Introduction to Parallel Programming

Pablo Brubeck

Department of Physics
Tecnologico de Monterrey

October 14, 2016

Outline

The basics
- Data types
- Pointers

Parallel programming
- Hardware
- Parallel primitives

The CUDA programming model
- Thread Hierarchy
- Memory Hierarchy
- Simple code example

Outline

The basics
- Data types
- Pointers

Parallel programming
- Hardware
- Parallel primitives

The CUDA programming model
- Thread Hierarchy
- Memory Hierarchy
- Simple code example

Hello World!

This code simply prints “Hello World!”.

```
#include <stdlib.h>
#include <stdio.h>

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

C++ has explicit data types.

```
bool isEmpty = true;

int particles = 1024;

float Tf = 273.15;

double phi = 1.61803398875;
```

Structs are complex data types.

We can define the 3D vector data type.

```
struct float3{
    float x, y, z;
};

float3 pos;
pos.x = 3.2;
pos.y = -4.7;
pos.z = 1.34;
```

A pointer is a variable that stores the memory address of another variable.

- When a variable is declared, the memory needed to store its value is assigned to a specific location.

A pointer is a variable that stores the memory address of another variable.

- When a variable is declared, the memory needed to store its value is assigned to a specific location.
- The program does not explicitly decide the exact memory location.

A pointer is a variable that stores the memory address of another variable.

- When a variable is declared, the memory needed to store its value is assigned to a specific location.
- The program does not explicitly decide the exact memory location.
- A pointer is used to access data cells that are at a certain postion relative to it.

Pointers may be obtained from values and viceversa.

- & is the address-of operator, and can be read simply as "address of"

Pointers may be obtained from values and viceversa.

- & is the address-of operator, and can be read simply as "address of"
- * is the dereference operator, and can be read as "value pointed to by"

Pointers may be obtained from values and viceversa.

- & is the address-of operator, and can be read simply as "address of"
- * is the dereference operator, and can be read as "value pointed to by"

Pointers may be obtained from values and viceversa.

- & is the address-of operator, and can be read simply as "address of"
- * is the dereference operator, and can be read as "value pointed to by"

```
float val;  \\ val is declared as a variable
float *ptr; \\ ptr is declared as a pointer

val=0.577;
*ptr=0.577;

printf("Address␣of␣val␣%d\n", &val);
printf("Address␣of␣*ptr␣%d\n", ptr);
```

An array may be declared with a pointer.

```
int N = 1024;
double *foo = new double[N];

for(int i=0; i<N; i++){
    foo[i]=cos(pi/N*i));
}
```

An array may be declared with a pointer.

```
int N = 1024;
double *foo = new double[N];

for(int i=0; i<N; i++){
    foo[i]=cos(pi/N*i));
}
```

The pointer `foo` refers to the memory address of the first element `foo[0]`.

An array may be declared with a pointer.

```
int N = 1024;
double *foo = new double[N];

for(int i=0; i<N; i++){
    foo[i]=cos(pi/N*i));
}
```

The pointer `foo` refers to the memory address of the first element `foo[0]`.

The pointer `foo+k`, where `k` is an integer, refers to the memory address of the k-th element `foo[k]`.

Matrices are square-shaped arrays.

A matrix of size $m \times n$ may be indexed in column-major storage

```
int m=128, n=256;
double *A = new double[m*n];
```

Matrices are square-shaped arrays.

A matrix of size $m \times n$ may be indexed in column-major storage

```
int m=128, n=256;
double *A = new double[m*n];

