Programming in CUDA

Part V. Optimizing CUDA — 2

Frédéric S. Masset

Instituto de Ciencias Físicas, UNAM

Escuela de Verano GPU 2014

Outline

Outline

Outline

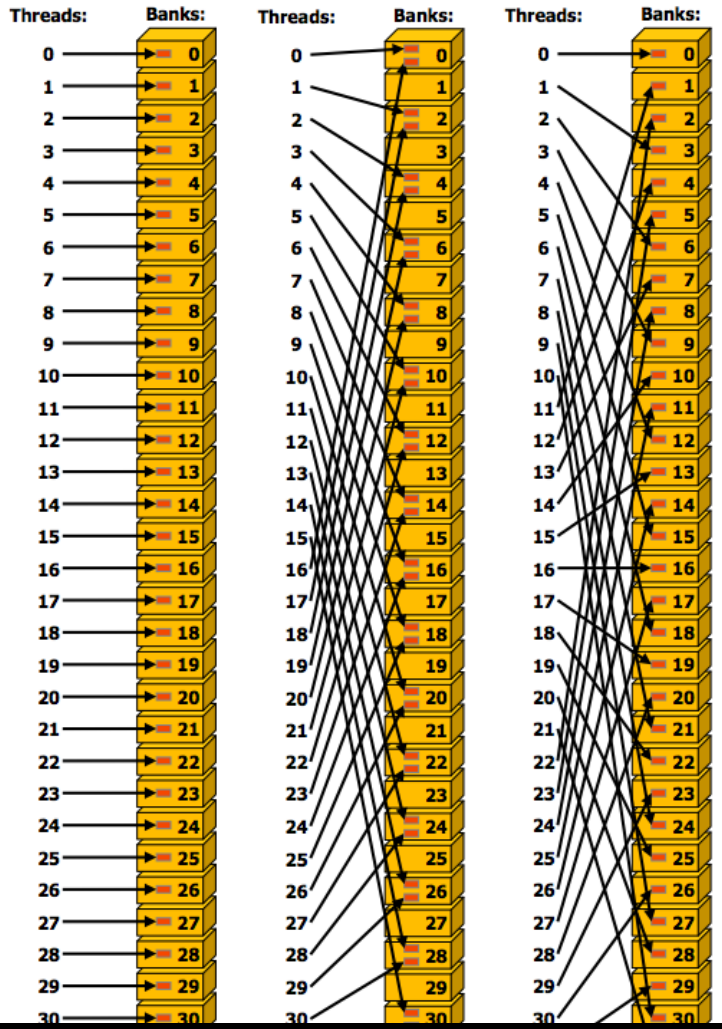
Outline

Bank conflicts

The shared memory is organized in banks (16 for 1.x architectures, 32 for Fermi's).

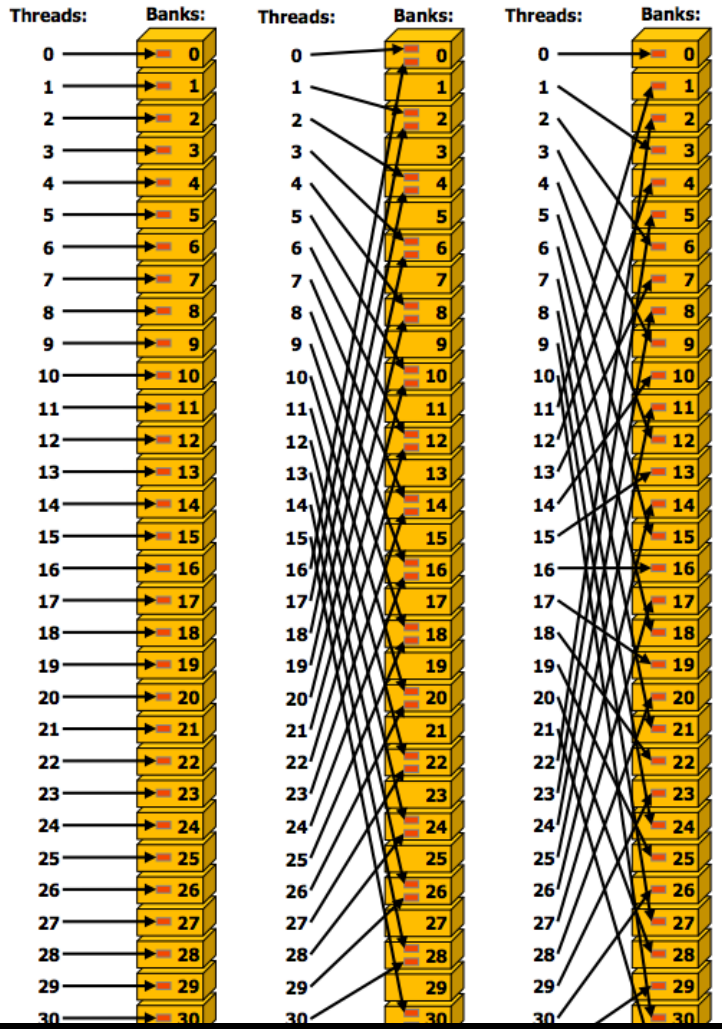
- If all threads in a half warp access different banks, they can do it in parallel $\Rightarrow$ fast access.
- If this is not the case, they will access the shared memory sequentially $\Rightarrow$ this is called `warp serialize`. It slows down the execution.

