# Programming in CUDA

## Part II. Interacting with the GPU from the host

Frédéric S. Masset

Instituto de Ciencias Físicas, UNAM

Lecture on CUDA

# Outline

# Outline

# Outline

# Outline

# Outline

# Reminder : allocating memory on the host

```
#define NB 1000
float *myarray;
myarray = malloc (sizeof(float) * NB);
...
myarray[5] = sqrt(3);
...
```

# Reminder : allocating memory on the host

```
#define NB 1000   vector of 1000 elements
float *myarray;
myarray = malloc (sizeof(float) * NB);
...
myarray[5] = sqrt(3);
...
```

# Reminder : allocating memory on the host

```
#define NB 1000
float *myarray; start address is myarray
myarray = malloc (sizeof(float) * NB);
...
myarray[5] = sqrt(3);
...
```

# Allocating memory on the device (GPU)

```
#define NB 1000
float *myarray;
cudaMalloc (&myarray, sizeof(float) * NB);

...
```

# Allocating memory on the device (GPU)

```
#define NB 1000  vector of 1000 elements
float *myarray;
cudaMalloc (&myarray, sizeof(float) * NB);

...
```

# Allocating memory on the device (GPU)

```
#define NB 1000
float *myarray; start address is myarray
cudaMalloc (&myarray, sizeof(float) * NB);

...
```

# Allocating memory on the device (GPU)

```
#define NB 1000
float *myarray;
cudaMalloc (&myarray, sizeof(float) * NB);
```

**floats have same size on CPU and GPU**

```
...

cudaMalloc (myarray, sizeof(float))
```

**cannot work !**

# Allocating memory on the device (GPU)

```
#define NB 1000
float *myarray;
cudaMalloc (&myarray, sizeof(float) * NB);

...
```

Note that a pointer to a float on the CPU and a pointer to a float on the GPU have same type. The compiler is **unable** to know whether a given pointer is CPU or GPU related... This is bug prone !

# CPU and GPU pointers

Dereferencing = using what is pointed to by the address represented by a pointer.

```
float *ptr;
...
*ptr = 3; pointer is dereferenced
ptr[1] = 4;
```

- Dereferencing a host pointer on the GPU will crash.
- Dereferencing a device pointer on the host will crash.

So beware ! You may use a notation that conveys information about the destination of the pointer (host / device)

# Example of notation

So beware ! You may use a notation that conveys information about the destination of the pointer (host / device)
For instance :

```
float *ptr; pointer to host memory space
```

versus

```
float *gpu_ptr; pointer to video RAM
```

It is entirely up to you.

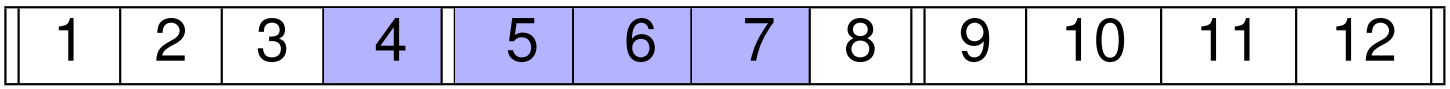
# Outline

# Importance of alignment

As we shall see later, *data alignment* is extremely important on the device memory.
What is memory alignment ? Assume that the data travels by chunks of four bytes on the bus.

# Importance of alignment

As we shall see later, *data alignment* is extremely important on the device memory.
What is memory alignment ? Assume that the data travels by chunks of four bytes on the bus.




Assume that we have a four byte array that starts at 4 : it is misaligned : reading this array (*e.g.* to the cache) requires two transactions.

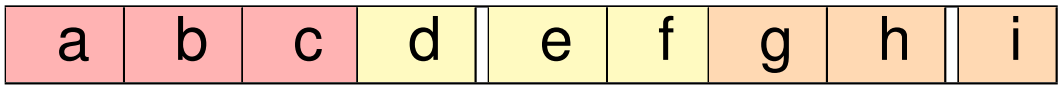
# cudaMalloc and alignment

The function `cudaMalloc ()` always takes care of alignment (the start address is properly aligned).
*An aligned memory segment is a segment with a starting address that is a multiple of some quantity, generally some integer power of 2.*
What happens if we have a two-dimensional array ?
Bad news : multi-dimensional arrays do not exist in C. You want to consider a 1D array that contains all elements instead.

