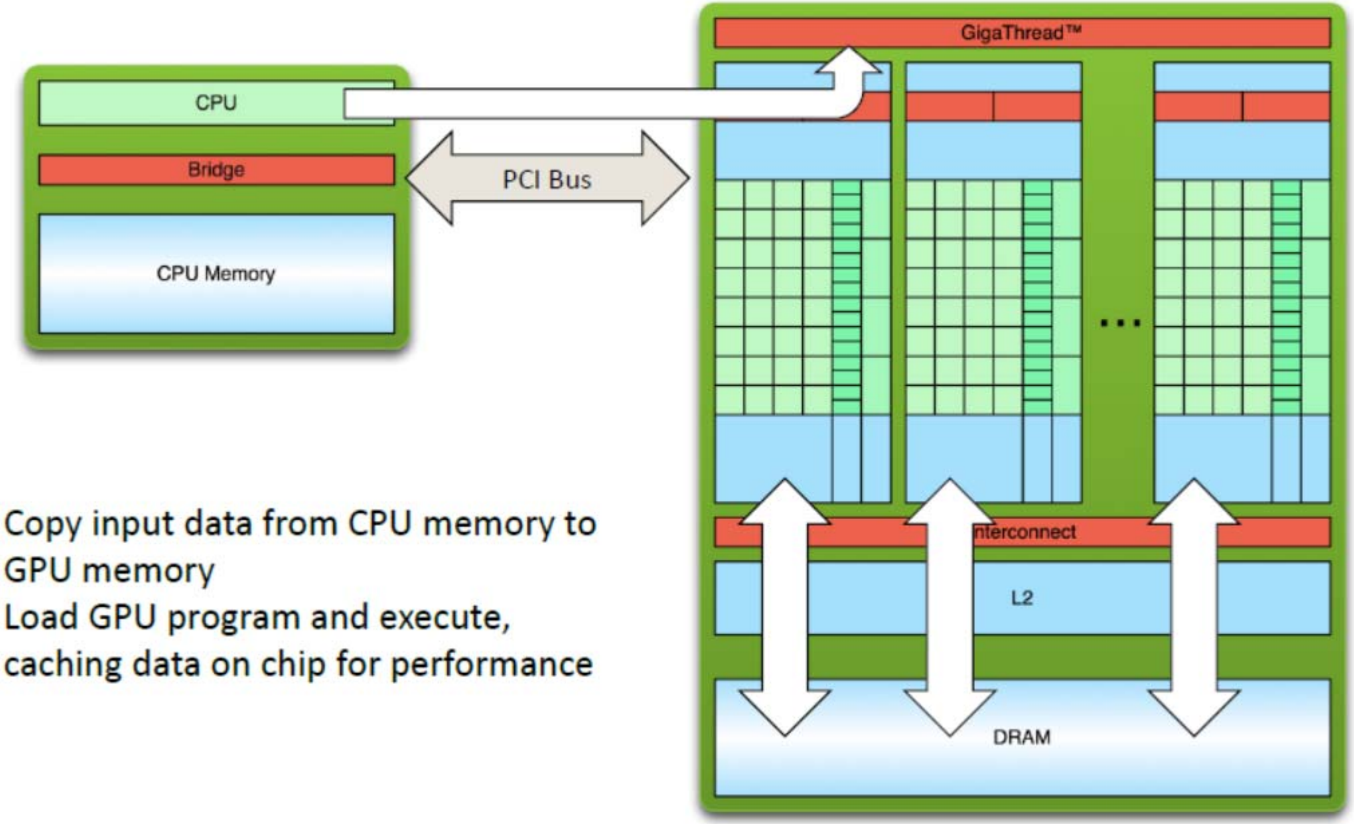


Heterogeneous parallel computing



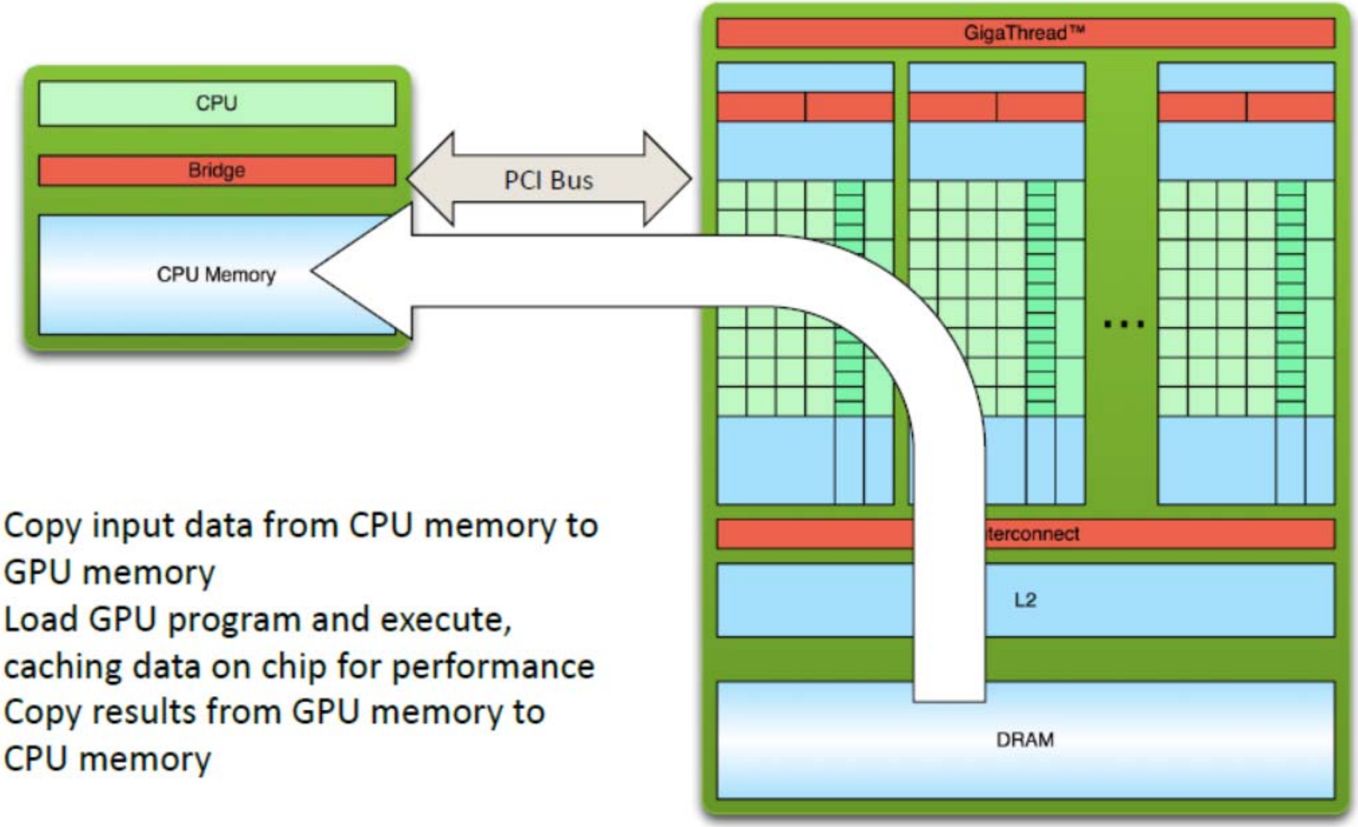
1. Copy input data from CPU memory to GPU memory

