General-purpose programming on GPU

Launching kernels

Eugenio Rustico
rustico@dmi.unict.it

D.M.I. - Università di Catania

Updated: April 9, 2011

Overview:

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

Requisites:

- Mastery of C/C++
- Coffee!

Web: `http://www.dmi.unict.it/~bilotta/gpgpu/`
Forum: `http://forum.sdai.unict.it/index.php?board=116.0`

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:

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

Let’s start with a trivial example: initializing an array.

Allocate space:

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

Iterate on elements:

```
for (int i=0; i<vector_size; i++)
    hVector[i] = 0;
```

Let's start with a trivial example: initializing an array.

Allocate space:

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

Use `memset`:

```
// must include <string.h>
memset(hVector, 0, vector_size*sizeof(int));
```

Initialization with non-constant value

Allocate space:

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

Iterate on elements (cannot use `memset`):

```
for (int i=0; i<vector_size; i++)
    hVector[i] = i;
```

Same example, with body of `for` cycle replaced by a function

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

Same example, with body of `for` cycle replaced by a function

```
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)

```
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;
```

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

Same initialization with CUDA:

```
__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:

```
__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

```
__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);
```

```
__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__ 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

```
__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`

```
__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*:

```
__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 `cudaMalloc()`
  - 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

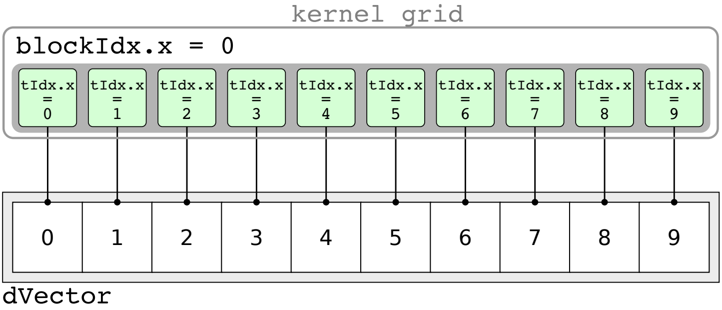
Blocks

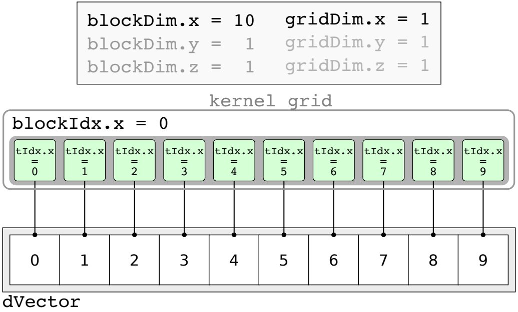
- Kernel launches are grouped in *blocks*, reflecting the hardware architecture
- Each block is assigned to a multiprocessor
- The block size is up to the programmer (we will discuss it later)

Blocks

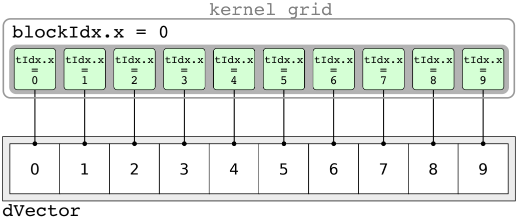
- Kernel launches are grouped in *blocks*, reflecting the hardware architecture
- Each block is assigned to a multiprocessor
- The block size is up to the programmer (we will discuss it later)
- Thread indexing becomes two-level

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



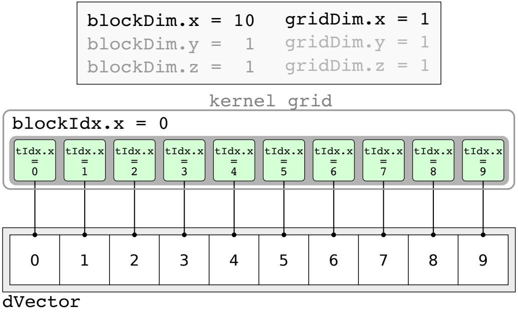

blockDim.x = 10
gridDim.x = 1
blockDim.y = 1
gridDim.y = 1
blockDim.z = 1
gridDim.z = 1
kernel grid
blockIdx.x = 0
tIdx.x = 0
tIdx.x = 1
tIdx.x = 2
tIdx.x = 3
tIdx.x = 4
tIdx.x = 5
tIdx.x = 6
tIdx.x = 7
tIdx.x = 8
tIdx.x = 9
0 1 2 3 4 5 6 7 8 9
dVector

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



Kernel call:

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



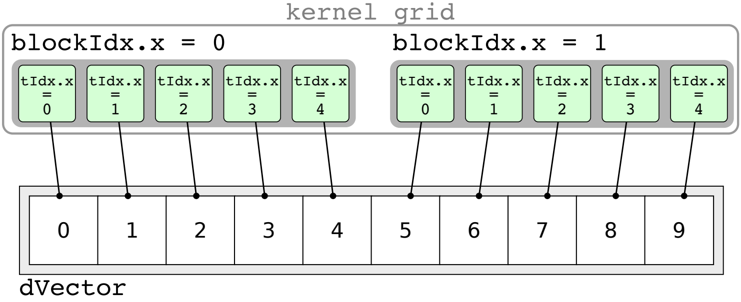
Kernel call:

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

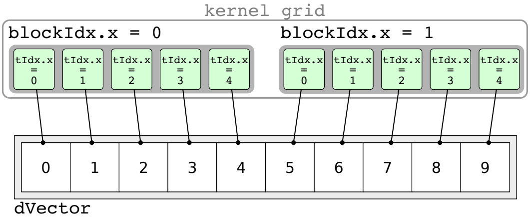
Inside kernel:

```
int myidx = threadIdx.x;
dVector[myidx] = ... ;
```

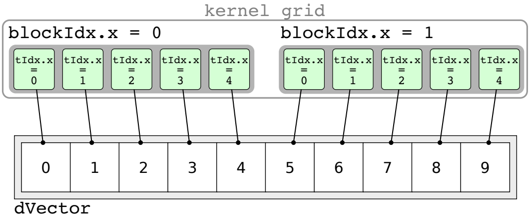
```
blockDim.x = 5   gridDim.x = 2
blockDim.y = 1   gridDim.y = 1
blockDim.z = 1   gridDim.z = 1
```



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



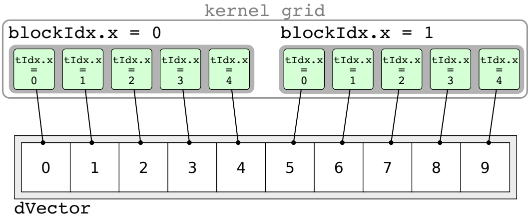
```
blockDim.x = 5   gridDim.x = 2
blockDim.y = 1   gridDim.y = 1
blockDim.z = 1   gridDim.z = 1
```



Kernel call:

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

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



Kernel call:

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

Inside kernel:

```
int myidx = blockDim.x*blockIdx.x + threadIdx.x;
dVector[myidx] = ... ;
```

Kernel launch syntax:

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

Where

Kernel launch syntax:

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

Where

- `dimBlock` and `dimGrid` are `dim3`

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)

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:

```
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 `bimBlock` is smaller than maximum `gridDim`, see `deviceQuery`

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

What if `dimBlock` does not multiply `dimGrid`?

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

What if `dimBlock` does not multiply `dimGrid`?

- Launch sufficient number of blocks to cover all data (round up)

```
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,...);
```

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

What if `dimBlock` does not multiply `dimGrid`?