Intended array

| | | |
|---|---|---|
| a | b | c |
| d | e | f |
| g | h | i |

Representation in memory




Rows are misaligned !

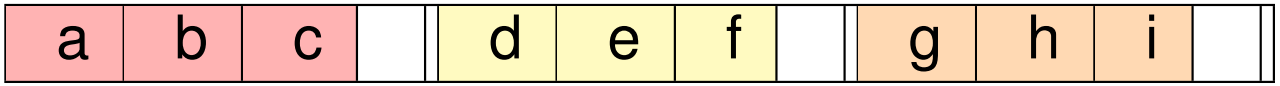
# Alignment for two-dimensional arrays

A better memory scheme would be the following:

Intended array

| | | |
|---|---|---|
| a | b | c |
| d | e | f |
| g | h | i |

Representation in memory




## Solution

We *pad* the array with extra cells (here in white), whose content is never used, so that each rows starts at an aligned address. `cudaMallocPitch ()` does that automatically for you.

# Usage of `cudaMallocPitch`

Assume you want to reserve memory for an array of `w` by `h` elements (say `floats`)

```
float *gpu_ptr;
size_t w, h, pitch;
cudaMallocPitch (&gpu_ptr, &pitch, w*sizeof(float), h);
```

```
cudaError_t cudaMallocPitch ( void **  devPtr,
                              size_t *  pitch,
                              size_t    width,
                              size_t    height
                            )
```

There also exists `cudaMalloc3D ()`.
See `examples/pitch.cu`

# Usage of `cudaMallocPitch`

Assume you want to reserve memory for an array of `w` by `h` elements (say `floats`)

```
float *gpu_ptr;
size_t w, h, pitch;
cudaMallocPitch (&gpu_ptr, &pitch, w*sizeof(float), h);
                 pointer to gpu_ptr
```

```
cudaError_t cudaMallocPitch ( void **   devPtr,
                              size_t *  pitch,
                              size_t    width,
                              size_t    height
                            )
```

There also exists `cudaMalloc3D ()`.

See `examples/pitch.cu`

# Usage of `cudaMallocPitch`

Assume you want to reserve memory for an array of `w` by `h` elements (say `floats`)

```
float *gpu_ptr;
size_t w, h, pitch;
cudaMallocPitch (&gpu_ptr, &pitch, w*sizeof(float), h);
```

**pointer to pitch**

```
cudaError_t cudaMallocPitch ( void **  devPtr,
                              size_t * pitch,
                              size_t   width,
                              size_t   height
                            )
```

There also exists `cudaMalloc3D ()`.
See `examples/pitch.cu`

# Usage of `cudaMallocPitch`

Assume you want to reserve memory for an array of `w` by `h` elements (say `floats`)

```
float *gpu_ptr;
size_t w, h, pitch;
cudaMallocPitch (&gpu_ptr, &pitch, w*sizeof(float), h);
```

**width in bytes**

```
cudaError_t cudaMallocPitch ( void **   devPtr,
                              size_t *  pitch,
                              size_t    width,
                              size_t    height
                            )
```

There also exists `cudaMalloc3D ()`.
See `examples/pitch.cu`

# Outline

# Freeing device memory upon completion

In the same manner as you must `free` memory `malloc`ated on the host, you must free device memory (video RAM) allocated on the device

```
cudaMalloc (&myarray, sizeof(float) * NB);
...
cudaFree (myarray);
```

Note that once that is done, the data referred to by `myarray` is no longer accessible, but it persists on the device and may be retrieved by others. You may want to overwrite it prior to calling `cudaFree ()`.