Heterogeneous parallel computing



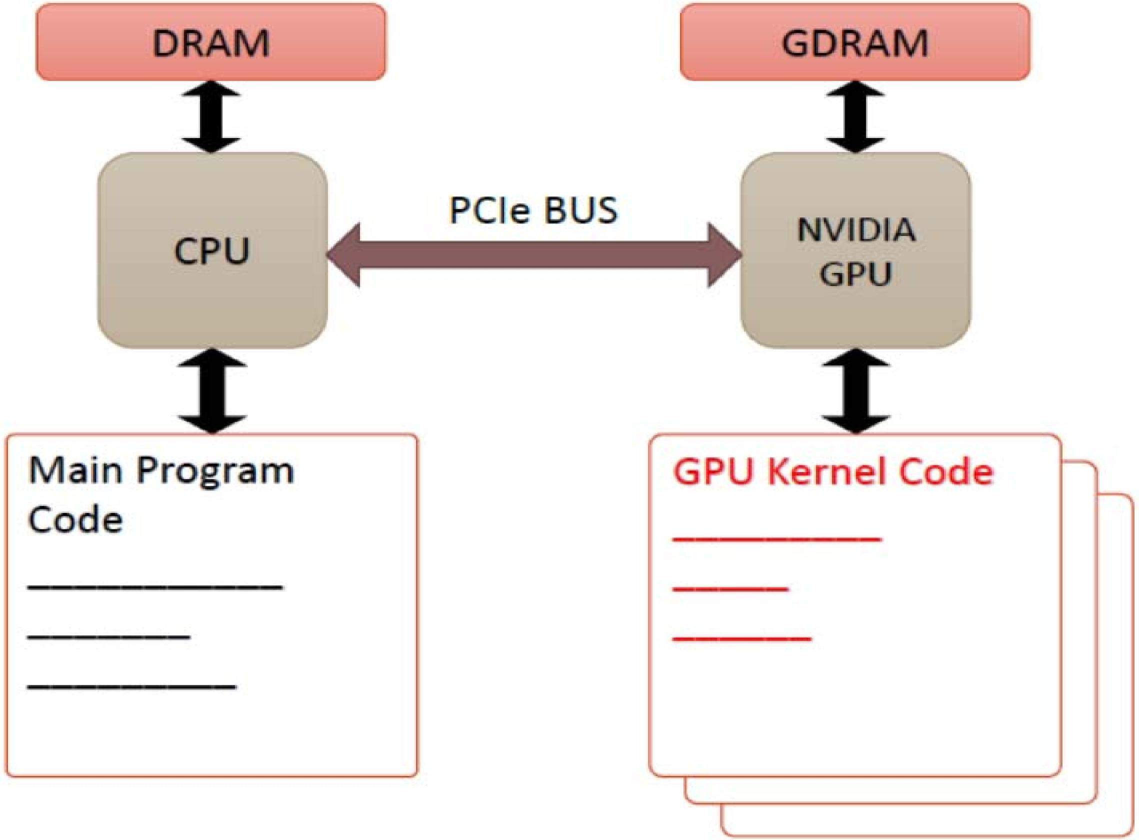
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Heterogeneous parallel computing