Bank conflicts — from NVIDIA's CUDA C Programming guide



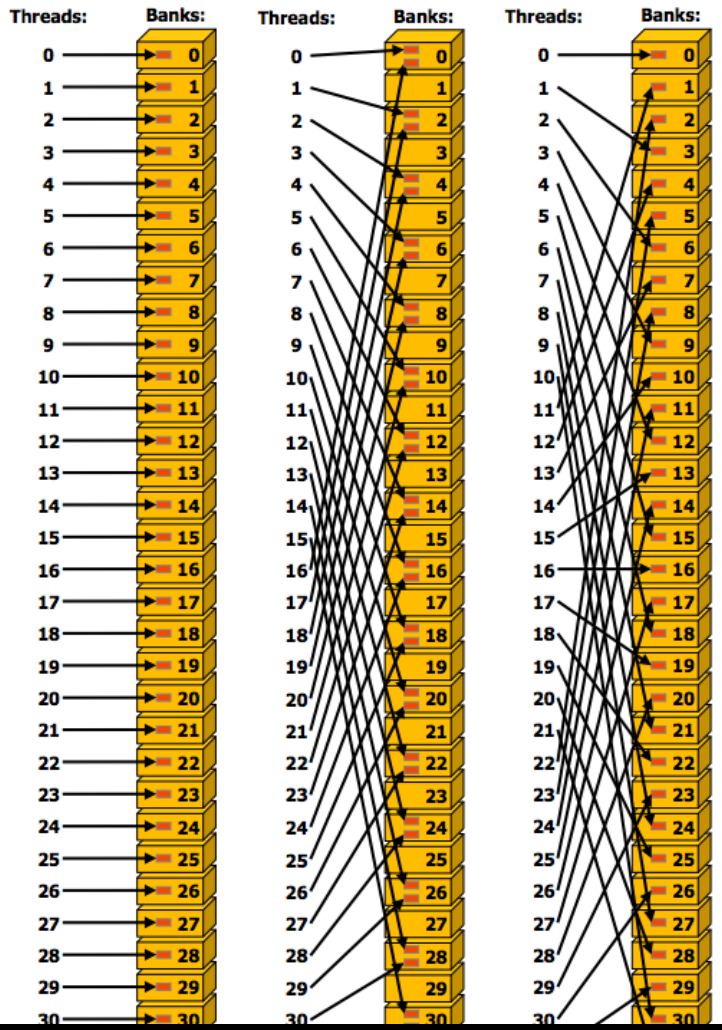
- Left : linear addressing with a stride of one 32-bit word $\Rightarrow$ no bank conflict.
- Middle : linear addressing with a stride of two 32-bit words $\Rightarrow$ 2-way bank conflict.
- Right : Linear addressing with a stride of three 32-bit words (no bank