for(int i=0; i<m; i++){
    for(int j=0; j<n; j++){
        A[j*m+i]=cos(pi/m*i*j);
    }
}
```

Outline

The basics
- Data types
- Pointers

Parallel programming
- Hardware
- Parallel primitives

The CUDA programming model
- Thread Hierarchy
- Memory Hierarchy
- Simple code example

Why parallelism?

If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?

Seymour Cray, the father of supercomputing.

Parallelization is desired to increase the throughput.

Throughput = Number of computations / Amount of time.

Parallelization is desired to increase the throughput.

Throughput = Number of computations / Amount of time.

Throughput is measured in **FL**oating-**P**oint **O**perations **P**er **S**econd or **flops**.

The CPU is excellent for serial tasks.

- The CPU is in charge of coordinating hardware and software.

The CPU is excellent for serial tasks.

- The CPU is in charge of coordinating hardware and software.
- Typically, it may have from 2 to 12 processing cores.

The CPU is excellent for serial tasks.

- The CPU is in charge of coordinating hardware and software.
- Typically, it may have from 2 to 12 processing cores.
- Many programs run entirely on the CPU.

The GPU is excellent for parallel tasks.

- The GPU is in charge of processing and rendering graphics onto your display.

The GPU is excellent for parallel tasks.

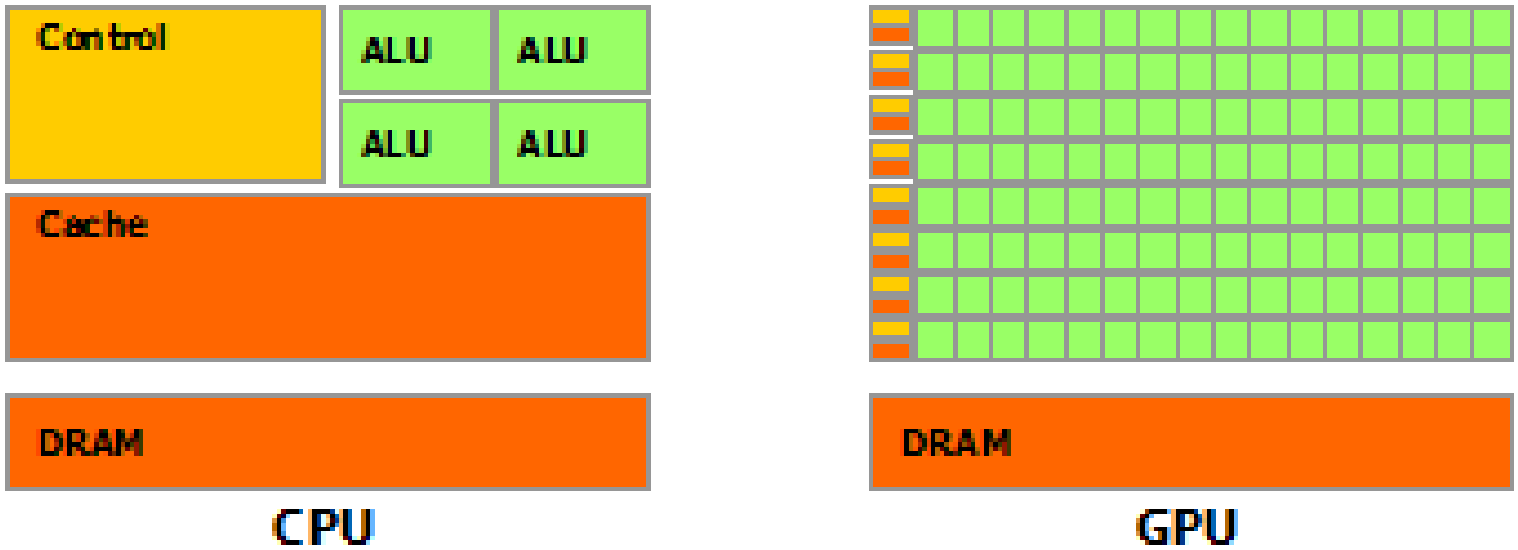
- The GPU is in charge of processing and rendering graphics onto your display.
- Typically, it may have from hundreds to thousands of processing cores.

The GPU is excellent for parallel tasks.

- The GPU is in charge of processing and rendering graphics onto your display.
- Typically, it may have from hundreds to thousands of processing cores.
- General Purpose GPU Programming takes advantage of this enormous computing power.

The GPU devotes more transistors to data processing.

GPUs have many more Arithmetic Processing Units (ALUs).



Parallel primitives are common, highly parallelizable operations

- Mapping
- Stencil
- Reduction
- Scan or prefix sum

Mapping

A mapping is a transformation of N inputs into N outputs where the same **unary function** is applied **independently** to every array element.

Mapping

A mapping is a transformation of N inputs into N outputs where the same **unary function** is applied **independently** to every array element.

$$y_i = f(x_i), \qquad i = 0, 1, \ldots, N-1.$$

Mapping

A mapping is a transformation of N inputs into N outputs where the same **unary function** is applied **independently** to every array element.

$$y_i = f(x_i), \qquad i = 0, 1, \ldots, N-1.$$

- Compute the square of each array element.

Mapping

A mapping is a transformation of N inputs into N outputs where the same **unary function** is applied **independently** to every array element.

$$y_i = f(x_i), \qquad i = 0, 1, \ldots, N-1.$$

- Compute the square of each array element.
- Compute the color of each pixel on an image.

Mapping

A mapping is a transformation of N inputs into N outputs where the same **unary function** is applied **independently** to every array element.