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

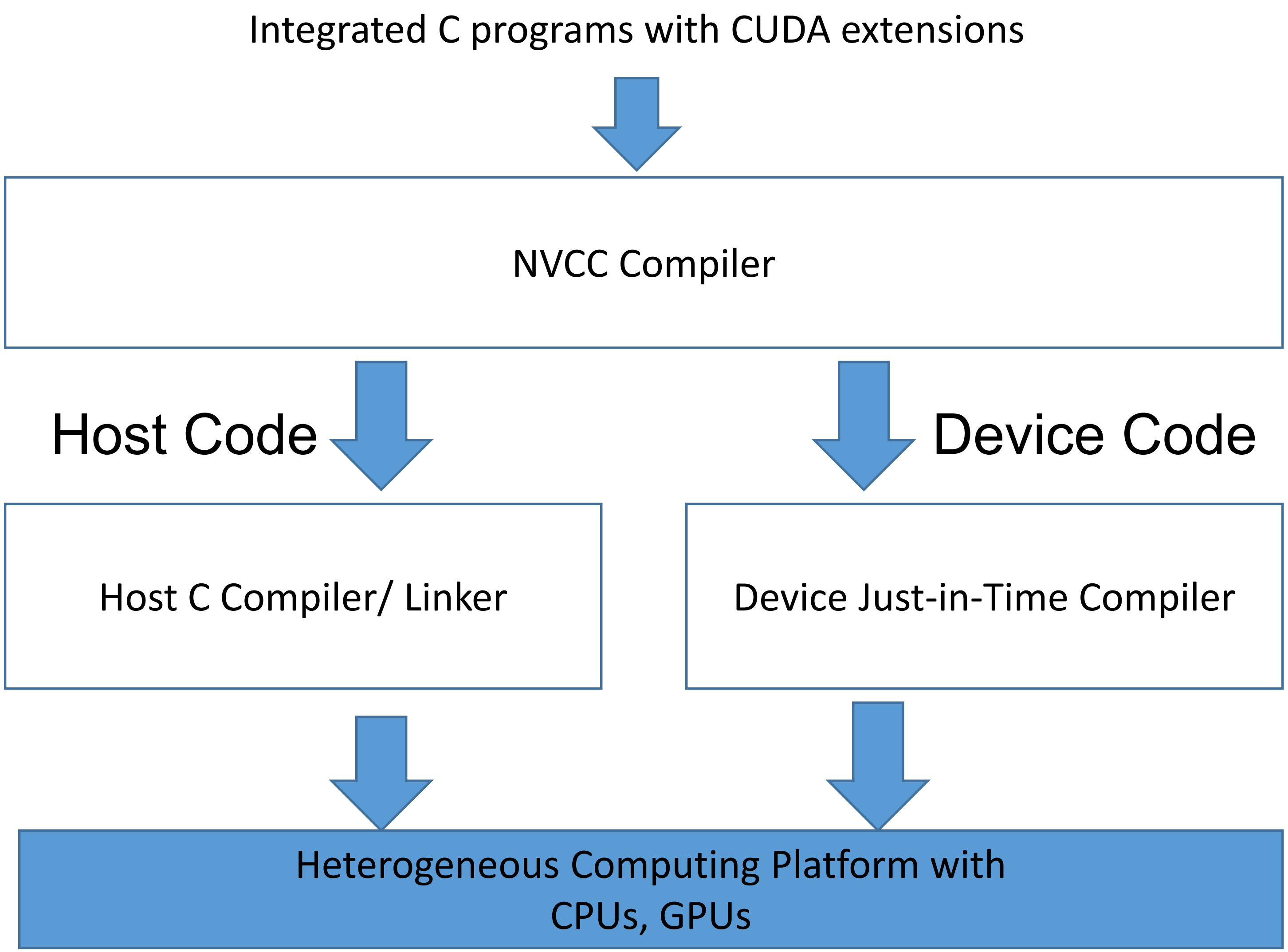
Heterogeneous parallel computing



Host code

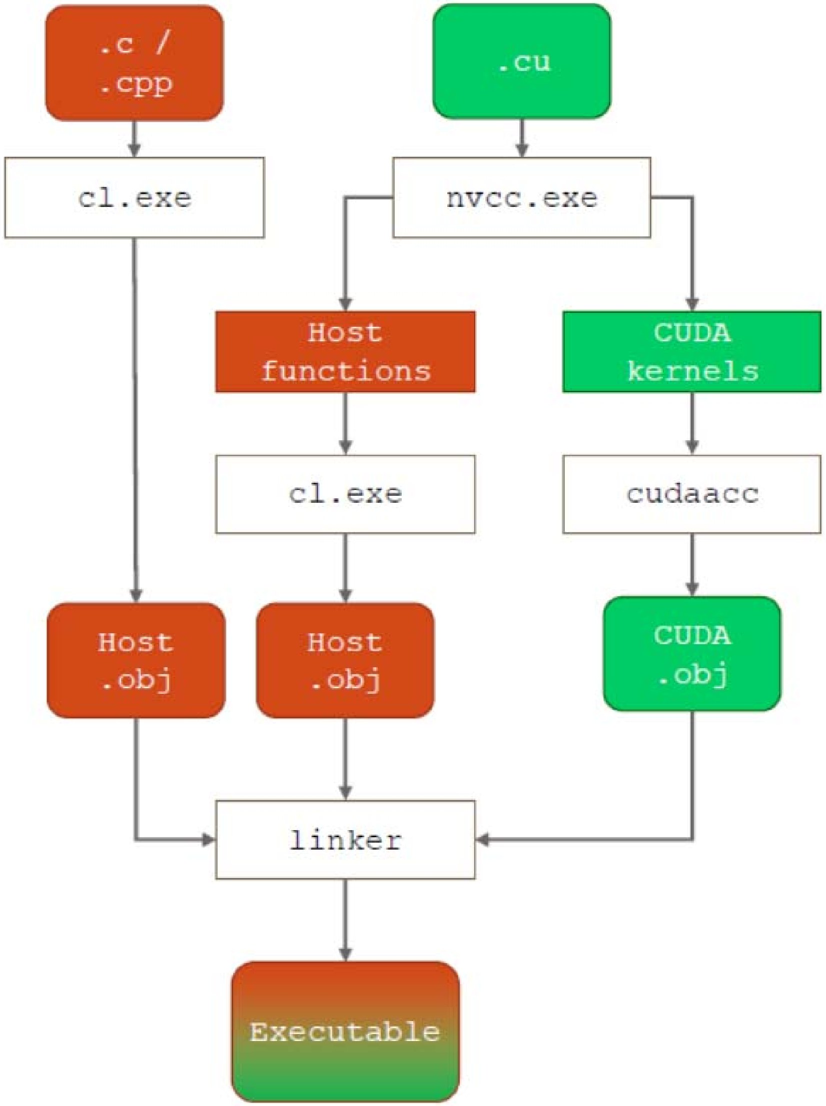
Device kernels

Compiling A CUDA Program



Compiling A CUDA Program (MSVC)

- CUDA source file (*.cu) are compiled by nvcc
- An existing cuda.rules file creates property page for CUDA source files