Bank conflicts — from NVIDIA's CUDA C Programming guide



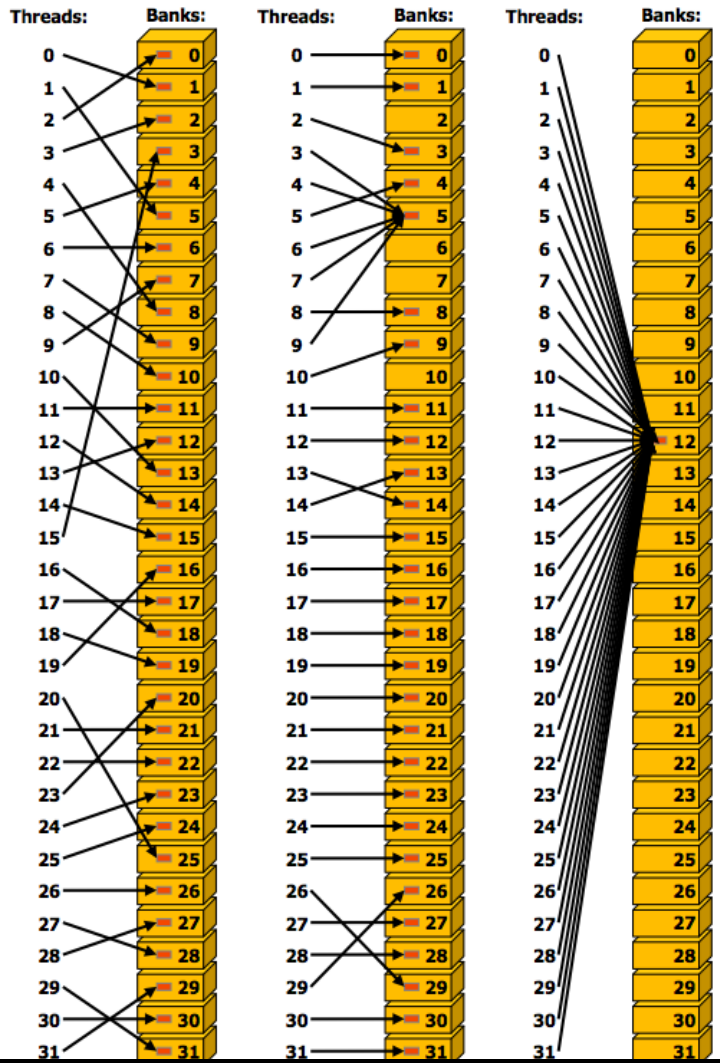
- Left : linear addressing with a stride of one 32-bit word ⇒ no bank conflict.
- Middle : linear addressing with a stride of two 32-bit words ⇒ 2-way bank conflict.
- Right : Linear addressing with a stride of three 32-bit words (no bank

Bank conflicts — from NVIDIA's CUDA C Programming guide



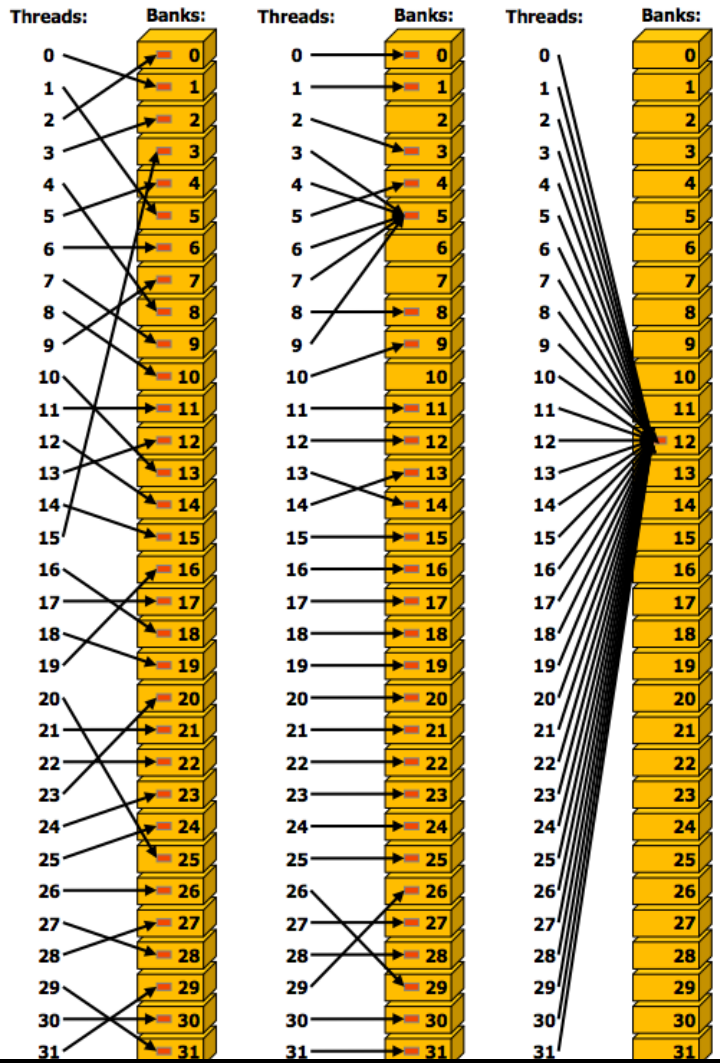
- Left : linear addressing with a stride of one 32-bit word $\Rightarrow$ no bank conflict.
- Middle : linear addressing with a stride of two 32-bit words $\Rightarrow$ 2-way bank conflict.
- Right : Linear addressing with a stride of three 32-bit words (no bank