# Input / Outputs

- Until a few years ago, there was no possibility of input/outputs on the device
- There is now a `printf ()` equivalent... One can now do some `printf ()` debugging.
- The endpoint of a computation must be accessible (commonly written on disk).
- Data must travel as fast as possible between device and host.
- communication device $\Rightarrow$ host, then written by host in a standard manner.

# Input / Outputs

- Until a few years ago, there was no possibility of input/outputs on the device
- There is now a `printf ()` equivalent... One can now do some `printf ()` debugging.
- The endpoint of a computation must be accessible (commonly written on disk).
- Data must travel as fast as possible between device and host.
- communication device $\Rightarrow$ host, then written by host in a standard manner.

# Input / Outputs

- Until a few years ago, there was no possibility of input/outputs on the device
- There is now a `printf ()` equivalent... One can now do some `printf ()` debugging.
- The endpoint of a computation must be accessible (commonly written on disk).
- Data must travel as fast as possible between device and host.
- communication device $\Rightarrow$ host, then written by host in a standard manner.

# Input / Outputs

- Until a few years ago, there was no possibility of input/outputs on the device
- There is now a `printf ()` equivalent... One can now do some `printf ()` debugging.
- The endpoint of a computation must be accessible (commonly written on disk).
- Data must travel as fast as possible between device and host.
- communication device $\Rightarrow$ host, then written by host in a standard manner.

# Outline

# Usage of `cudaMemcpy ()`

The function `cudaMemcpy ()` is used to transfer chunks of memory from host to device, or vice-versa. It is a synchronous function : it returns only upon transfer completion.
*E.g.* transfer NB floats from device to host

```
#define NB 1000
float *ptr, *gpu_ptr;
...
cudaMemcpy (ptr,\
            gpu_ptr,\
            NB*sizeof(float),\
            cudaMemcpyDeviceToHost);
```

# Usage of `cudaMemcpy ()`

The function `cudaMemcpy ()` is used to transfer chunks of memory from host to device, or vice-versa. It is a synchronous function : it returns only upon transfer completion.
*E.g.* transfer NB floats from device to host

```
#define NB 1000
float *ptr, *gpu_ptr;
...
cudaMemcpy (ptr,\   destination first
            gpu_ptr,\
            NB*sizeof(float),\
            cudaMemcpyDeviceToHost);
```

# Usage of `cudaMemcpy ()`

The function `cudaMemcpy ()` is used to transfer chunks of memory from host to device, or vice-versa. It is a synchronous function : it returns only upon transfer completion.
*E.g.* transfer NB floats from device to host

```
#define NB 1000
float *ptr, *gpu_ptr;
...
cudaMemcpy (ptr,\
            gpu_ptr,\ then source
            NB*sizeof(float),\
            cudaMemcpyDeviceToHost);
```

# Usage of `cudaMemcpy ()`

The function `cudaMemcpy ()` is used to transfer chunks of memory from host to device, or vice-versa. It is a synchronous function : it returns only upon transfer completion.
*E.g.* transfer NB floats from device to host

```
#define NB 1000
float *ptr, *gpu_ptr;
...
cudaMemcpy (ptr,\
            gpu_ptr,\
            NB*sizeof(float),\ nb of bytes to transfer
            cudaMemcpyDeviceToHost);
```

# Usage of `cudaMemcpy ()`

The function `cudaMemcpy ()` is used to transfer chunks of memory from host to device, or vice-versa. It is a synchronous function : it returns only upon transfer completion.
*E.g.* transfer NB floats from device to host

```
#define NB 1000
float *ptr, *gpu_ptr;
...
cudaMemcpy (ptr,\
            gpu_ptr,\
            NB*sizeof(float),\
            cudaMemcpyDeviceToHost); Transfer direction
```

Remember that the compiler has no idea that `ptr` is on the host and `gpu_ptr` is on the device. You have to help it.

