General-purpose programming on GPU

Launching kernels

Eugenio Rustico
rustico@dmi.unict.it

D.M.I. - Università di Catania

Updated: May 19, 2011
Overview:

- Recall: launching kernels
- Further step: 2D grids
- Bottom-up approach
- Interact, any time!

Requisites:

- Mastery of C/C++
- Coffee!

Web: http://www.dmi.unict.it/~bilotta/gpgpu/
Overview

1 Introduction

2 Dive in: array initialization
   - Constant init
   - Non-constant init
   - Function init
   - Threaded init
   - CUDA init

3 Launch granularity: blocks
   - 1D blocks
   - 2D blocks

4 Hands on code
Let’s start with a trivial example: initializing an array.
Let’s start with a trivial example: initializing an array.

Allocate space:

```c
1  unsigned int vector_size = 10;
2  hVector = (int*) malloc(vector_size*sizeof(int));
```
Let’s start with a trivial example: initializing an array.

Allocate space:

```c
1 unsigned int vector_size = 10;
2 hVector = (int*) malloc(vector_size*sizeof(int));
```

Iterate on elements:

```c
3 for (int i=0; i<vector_size; i++)
4    hVector[i] = 0;
```
Let’s start with a trivial example: initializing an array.

Allocate space:

```c
1 unsigned int vector_size = 10;
2 hVector = (int*) malloc(vector_size*sizeof(int));
```

Use memset:

```c
3 // must include <string.h>
4 memset(hVector, 0, vector_size*sizeof(int));
```
Initialization with non-constant value

Allocate space:

1. `unsigned int vector_size = 10;`
2. `hVector = (int*) malloc(vector_size*sizeof(int));`

Iterate on elements (cannot use memset):

3. `for (int i=0; i<vector_size; i++)`
4. `hVector[i] = i;`
Same example, with body of for cycle replaced by a function

```c
void set_array_element(int* array, 
    unsigned int element)
{
    array[element] = element;
}
```

Eugenio Rustico  
rustico@dmi.unict.it  
Università di Catania
Same example, with body of for cycle replaced by a function

```c
void set_array_element( int* array, \
    unsigned int element)
{
    array[element] = element;
}

hVector = (int*) malloc(vector_size*sizeof(int));
for (int i=0; i<vector_size; i++)
    set_array_element(hVector, i);
//hVector[i] = i;
```
With threads: iterations of for cycle are now parallel (in theory)

```
static void set_array_element(...) {
    array[element] = element;
}

hVector = (int*) malloc(vector_size*sizeof(int));
for (int i=0; i<vector_size; i++)
    pthread_create(&threadID, NULL, \
    set_array_element, (...));
    //hVector[i] = i;
```
With threads: iterations of for cycle are now parallel (in theory)

```c
static void set_array_element(...)
{
    array[element] = element;
}
```

```c
hVector = (int*) malloc(vector_size*sizeof(int));
for (int i=0; i<vector_size; i++)
    pthread_create(&threadID, NULL, \n        set_array_element, (...));
//hVector[i] = i;
```

**Note**: actually, pthreads can take only one parameter (e.g. pointer to class or struct).
Same initialization with CUDA:

```c
__global__ void set_array_element( int* array)
{
    array[threadIdx.x] = threadIdx.x;
}

cudaMalloc(&dVector, vector_size*sizeof(int));
//for (int i=0; i<vector_size; i++)
set_array_element<<<1, vector_size>>>(dVector);
//hVector[i] = i;
```
Same initialization with CUDA:

```c
__global__ void set_array_element(int* array)
{
    array[threadIdx.x] = threadIdx.x;
}

cudaMalloc(&dVector, vector_size*sizeof(int));
// for (int i=0; i<vector_size; i++)
set_array_element<<<1, vector_size>>>(dVector);
    // hVector[i] = i;

// download array to access data
cudaMemcpy(hVector, dVector, 
   vector_size*sizeof(int), 
   cudaMemcpyDeviceToHost);
```
void set_array_element( int* array, uint element) {
    array[element] = element;
}

... 

hVector = (int*) malloc(vector_size*sizeof(int));
for (int i=0; i<vector_size; i++)
    set_array_element(hVector, i);

__global__ void set_array_element( int* array) {
    array[threadIdx.x] = threadIdx.x;
}

... 

cudaMalloc(&dVector, vector_size*sizeof(int));
//for (int i=0; i<vector_size; i++)
set_array_element<<<1, vector_size>>>(dVector);