  - Configures nvccin the same way as configuring the C compiler
  - Options such as optimisation and include directories can be inherited from project defaults
- C and C++ files are compiled with cl(MSVCC compiler)



CUDA/OpenCL – Execution Model

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

Serial Code (host)

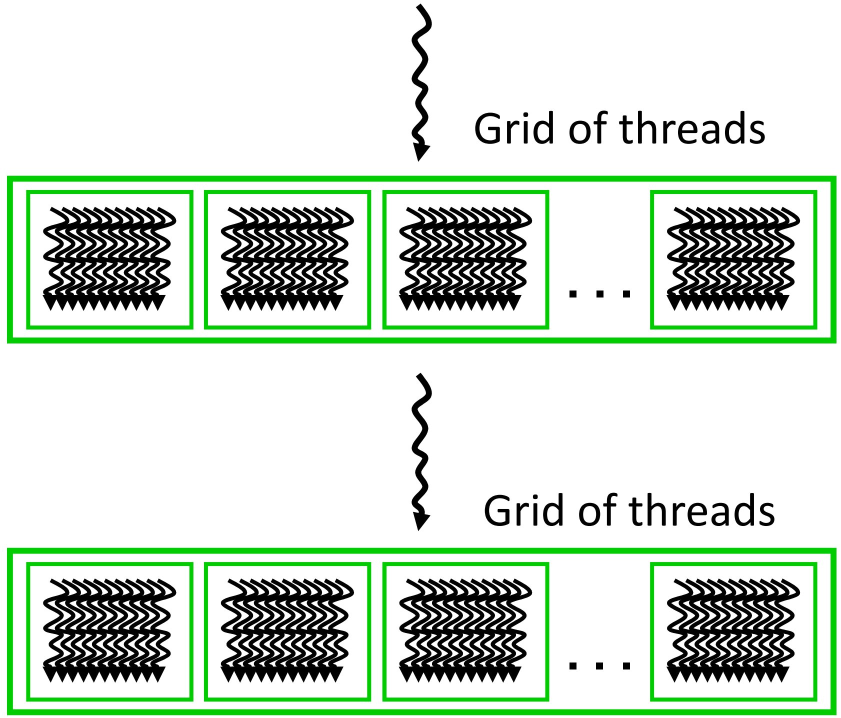
Parallel Kernel (device)