# Copy from host to device

Other example : assume you have initialized an array on the host, and want to perform calculations on the device.

```
#define NB 100000
float ptr[NB], *gpu_ptr;
...
Init (ptr, NB);
cudaMemcpy (gpu_ptr,\
            ptr,\
            NB*sizeof(float),\
            cudaMemcpyHostToDevice);
```

# Copy from host to device

Other example : assume you have initialized an array on the host, and want to perform calculations on the device.

```
#define NB 100000
float ptr[NB], *gpu_ptr; ptr non dynamically allocated
...
Init (ptr, NB);
cudaMemcpy (gpu_ptr,\
            ptr,\
            NB*sizeof(float),\
            cudaMemcpyHostToDevice);
```

# Copy from host to device

Other example : assume you have initialized an array on the host, and want to perform calculations on the device.

```
#define NB 100000
float ptr[NB], *gpu_ptr;
...
Init (ptr, NB);  Some initialization function on host
cudaMemcpy (gpu_ptr,\
            ptr,\
            NB*sizeof(float),\
            cudaMemcpyHostToDevice);
```

CPU code is easier to implement then GPU. If initialization is fast, it is better done on the host. Only the computationally demanding part is exported to the GPU

# Copy from host to device

Other example : assume you have initialized an array on the host, and want to perform calculations on the device.

```
#define NB 100000
float ptr[NB], *gpu_ptr;
...
Init (ptr, NB);
cudaMemcpy (gpu_ptr,\ destination first
            ptr,\
            NB*sizeof(float),\
            cudaMemcpyHostToDevice);
```

CPU code is easier to implement then GPU. If initialization is fast, it is better done on the host. Only the computationally demanding part is exported to the GPU

# Copy from host to device

Other example : assume you have initialized an array on the host, and want to perform calculations on the device.

```
#define NB 100000
float ptr[NB], *gpu_ptr;
...
Init (ptr, NB);
cudaMemcpy (gpu_ptr,\
            ptr,\  then source
            NB*sizeof(float),\
            cudaMemcpyHostToDevice);
```

CPU code is easier to implement then GPU. If initialization is fast, it is better done on the host. Only the computationally demanding part is exported to the GPU

# Copy from host to device

Other example : assume you have initialized an array on the host, and want to perform calculations on the device.

```
#define NB 100000
float ptr[NB], *gpu_ptr;
...
Init (ptr, NB);
cudaMemcpy (gpu_ptr,\
            ptr,\
            NB*sizeof(float),\  nb of bytes
            cudaMemcpyHostToDevice);
```

CPU code is easier to implement then GPU. If initialization is fast, it is better done on the host. Only the computationally demanding part is exported to the GPU

# Copy from host to device

Other example : assume you have initialized an array on the host, and want to perform calculations on the device.

```
#define NB 100000
float ptr[NB], *gpu_ptr;
...
Init (ptr, NB);
cudaMemcpy (gpu_ptr,\
            ptr,\
            NB*sizeof(float),\
            cudaMemcpyHostToDevice);   transfer direction
```

CPU code is easier to implement then GPU. If initialization is fast, it is better done on the host. Only the computationally demanding part is exported to the GPU

# A bits of hands on practice

## Bandwidth measurement

Write a piece of code that evaluates the data transfer rate between host and device.

We need:

- big arrays allocated on host and device
- a chronometer

Use the file `bandwidth.cu` provided in this lecture. Look up comments that say *UP TO YOU* and add the instructions requested.
Then compile it and run it:

```
nvcc bandwidth.cu
./a.out
```

# Playing with `bandwidth.cu`

The original executable gives the host $\Rightarrow$ device bandwidth. Modify it to check also the device $\Rightarrow$ host and device $\Rightarrow$ device bandwidths.

## Conclusions

Compare with other groups' results. How does the bandwidth vary for low-end vs high-end GPUs ?