Find the differences

Eugenio Rustico rustico@dmi.unict.it Università di Catania
General-purpose programming on GPU
__global__ void set_array_element(int* array) {
    array[threadIdx.x] = threadIdx.x;
}

cudaMalloc(&dVector, vector_size*sizeof(int));
set_array_element<<<1, vector_size>>>(dVector);
Dive in: array initialization
Launch granularity: blocks
Hands on code

CUDA init

```c
__global__ void set_array_element( int* array )
{
    array[threadIdx.x] = threadIdx.x;
}

... 
cudaMalloc(&dVector, vector_size*sizeof(int));
set_array_element<<<1, vector_size>>>(dVector);
```

- threadIdx is *implicit* (it is a dim3: has x, y, and z fields)
Introduction

Dive in: array initialization

Launch granularity: blocks

Hands on code

CUDA init

```c
__global__ void set_array_element(int* array)
{
    array[threadIdx.x] = threadIdx.x;
}
```

```
cudaMalloc(&dVector, vector_size * sizeof(int));
set_array_element<<<1, vector_size>>>(dVector);
```

- `threadIdx` is *implicit* (it is a `dim3`: has x, y, and z fields)
- `__global__` and `void` are *mandatory* for kernels

Eugenio Rustico rustico@dmi.unict.it Università di Catania

General-purpose programming on GPU
```c
__global__ void set_array_element(int * array)
{
    array[threadIdx.x] = threadIdx.x;
}

... cudaMalloc(&dVector, vector_size*sizeof(int));
set_array_element<<<1, vector_size>>>(dVector);
```

- threadIdx is **implicit** (it is a dim3: has x, y, and z fields)
- __global__ and void are **mandatory** for kernels
- Non-standard syntax: compile with `nvcc`
Cuda init

```c
__global__ void set_array_element ( int* array )
{
    array[threadIdx.x] = threadIdx.x;
}
...
cudaMalloc (&dVector, vector_size*sizeof(int));
set_array_element <<<1, vector_size>>>(dVector);
```

- `threadIdx` is *implicit* (it is a `dim3`: has x, y, and z fields)
- `__global__` and `void` are *mandatory* for kernels
- Non-standard syntax: compile with `nvcc`
- `hVector` and `dVector` belong to separate *address spaces*:
Introduction

Dive in: array initialization

Launch granularity: blocks

Hands on code

CUDA init

```c
__global__ void set_array_element( int * array )
{
    array[threadIdx.x] = threadIdx.x;
}
...
cudaMalloc(&dVector, vector_size*sizeof(int));
set_array_element<<<1, vector_size>>>(dVector);
```

- `threadIdx` is *implicit* (it is a `dim3`: has x, y, and z fields)
- `__global__` and `void` are *mandatory* for kernels
- Non-standard syntax: compile with `nvcc`
- `hVector` and `dVector` belong to separate *address spaces*:
  - `hVector` is allocated with `malloc()`, `dVector` with `cudaMalloc()`
__global__ void set_array_element ( int* array )
{
    array [ threadIdx .x ] = threadIdx .x ;
}

... 
cudaMalloc ( & dVector , vector_size * sizeof ( int ));
set_array_element <<<1, vector_size >>> ( dVector );

- threadIdx is *implicit* (it is a dim3: has x, y, and z fields)
- __global__ and void are *mandatory* for kernels
- Non-standard syntax: compile with nvcc
- hVector and dVector belong to separate *address spaces*:
  - hVector is allocated with malloc(), dVector with cudaMemcpy()
  - we need to copy back the array as soon as we need to access it ("download")
Blocks

- Kernel launches are grouped in *blocks*, reflecting the hardware architecture
Blocks

- Kernel launches are grouped in *blocks*, reflecting the hardware architecture
- Each block is assigned to a multiprocessor
Blocks

- Kernel launches are grouped in *blocks*, reflecting the hardware architecture
- Each block is assigned to a multiprocessor
- The block size is up to the programmer (we will discuss it later)
**Blocks**

- Kernel launches are grouped in *blocks*, reflecting the hardware architecture
- Each block is assigned to a multiprocessor
- The block size is up to the programmer (we will discuss it later)
- Thread indexing becomes two-level
blockDim.x = 10  gridDim.x = 1
blockDim.y =  1  gridDim.y = 1
blockDim.z =  1  gridDim.z = 1

kernel grid

blockIdx.x = 0

tIdx.x = 0  tIdx.x = 1  tIdx.x = 2  tIdx.x = 3  tIdx.x = 4  tIdx.x = 5  tIdx.x = 6  tIdx.x = 7  tIdx.x = 8  tIdx.x = 9