Bank conflicts — from NVIDIA's CUDA C Programming guide



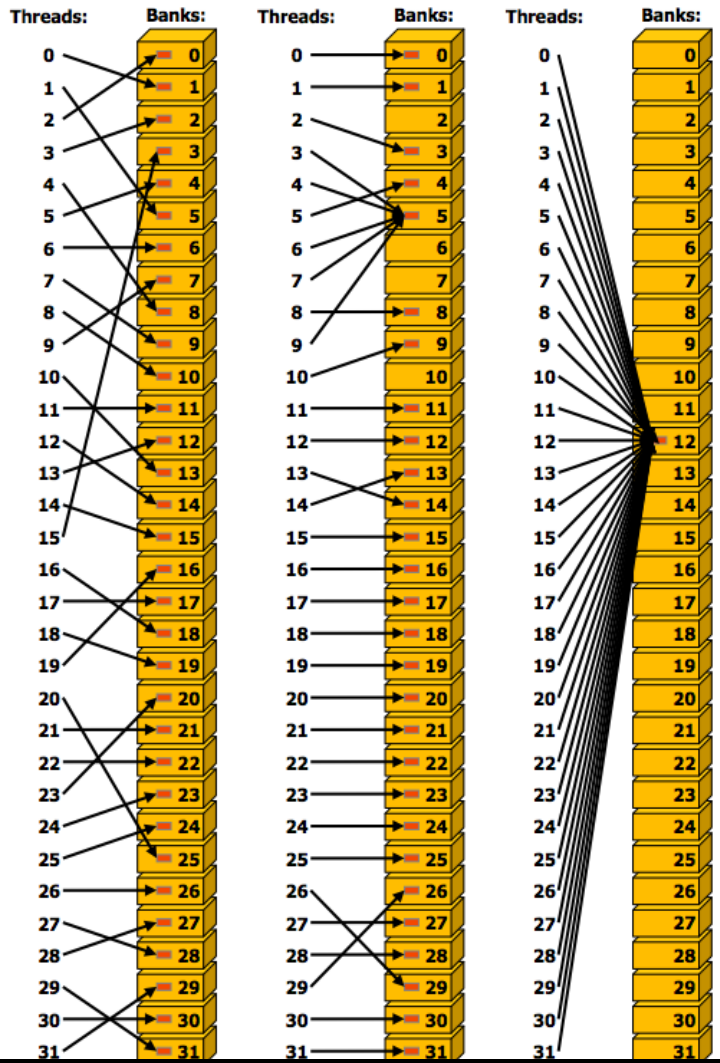
- Left : Conflict-free access via random permutation
- Middle : Conflict-free access since threads 3, 4, 6, 7 and 9 access the same word within bank 5.
- Right : Conflict-free broadcast access (all threads access the same word)

Bank conflicts — from NVIDIA's CUDA C Programming guide



- Left : Conflict-free access via random permutation
- Middle : Conflict-free access since threads 3, 4, 6, 7 and 9 access the same word within bank 5.
- Right : Conflict-free broadcast access (all threads access the same word)

Bank conflicts — from NVIDIA's CUDA C Programming guide



- Left : Conflict-free access via random permutation
- Middle : Conflict-free access since threads 3, 4, 6, 7 and 9 access the same word within bank 5.
- Right : Conflict-free broadcast access (all threads access the same word).

Outline

Warp divergence

Sets of 32 threads (warps) are executed in a SIMD fashion (*Single Instruction Multiple Data*). Assume the execution flow encounters an `if` statement:

```
if (condition1) {   true for some threads of the warp
   statement 1;     Threads true execute, others wait
   ....
} else {
   statement 2;     Threads true wait, others execute
   ....
}
```

This is called warp divergence. It should be avoided.