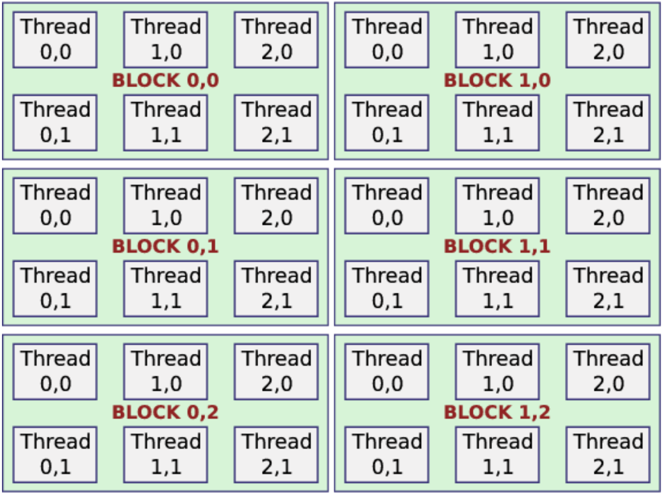
- Launch sufficient number of blocks to cover all data (round up)
- Inside kernel, check address before accessing

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

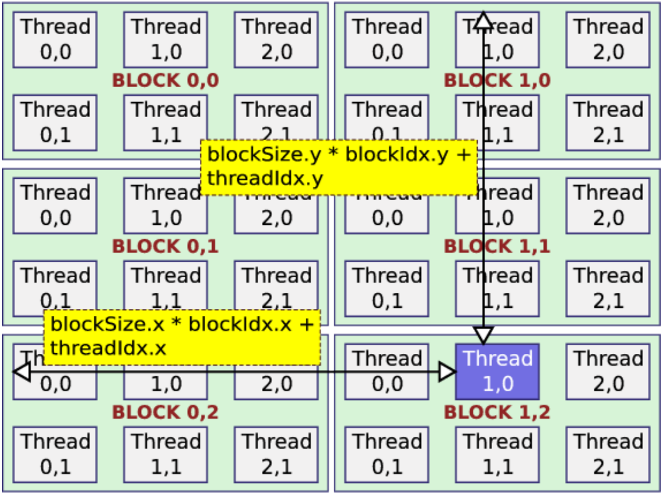
Inside kernel:

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

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

2D blocks


Thread 0,0
Thread 1,0
Thread 2,0
BLOCK 0,0
Thread 0,1
Thread 1,1
Thread 2,1
BLOCK 1,0
BLOCK 0,1
BLOCK 1,1
BLOCK 0,2
BLOCK 1,2
blockSize.y * blockIdx.y + threadIdx.y
blockSize.x * blockIdx.x + threadIdx.x

2D blocks: same addressing on both X and Y

```
dim3 vector_size(15,10,1);
dim3 dimBlock(10,10,1);
dim3 dimGrid( \
    vector_size.x / dimBlock.x),
    vector_size.y / dimBlock.y),
    1);
kernel<<<dimGrid, dimBlock>>>(dim3 vec_size,...);
```

2D blocks: same addressing on both X and Y

```
dim3 vector_size(15,10,1);
dim3 dimBlock(10,10,1);
dim3 dimGrid( \
    vector_size.x / dimBlock.x),
    vector_size.y / dimBlock.y),
    1);
kernel<<<dimGrid, dimBlock>>>(dim3 vec_size,...);
```

Inside kernel:

```
int x_coord = blockDim.x*blockIdx.x + threadIdx.x;
int y_coord = blockDim.y*blockIdx.y + threadIdx.y;
dVector[x_coord][y_coord] = ... ;
```

...and same checks: what if does not multiply?

...and same checks: what if does not multiply?

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

...and same checks: what if does not multiply?

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

Inside kernel:

```
int x_coord = blockDim.x*blockIdx.x + threadIdx.x;
if (x_coord >= vec_size.x) return;
int y_coord = blockDim.y*blockIdx.y + threadIdx.y;
if (y_coord >= vec_size.y) return;
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