$$y_i = f(x_i), \qquad i = 0, 1, \ldots, N-1.$$

- Compute the square of each array element.
- Compute the color of each pixel on an image.
- Transform each vertex of a 3D model.

Stencil

A stencil operation is a transformation of N inputs into N outputs where the operator takes some of the neighboring elements.

Stencil

A stencil operation is a transformation of N inputs into N outputs where the operator takes some of the neighboring elements.

$$y_i = f(\ldots, x_{i-1}, x_i, x_{i+1}, \ldots), \qquad i = 0, 1, \ldots, N-1.$$

Stencil

A stencil operation is a transformation of N inputs into N outputs where the operator takes some of the neighboring elements.

$$y_i = f(\ldots, x_{i-1}, x_i, x_{i+1}, \ldots), \qquad i = 0, 1, \ldots, N-1.$$

- Finite differences.

Stencil

A stencil operation is a transformation of N inputs into N outputs where the operator takes some of the neighboring elements.

$$y_i = f(\ldots, x_{i-1}, x_i, x_{i+1}, \ldots), \qquad i = 0, 1, \ldots, N-1.$$

- Finite differences.
- Image filtering.

Stencil

A stencil operation is a transformation of N inputs into N outputs where the operator takes some of the neighboring elements.

$$y_i = f(\ldots, x_{i-1}, x_i, x_{i+1}, \ldots), \qquad i = 0, 1, \ldots, N-1.$$

- Finite differences.
- Image filtering.
- Convolutional Neural Networks.

Reduction

A reduction operation takes N inputs to produce a single output with an **associative binary operator**.

Reduction

A reduction operation takes N inputs to produce a single output with an **associative binary operator**.

$$y = x_0 \oplus x_1 \oplus \ldots \oplus x_{N-1}$$

Reduction

A reduction operation takes N inputs to produce a single output with an **associative binary operator**.

$$y = x_0 \oplus x_1 \oplus \ldots \oplus x_{N-1}$$

- Find the total sum of an array.

Reduction

A reduction operation takes N inputs to produce a single output with an **associative binary operator**.

$$y = x_0 \oplus x_1 \oplus \ldots \oplus x_{N-1}$$

- Find the total sum of an array.
- Find the maximum element of an array.

Reduction

A reduction operation takes N inputs to produce a single output with an **associative binary operator**.

$$y = x_0 \oplus x_1 \oplus \ldots \oplus x_{N-1}$$

- Find the total sum of an array.
- Find the maximum element of an array.
- Find the minimum element of an array.

Scan or prefix sum

A scan operation takes N inputs to produce N partial outputs by applying an **associative binary operator**.

Scan or prefix sum

A scan operation takes N inputs to produce N partial outputs by applying an **associative binary operator**.

$$y_i = x_0 \oplus x_1 \oplus ... \oplus x_i, \qquad i = 0, 1, \ldots, N-1$$

Scan or prefix sum

A scan operation takes N inputs to produce N partial outputs by applying an **associative binary operator**.

$$y_i = x_0 \oplus x_1 \oplus ... \oplus x_i, \qquad i = 0, 1, \ldots, N-1$$

- Find the accumulated sum of an array.

Scan or prefix sum