Divergence or not ?

Consider the filling of extra rows and columns in the shared memory (practice `convol2`).

```
if ((il == 1) && (ig > 0))
  la[ll-1] = gpu_a[lg-1];
if ((jl == 1) && (jg > 0))
  la[ll-BLOCK_X-2] = gpu_a[lg-pitch];
```

Is the first test divergent ?
Is the second test divergent ?

Outline

Diagnostic of a kernel : the CUDA profiler

The CUDA profiler is extremely simple to activate. Before running your executable, set the `CUDA_PROFILE` environment variable to 1 and you are done !

- Go back to the `convol2` practice
- Issue: `export CUDA_PROFILE=1`, then run `./a.out` (*Note: no need to rebuild*).
- Do a `ls` command. There is a new file. Have a look at it.

By default, the CUDA profiler gives the execution time on CPU and GPU (in microseconds), and the occupancy, for all methods which are GPU related (kernels and memcpy's). From the examination of the profiler's output, recover the fact that standard `memcpy`'s are synchronous, whereas kernel calls are asynchronous.

Other output of CUDA profiler

In the `convol2` directory, create a file `foo`, which contains:

```
gld_coherent
gst_coherent
gld_incoherent
gst_incoherent
```

In order to tell the profiler to use this file, issue:

```
export CUDA_PROFILE_CONFIG=foo
```

then run the executable again, and check the content of the log file. Can you interpret its output, both qualitatively and quantitatively ? Note that the count is not exact, and that it can omit some multiprocessors.

More output of CUDA profiler

Other interesting diagnostic are the number of divergent branches, the number of warp serializes (bank conflicts ?):

```
warp_serialize
divergent_branch
```

More output of CUDA profiler

Other interesting diagnostic are the number of divergent branches, the number of warp serializes (bank conflicts ?):

```
warp_serialize
divergent_branch
```

Note: there exists a GUI front-end to the CUDA profiler

Outline

Occupancy

Threads instructions are executed sequentially, so launching other warps is the only way to hide latencies (essentially `gmem` transactions) and keep the hardware busy.

Occupancy: number of active warps divided by the maximum number of warps that can run concurrently.

Limited by resource usage : Registers, shared memory, threads / blocks.

Estimating the occupancy of your kernel

NVIDIA has released an Excel spreadsheet that allows to estimate the occupancy of a given kernel. You need to select your compute capability, then you must enter your resource usage in the orange zone.

How do I know my resource usage ?
Prior to version 3.0, you could compile using option `-cubin`:

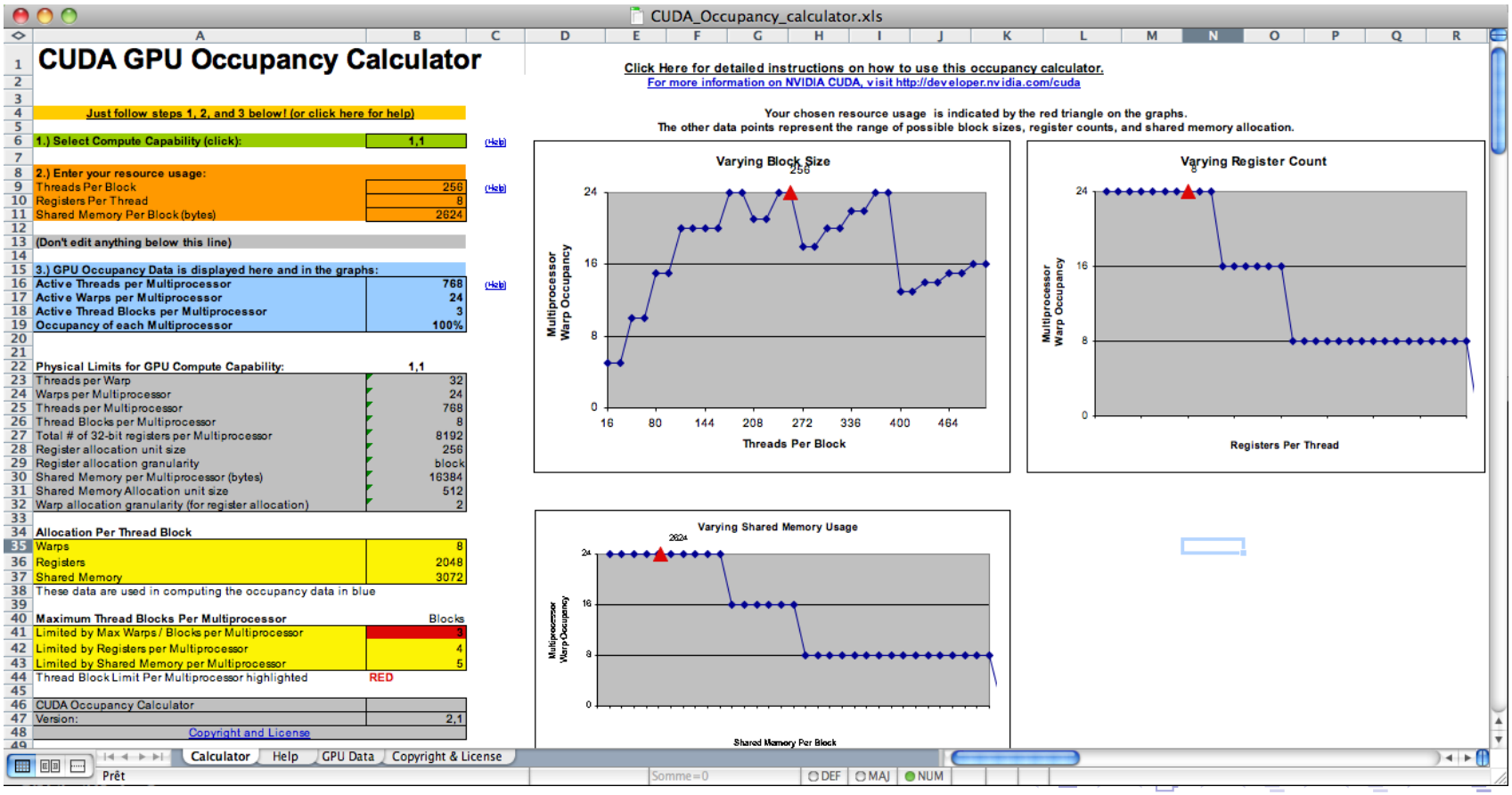
```
nvcc -cubin myfile.cu
```

which produces a `.cubin` file that contains the resource usage. On more recent versions the cubin file is unreadable (binary format). You still can know your resource usage by issuing:

```
nvcc -ptxas-options=-v -c myfile.cu
```

Occupancy calculator spreadsheet

```
masset:sol> nvcc -ptxas-options=-v -c convol.cu
ptxas info    : Compiling entry function '_Z7kconvolPfS_ii' for 'sm_10'
ptxas info    : Used 8 registers, 2608+16 bytes smem, 20 bytes cmem[1]
```



Outline

Debugging with CUDA

Prior to version 3.0, one could build a code with option `-deviceemu`. This would build an executable that would actually run on the CPU and could be debugged on the CPU.

This no longer exists in more recent versions of CUDA. If you want to use this, then you must install an older version of CUDA (3.0 at most).

There is now a debugger directly for the gpu: `cuda-gdb`. It exists only for LINUX, and will be soon released for Mac OS X.

Debugging with CUDA

Prior to version 3.0, one could build a code with option `-deviceemu`. This would build an executable that would actually run on the CPU and could be debugged on the CPU.

This no longer exists in more recent versions of CUDA. If you want to use this, then you must install an older version of CUDA (3.0 at most).

There is now a debugger directly for the gpu: `cuda-gdb`. It exists only for LINUX, and will be soon released for Mac OS X.

In any case a kernel should be always compared to the sequential CPU version, as in the practice, to make sure it behaves as expected.

What is OpenCL

- Open Computational Language
- Cross-platform parallel computing API (GPUs, CPUs)
- C-like language
- Code is portable across devices
- OpenCL supports for AMD and NVIDIA GPUs, x86 CPUs, IBM Cell, etc.

OpenCL / CUDA analogies

- Work is submitted by kernels
- Kernels are divided into work groups (CUDA blocks)
- Work groups are divided into work items (CUDA threads)
- Work items can synchronize with each other within a work group
- Global and constant memory are called the same in OpenCL and CUDA

OpenCL / CUDA differences

- Kernel is a string in OpenCL !
- Build at runtime ! No errors can be detected before
- The user is responsible for taking care of the compiled object.
- Shared memory (CUDA) is called local memory in OpenCL
- Local memory (CUDA) is called private memory in OpenCL...