KernelA<<< nBlk, nTid >>>(args);

Serial Code (host)

Parallel Kernel (device)

KernelB<<< nBlk, nTid >>>(args);



Simplified CUDA model

Content

- Data Parallelism
  - Example – color to grey conversion

- CUDA Programming model
  - CUDA C Structure
  - Example – vector add
  - Memory and Data Transfer
  - Kernel Function
  - Kernel Launch

Vector Addition: Traditional

```
// Compute vector sum h_C = h_A+h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  for (int i = 0; i < n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements each
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

A simple traditional vector addition C code example.

Vector Addition: Traditional

- Pointers and Array in C:
  - In the C language, a pointer can be used to access variables and data structures. A pointer variable *P* can be declared with:

```
float *P;
```

  - An array in a C program can be accessed through a pointer that points to its 0th element.
    - For example, the statement P=&(A[0]) makes P point to the 0th element of array A.
  - P[i] becomes a synonym for A[i].
  - In fact, the array name A is in itself a pointer to its 0th element.

Vector Addition: Traditional

- Pointers and Array in C:
  - passing an array name A as the first argument to function call to vecAdd makes the function's first parameter h_A point to the 0th element of A.
  - We say that A is passed by reference to vecAdd.
  - As a result, h_A[i] in the function body can be used to access A[i].

Vector Addition: CUDA

```
#include <cuda.h>
…
void vecAdd(float* A, float* B, float* C, int n)
{
  int  size = n* sizeof(float);
  float  *d_A *d_B, *d_C;
  …
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code – to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```

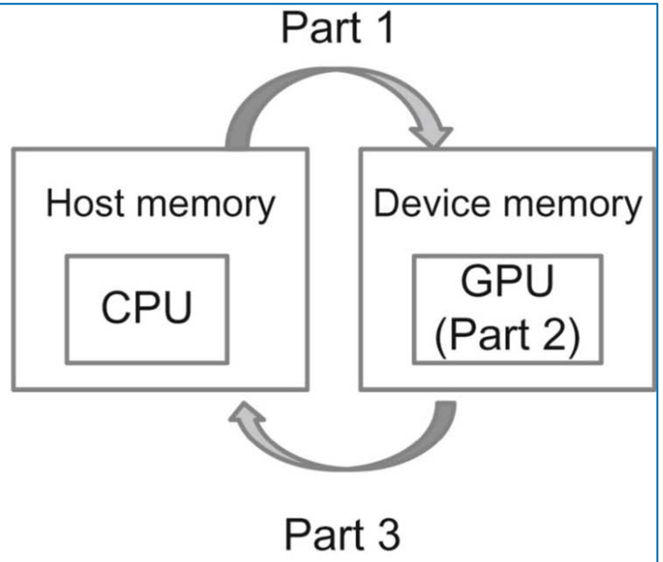


FIGURE 2.6: Outline of a revised vecAdd function that moves the work to a device.