dVector

0 1 2 3 4 5 6 7 8 9
Dive in: array initialization

Launch granularity: blocks

Hands on code

1D blocks

blockDim.x = 10  gridDim.x = 1
blockDim.y = 1  gridDim.y = 1
blockDim.z = 1  gridDim.z = 1

kernel grid

blockIdx.x = 0

int myidx = threadIdx.x;
dVector[myidx] = ... ;
1D blocks

Introduction

Dive in: array initialization

Launch granularity: blocks

Hands on code

```
blockDim.x = 10  blockDim.y = 1  blockDim.z = 1
gridDim.x = 1   gridDim.y = 1   gridDim.z = 1
```

```
blockIdx.x = 0
```

```
0 1 2 3 4 5 6 7 8 ...
blockDim.x = 10
blockDim.y = 1
blockDim.z = 1
gridDim.x = 1
gridDim.y = 1
gridDim.z = 1
kernel grid
```

```
Kernel call:

1 kernel_name<<<1, vector_size>>>(dVector);
```
Kernel call:

1. `kernel_name<<<1, vector_size>>>(dVector);`

Inside kernel:

1. `int myidx = threadIdx.x;`
2. `dVector[myidx] = ... ;`
Dive in: array initialization
Launch granularity: blocks
Hands on code

1D blocks

blockDim.x = 5  blockDim.y = 1  blockDim.z = 1
gridDim.x = 2  gridDim.y = 1  gridDim.z = 1

dVector

kernel grid

blockIdx.x = 0

tIdx.x = 0  tIdx.x = 1  tIdx.x = 2  tIdx.x = 3  tIdx.x = 4

blockIdx.x = 1

tIdx.x = 0  tIdx.x = 1  tIdx.x = 2  tIdx.x = 3  tIdx.x = 4

Eugenio Rustico  rustico@dmi.unict.it  Università di Catania
General-purpose programming on GPU
1D blocks

blockDim.x = 5  blockDim.y = 1  blockDim.z = 1
gridDim.x = 2  gridDim.y = 1  gridDim.z = 1

Kernel call:

```
1
kernel_name <<<2, vector_size /2 >>>( dVector );
```

Inside kernel:

```
1
2
int myidx = blockDim.x* blockIdx.x + threadIdx.x;
dVector[ myidx ] = ...;
```
1D blocks

Kernel call:

```c
1 kernel_name <<2, vector_size/2>>>(dVector);
```
Kernel call:

1. `kernel_name<<<2, vector_size/2>>>(dVector);`

Inside kernel:

1. `int myidx = blockDim.x*blockIdx.x + threadIdx.x;`
2. `dVector[myidx] = ...;`
Kernel launch syntax:

```cpp
kernel_name<<<dimGrid, dimBlock>>>(...);
```

Where
Kernel launch syntax:

```c
kernel_name<<<dimGrid, dimBlock>>>(...);
```

Where

- `dimBlock` and `dimGrid` are `dim3`
Kernel launch syntax:

```cpp
kernel_name <<<dimGrid, dimBlock>>>(...);
```

Where

- `dimBlock` and `dimGrid` are `dim3`
- `dimBlock` is 1D, 2D or 3D, `dimGrid` is 1D or 2D (up to CUDA 3.2)
Kernel launch syntax:

```
kernel_name <<<dimGrid, dimBlock>>>(...);
```

Where

- `dimBlock` and `dimGrid` are `dim3`
- `dimBlock` is 1D, 2D or 3D, `dimGrid` is 1D or 2D (up to CUDA 3.2)
- specifying an int is equivalent to a 1D `dim3`
Kernel launch syntax:

```c
kernel_name <<<dimGrid, dimBlock>>>(...);
```

Where

- `dimBlock` and `dimGrid` are `dim3`
- `dimBlock` is 1D, 2D or 3D, `dimGrid` is 1D or 2D (up to CUDA 3.2)
- specifying an int is equivalent to a 1D `dim3`
- maximum `dimBlock` is smaller than maximum `gridDim`, see `deviceQuery`
What if `dimBlock` does not multiply `dimGrid`?
1D blocks

What if \( \text{dimBlock} \) does not multiply \( \text{dimGrid} \)?

- Launch sufficient number of blocks to cover all data (round up)
Dive in: array initialization

Launch granularity: blocks

1D blocks

kernel_name<<<dimGrid, dimBlock>>>(...);

What if $\text{dimBlock}$ does not multiply $\text{dimGrid}$?

- Launch sufficient number of blocks to cover all data (round up)
- Inside kernel, check address before accessing

```c
int vector_size = 101;
int dimBlock = 10;
int dimGrid = \\
(int) ceil(vector_size/(float)dimBlock);
kernel<<<dimGrid, dimBlock>>>(uint vec_size,...);
```
1D blocks

kernel_name << dimGrid, dimBlock >> (...);

What if dimBlock does not multiply dimGrid?