# Outline

# Use of `cudaMallocHost ()`

- Normally, the memory allocated on the host (via `malloc`) is "mobile", *i.e.* it can go to the swap, then go to other segments of the host RAM. It is called *pageable memory*.
- CUDA provides the possibility to allocate non-pageable memory (pinned memory). It resides in a given part of the RAM and *does not move* until it is freed. That speeds up transfers with the host.
- Only root can normally do that. Beware of the use of excessive non-pageable memory $\Rightarrow$ risk of decreasing performance of the host.
- Syntax : same as `cudaMalloc ()`:

# Use of `cudaMallocHost ()`

- Normally, the memory allocated on the host (via `malloc`) is "mobile", *i.e.* it can go to the swap, then go to other segments of the host RAM. It is called *pageable memory*.
- CUDA provides the possibility to allocate non-pageable memory (pinned memory). It resides in a given part of the RAM and *does not move* until it is freed. That speeds up transfers with the host.
- Only root can normally do that. Beware of the use of excessive non-pageable memory $\Rightarrow$ risk of decreasing performance of the host.
- Syntax : same as `cudaMalloc ()`:

# Use of `cudaMallocHost ()`

- Normally, the memory allocated on the host (via `malloc`) is "mobile", *i.e.* it can go to the swap, then go to other segments of the host RAM. It is called *pageable memory*.
- CUDA provides the possibility to allocate non-pageable memory (pinned memory). It resides in a given part of the RAM and *does not move* until it is freed. That speeds up transfers with the host.
- Only root can normally do that. Beware of the use of excessive non-pageable memory $\Rightarrow$ risk of decreasing performance of the host.
- Syntax : same as `cudaMalloc ()`:

# Use of `cudaMallocHost ()`

- Normally, the memory allocated on the host (via `malloc`) is "mobile", *i.e.* it can go to the swap, then go to other segments of the host RAM. It is called *pageable memory*.
- CUDA provides the possibility to allocate non-pageable memory (pinned memory). It resides in a given part of the RAM and *does not move* until it is freed. That speeds up transfers with the host.
- Only root can normally do that. Beware of the use of excessive non-pageable memory $\Rightarrow$ risk of decreasing performance of the host.
- Syntax : same as `cudaMalloc ()`:

# Freeing non-pageable memory on host

This memory MUST be freed after use, obviously.

- Can you do that with `free ()` ?
- Can you do that with `CudaFree ()` ?

# Freeing non-pageable memory on host

This memory MUST be freed after use, obviously.

- Can you do that with `free ()` ?
- Can you do that with `CudaFree ()` ?

# Freeing non-pageable memory on host

This memory MUST be freed after use, obviously.

- Can you do that with `free ()` ?
- Can you do that with `CudaFree ()` ?

None of these !... It must be freed with `CudaFreeHost ()`

# bandwidth with pinned memory

Modify the `bandwidth.cu` file in order to allocate (and free !) non-pageable memory.

Check how the bandwidth is improved

Conclusions ?

# SDK's `bandwidthTest`

A more sophisticated implementation of this bandwidth test already exists in NVIDIA's SDK.
Check the following, and compare with your own results:

```
./bandwidthTest
```

```
./bandwidthTest -memory=pinned
```

More information can be obtained with

```
./bandwidthTest -help
```

# So far...

we have not programmed the GPU yet.

- We have not written a kernel yet
- Everything we have written runs on the host
- We have only interacted with the GPU for
  - allocations in the global memory
  - transfers involving the global memory
- That's all !

# So far...

we have not programmed the GPU yet.

- We have not written a kernel yet
- Everything we have written runs on the host
- We have only interacted with the GPU for
  - allocations in the global memory
  - transfers involving the global memory
- That's all !

# So far...

we have not programmed the GPU yet.

- We have not written a kernel yet
- Everything we have written runs on the host
- We have only interacted with the GPU for
  - allocations in the global memory
  - transfers involving the global memory
- That's all !

# So far...

we have not programmed the GPU yet.

- We have not written a kernel yet
- Everything we have written runs on the host
- We have only interacted with the GPU for
  - allocations in the global memory
  - transfers involving the global memory
- That's all !

# So far...

we have not programmed the GPU yet.

- We have not written a kernel yet
- Everything we have written runs on the host
- We have only interacted with the GPU for
  - allocations in the global memory
  - transfers involving the global memory
- That's all !

Programming our first kernel is the purpose of the next lecture.