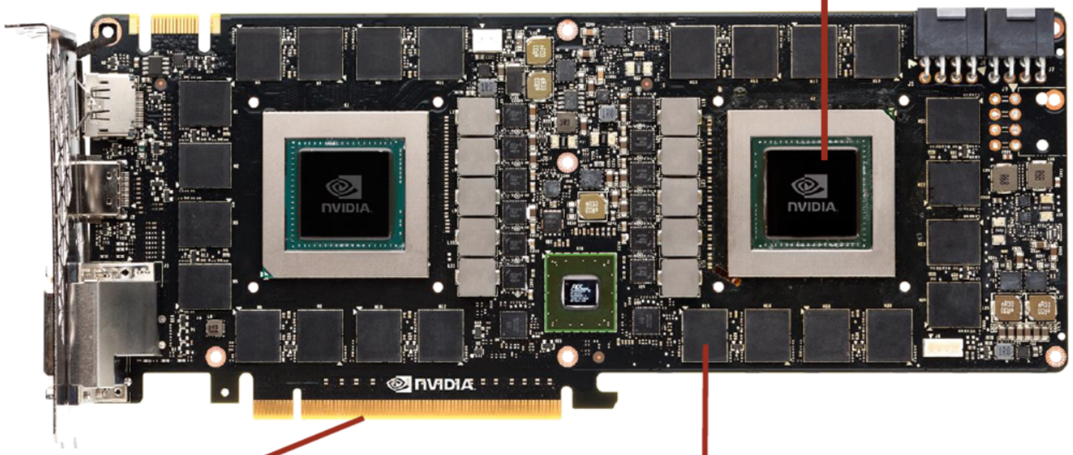
Content

- Data Parallelism
  - Example – color to grey conversion

- <u>CUDA Programming model</u>
  - CUDA C Structure
  - Example – vector add
  - <u>Memory and Data Transfer</u>
  - Kernel Function
  - Kernel Launch

Device Global Memory

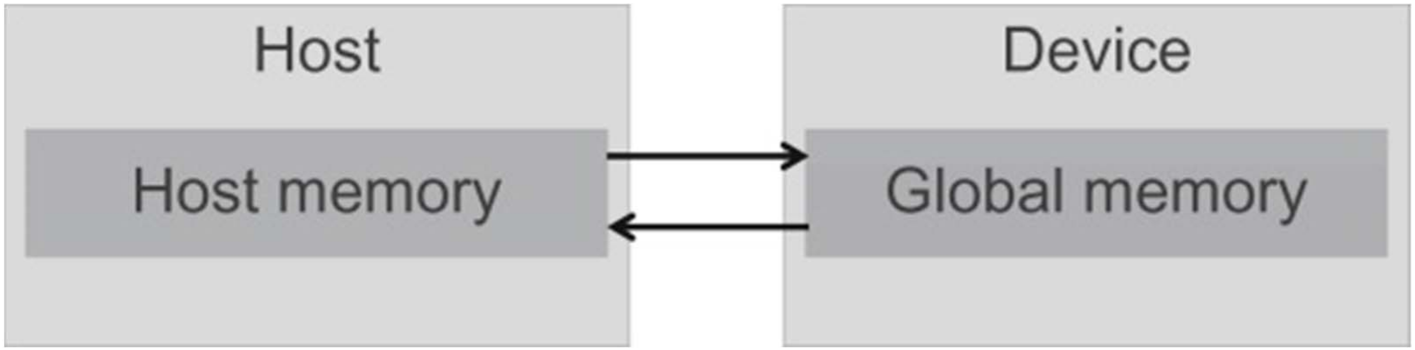
- Device Memory:



  - Global Memory

Device Global Memory

- Host memory and device global memory:



The device global memory can be accessed by the host to transfer data to and from the device.

Assuming no unified memory space.

Device Global Memory

- CUDA API functions for managing device global memory:

cudaMalloc()

- Allocates object in the device global memory
- Two parameters
  - **Address of a pointe**r to the allocated object
  - **Size** of allocated object in terms of bytes

cudaFree()

- Frees object from device global memory
  - **Pointer** to freed object

cudaMalloc() is a generic function that is not restricted to any particular type of objects.

Device Global Memory

- cudaMalloc()
  - The address of the pointer variable should be cast to (void **) because the function expects a generic pointer;
  - is a generic function that is not restricted to any particular type of objects.

```
float *d_A;
int size=n * sizeof(float);
cudaMalloc((void**)&d_A, size);
...
cudaFree(d_A);
```

Device Global Memory

- CUDA API function for data transfer between host and device:

cudaMemcpy()

- Memory data transfer
- Requires four parameters
  - o Pointer to destination
  - o Pointer to source
  - o Number of bytes copied
  - o Type/Direction of transfer

A more complete version of vecAdd()

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code - to be shown later
    ...

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory for A, B, C
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

Error Checking in CUDA

- it is very important for a program to check and handle errors.
    - CUDA API functions return flags that indicate whether an error has occurred when they served the request.
    - In practice, we should surround the call with code that test for error condition and print out error messages so that the user can be aware of the fact that an error has occurred.