A scan operation takes N inputs to produce N partial outputs by applying an **associative binary operator**.

$$y_i = x_0 \oplus x_1 \oplus ... \oplus x_i, \qquad i = 0, 1, \ldots, N-1$$

- Find the accumulated sum of an array.
- Compute the histogram of a series of data.

Scan or prefix sum

A scan operation takes N inputs to produce N partial outputs by applying an **associative binary operator**.

$$y_i = x_0 \oplus x_1 \oplus ... \oplus x_i, \qquad i = 0, 1, \ldots, N-1$$

- Find the accumulated sum of an array.
- Compute the histogram of a series of data.
- Image processing, High Dynamic Range (HDR).

Parallel primitives are already implemented on libraries

Thrust is a CUDA library that allows you to implement high performance parallel applications with minimal programming effort.

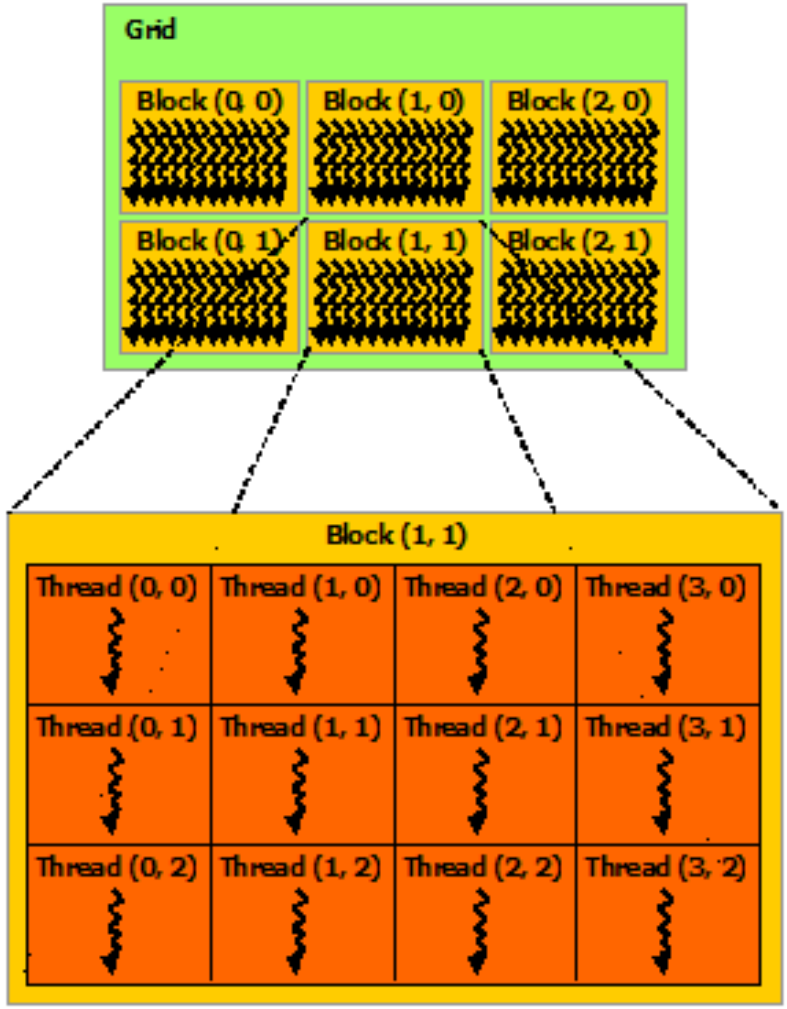
Outline

The basics
- Data types
- Pointers

Parallel programming
- Hardware
- Parallel primitives

The CUDA programming model
- Thread Hierarchy
- Memory Hierarchy
- Simple code example

Parallel code execution is organized into a grid of thread blocks.



Various threads execute serial code simultaneously.

- These pieces of serial code are known as **kernels**.

Various threads execute serial code simultaneously.

- These pieces of serial code are known as **kernels**.
- Their execution is grouped into **thread blocks**.

Various threads execute serial code simultaneously.

- These pieces of serial code are known as **kernels**.
- Their execution is grouped into **thread blocks**.
- Thread blocks are grouped into a **block grid**.

Various threads execute serial code simultaneously.

- These pieces of serial code are known as **kernels**.
- Their execution is grouped into **thread blocks**.
- Thread blocks are grouped into a **block grid**.
- Threads and blocks may have up to three-dimensional indices.

Parallelism is limited by the GPU.

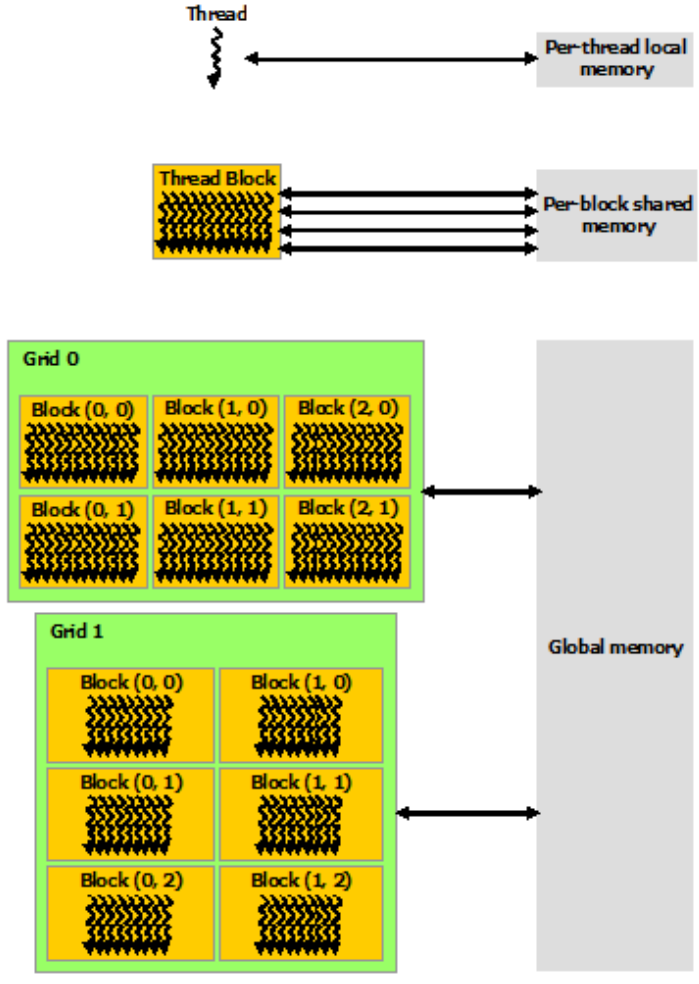
Since the whole task might require more parallel processors that the ones that are available, the task is divided into thread blocks.

Parallelism is limited by the GPU.

Since the whole task might require more parallel processors that the ones that are available, the task is divided into thread blocks.

All threads within a thread block are ensured to be synchronous.

Memory is shared at different levels.



Memory is shared at different levels.

- **Local memory** is the fastest, but it is limited to an individual thread.

Memory is shared at different levels.

- **Local memory** is the fastest, but it is limited to an individual thread.
- **Shared memory** is shared between threads of the same block.

Memory is shared at different levels.

- **Local memory** is the fastest, but it is limited to an individual thread.
- **Shared memory** is shared between threads of the same block.
- **Global memory** is the slowest, but may be accessed from any thread.

Memory allocation. Oh, no! pointers!

The CUDA unified memory model allows sharing between the host and the device. This is how to allocate global memory on the device.