- Launch sufficient number of blocks to cover all data (round up)
- Inside kernel, check address before accessing

```
int vector_size = 101;
int dimBlock = 10;
int dimGrid = (int) ceil(vector_size / (float) dimBlock);
kernel << dimGrid, dimBlock >> (uint vec_size, ...);
```

Inside kernel:

```
int myidx = blockDim.x * blockIdx.x + threadIdx.x;
if (myidx >= vec_size) return;
dVector[myidx] = ...;
```
Introduction

Dive in: array initialization

Launch granularity: blocks

Hands on code

2D blocks

Thread 0,0  Thread 1,0  Thread 2,0

**BLOCK 0,0**

Thread 0,1  Thread 1,1  Thread 2,1

Thread 0,0  Thread 1,0  Thread 2,0

**BLOCK 0,1**

Thread 0,1  Thread 1,1  Thread 2,1

Thread 0,1  Thread 1,1  Thread 2,1

**BLOCK 0,2**

Thread 0,0  Thread 1,0  Thread 2,0

Thread 0,0  Thread 1,0  Thread 2,0

**BLOCK 1,0**

Thread 0,1  Thread 1,1  Thread 2,1

Thread 0,1  Thread 1,1  Thread 2,1

**BLOCK 1,1**

Thread 0,1  Thread 1,1  Thread 2,1

Thread 0,1  Thread 1,1  Thread 2,1

**BLOCK 1,2**

Thread 0,0  Thread 1,0  Thread 2,0

Thread 0,0  Thread 1,0  Thread 2,0

Thread 0,1  Thread 1,1  Thread 2,1

Thread 0,1  Thread 1,1  Thread 2,1

Eugenio Rustico  rustico@dmi.unict.it

General-purpose programming on GPU
Introduction

Dive in: array initialization

Launch granularity: blocks

Hands on code

2D blocks

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

threadIdx.y = blockIdx.y * blockDim.y + threadIdx.y

blockSize.x * blockIdx.x + threadIdx.x

blockSize.y * blockIdx.y + threadIdx.y
2D blocks: same addressing on both X and Y

1. `dim3 vector_size(15,10,1);`
2. `dim3 dimBlock(10,10,1);`
3. `dim3 dimGrid(`
4. `  vector_size.x / dimBlock.x),`
5. `  vector_size.y / dimBlock.y),`
6. `  1);`
7. `kernel <<< dimGrid, dimBlock >>>(dim3 vec_size,...);`
2D blocks: same addressing on both X and Y

```cpp
1  dim3  vector_size(15,10,1);
2  dim3  dimBlock(10,10,1);
3  dim3  dimGrid( \n4    vector_size.x / dimBlock.x),
5    vector_size.y / dimBlock.y),
6    1);
7  kernel<<<dimGrid, dimBlock>>>(dim3 vec_size,...);
```

Inside kernel:

```cpp
8  int  x_coord = blockDim.x*blockIdx.x + threadIdx.x;
9  int  y_coord = blockDim.y*blockIdx.y + threadIdx.y;
10  dVector[x_coord][y_coord] = ... ;
```
...and same checks: what if does not multiply?
...and same checks: what if does not multiply?

```c
1  dim3  vector_size(150,100,1);
2  dim3  dimBlock(16,16,1);
3  dim3  dimGrid(  
4       (int) ceil(vector_size.x/(float)dimBlock.x),
5       (int) ceil(vector_size.y/(float)dimBlock.y),
6       1);
7  kernel<<dimGrid, dimBlock>>>(dim3  vec_size,...);
```
...and same checks: what if does not multiply?

```
1. dim3 vector_size(150,100,1);
2. dim3 dimBlock(16,16,1);
3. dim3 dimGrid( \
   (int) ceil(vector_size.x/(float)dimBlock.x),
   (int) ceil(vector_size.y/(float)dimBlock.y),
   1);
4. kernel <<< dimGrid, dimBlock >>>(dim3 vec_size,...);
```

Inside kernel:

```
8. int x_coord = blockDim.x*blockIdx.x + threadIdx.x;
9. if (x_coord >= vec_size.x) return;
10. int y_coord = blockDim.y*blockIdx.y + threadIdx.y;
11. if (y_coord >= vec_size.y) return;
12. dVector[x_coord][y_coord] = ... ;
```
Hands on code:

- GPU vector initialization
- GPU vector initialization, $k$ elements per thread
- GPU vector average
- GPU image smoothing, variable $K$
- Minimum scan