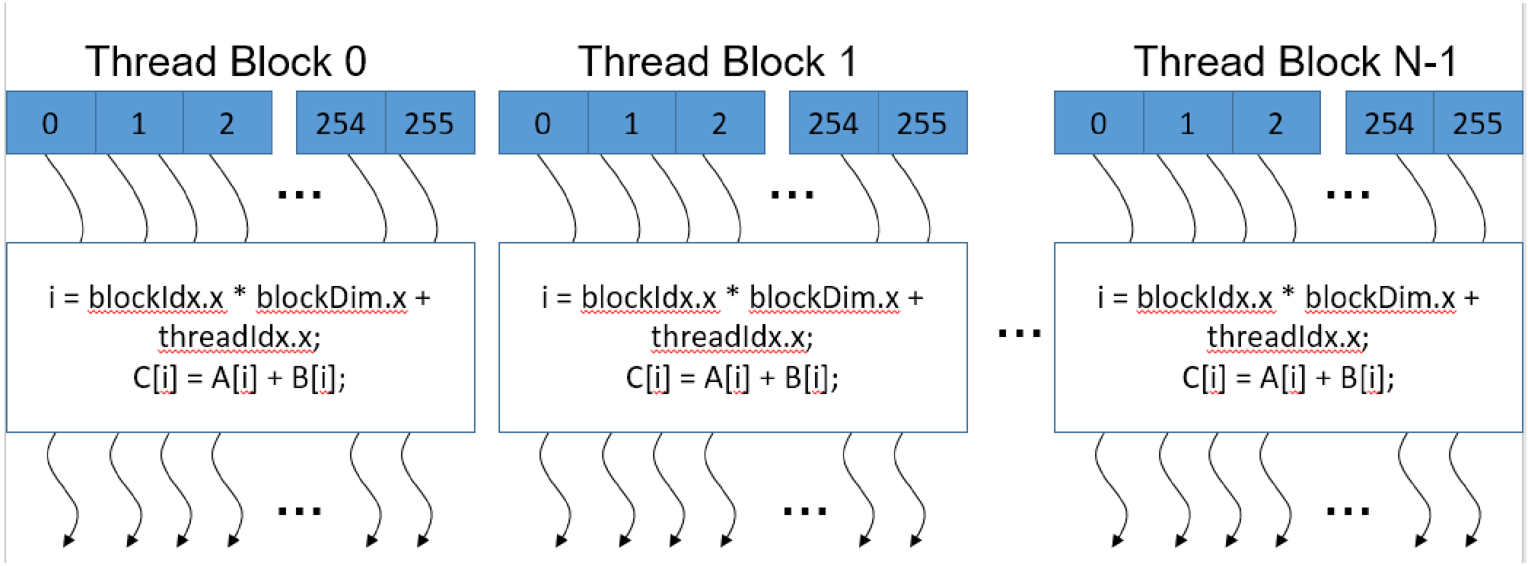
```
cudaError_t err=cudaMalloc((void **) &d_A, size);
if (error !=cudaSuccess) {
        printf("%s in %s at line %d\n", cudaGetErrorString(err),
        __FILE__,__LINE__);
        exit(EXIT_FAILURE);
}
```

Content

- Data Parallelism
  - Example – color to grey conversion
- <u>CUDA Programming model</u>
  - CUDA C Structure
  - Example – vector add
  - Memory and Data Transfer
  - <u>Kernel Function</u>
  - Kernel Launch

Arrays of Parallel Threads

- A CUDA kernel is executed by a grid (array) of threads that are organized into two-level hierarchy.
- Each grid is organized as an array of thread blocks.



–All threads in a grid run the same kernel code (SPMD)

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}
int vectAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
}
```

More on 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
  - Each "__" consists of two underscore characters
  - A kernel function must return `void`

More on 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

- `__device__` and `__host__` can be used together
  - This combination tells the compilation system to generate two versions of object files for the same function.
  - One is executed on the host and can only be called from a host function. The other is executed on the device and can only be called from a device or kernel function.

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
     int i = threadIdx.x + blockDim.x * blockIdx.x;
     if(i<n) C_d[i] = A_d[i] + B_d[i];
}
```

Where the loop went?
The answer is that the loop is now replaced with the grid of threads.

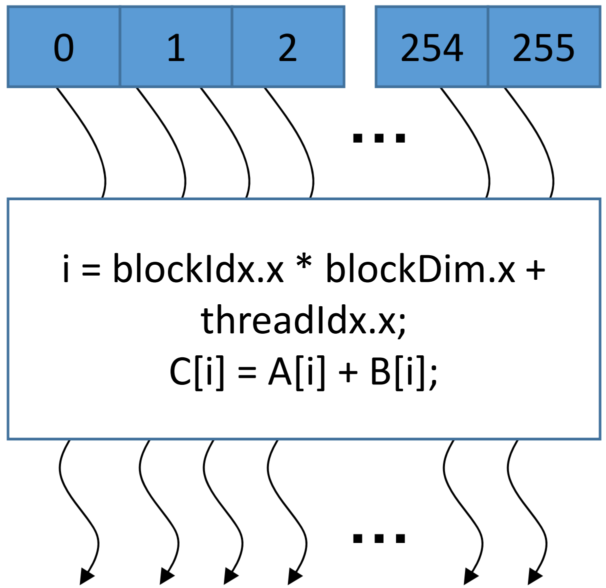
The entire grid forms the equivalent of the loop.
Each thread in the grid corresponds to one iteration of the original loop.
Sometimes also referred to as *loop parallelism*.