```
int N=1024;
float *data;
cudaMallocManaged(&data, N*sizeof(float));
```

Kernel declaration. Oh, no! weird syntax!

```
// Generate equally spaced vector from a to b

__global__ void linspace(int N, float *data,
                         float a, float b){
    int i=blockIdx.x*blockDim.x+threadIdx.x;
    data[i]=a+(b-a)*i/(N-1);
}
```

Kernel declaration. Oh, no! weird syntax!

```
// Generate equally spaced vector from a to b

__global__ void linspace(int N, float *data,
                         float a, float b){
    int i=blockIdx.x*blockDim.x+threadIdx.x;
    data[i]=a+(b-a)*i/(N-1);
}

// Compute the square of each array element

__global__ void mapsquare(int N, float *data){
    int i=blockIdx.x*blockDim.x+threadIdx.x;
    data[i]=data[i]*data[i];
}
```

Kernel invocation. Oh, no! even weirder syntax!

```
// Define grid and block dimensions
dim3 block(512);
dim3 grid((N+512-1)/512);

// Kernel invocation
linspace<<<grid, block>>>(N, data, -10, 10);

mapsquare<<<grid, block>>>(N, data);
```

We forgot our data on the device,
we must take it back to the host.

```
// Must do this before passing data
// from device to host
cudaDeviceSynchronize();

// Print the result
for(int i=0; i<N; i++){
    printf("%f\n", data[i]);
}
```

Conclusion

Efficient parallel programming may seem very complicated at first, but its benefits are truly worth the effort.

Conclusion

Efficient parallel programming may seem very complicated at first, but its benefits are truly worth the effort.

If you really liked it, enroll to the Introduction to Parallel Programming Course on Udacity.

References


"Pointers - C++ Tutorials." http://www.cplusplus.com/doc/tutorial/pointers/. (Accessed on 10/11/2016).


"Unified memory in cuda 6 — parallel forall." https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/. (Accessed on 10/13/2016).


"Programming guide :: Cuda toolkit documentation." https://docs.nvidia.com/cuda/cuda-c-programming-guide/. (Accessed on 10/13/2016).