Thread Blocks: Scalable Cooperation

- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate

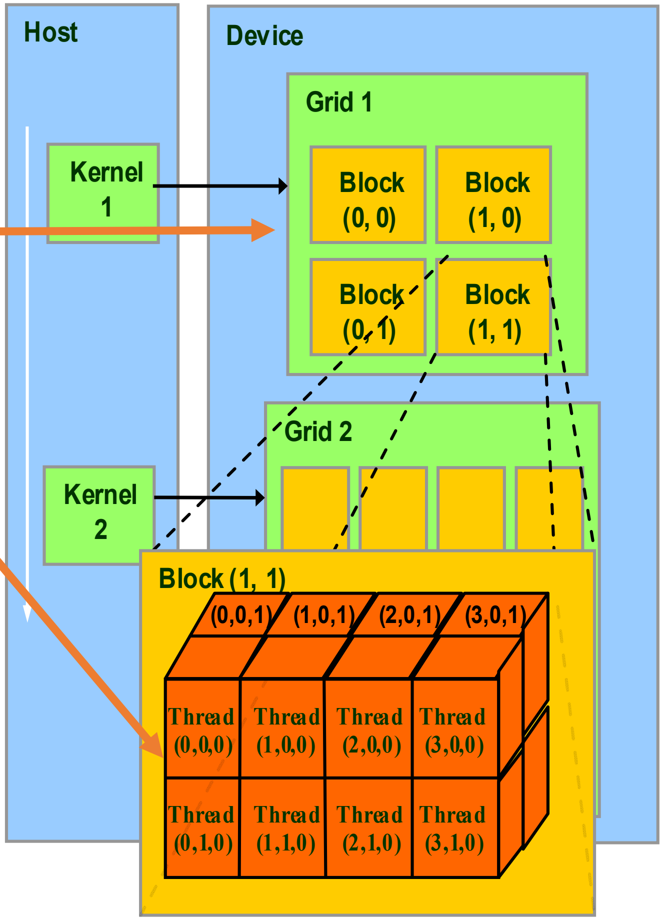
All blocks of a grid are of the same size; each block can contain up to 1024 threads;

in each dimension of thread blocks should be multiples of 32.



Thread Blocks: Scalable Cooperation

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D
- The variable is of struct type with three unsigned integer fields:
- *x*, *y*, and *z*;
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}
```

because not all vector lengths can be expressed as multiples of the block size.
For example, the vector length is 100. Assume that we picked 32 as block size.
Four thread blocks to process all the 100 vector elements.

However, the four thread blocks would have 128 threads.
We need to disable the last 28 threads in thread block 3.

Example: Vector Addition Kernel

When the host code launches a kernel, it sets the grid and thread block dimensions via *execution configuration parameters*.

The first configuration parameter gives the number of thread blocks in the grid. The second specifies the number of threads in each thread block.

```
int vectAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
}
```

Host Code

Content

- Data Parallelism
  - Example – color to grey conversion
- <u>CUDA Programming model</u>
  - CUDA C Structure
  - Example – vector add
  - Memory and Data Transfer
  - Kernel Function
  - <u>Kernel Launch</u>

A complete version of the host

```
void vecAdd(float* A, float* B, float* C, int n)
{

  int size = n * sizeof(float);
  float *d_A, *d_B, *d_C;

  cudaMalloc((void **) &d_A, size);
  cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
  cudaMalloc((void **) &d_B, size);
  cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

  cudaMalloc((void **) &d_C, size);

  vecAddKernel<<<ceil(n/256.0), 256>>>(d_A, d_B, d_C, n);

  cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

        // Free device memory for A, B, C
  cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

More on Kernel Launch

```
int vecAdd(float* A, float* B, float* C, int n)
{
 // A_d, B_d, C_d allocations and copies omitted
 // Run ceil(n/256) blocks of 256 threads each
  dim3 DimGrid(n/256, 1, 1);
  if (n%256) DimGrid.x++;
  dim3 DimBlock(256, 1, 1);

  vecAddKernel<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);
}
```

Host Code

More on Thread Blocks

- The number of thread blocks used depends on the length of the vectors (*n*):
  - If *n* is 750, three thread blocks will be used.
  - If *n* is 4000, 16 thread blocks will be used.
  - If *n* is 2,000,000, 7813 blocks will be used.
- All the thread blocks operate on different parts of the vectors. They can be executed in any arbitrary order.
- Scalability in execution speed with hardware:
  - same code runs at lower speed on small GPUs and higher speed on larger GPUs

Summary

- Data Parallelism
  - Example – color to grey conversion
- CUDA Programming model
  - CUDA C Structure
  - Example – vector add
  - Memory and Data Transfer
  - Kernel Function
  - Kernel Launch