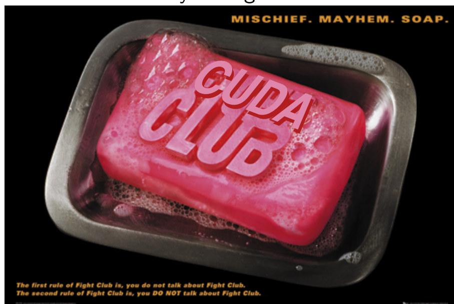
CUDAClub

Mike Peardon

School of Mathematics
Trinity College Dublin



January 13, 2009

Disclaimer:

I don't know anything about CUDA
which is why I'm at this journal club

http://www.nvidia.com/object/cuda_home.html
http://www.nvidia.com/object/cuda_develop.html

What is CUDA?

Compute Unified Device Architecture

- Extension to C programming language
- Adds library functions to access to GPU
- Adds directives to translate C into instructions that run on the host CPU or the GPU when needed
- Allows easy multi-threading - parallel execution on all thread processors on the GPU

What is a GPU?

Graphics Processing Unit

- Processor dedicated to rapid rendering of polygons - texturing, shading
- They are mass-produced, so very cheap 1 Tflop peak $\approx$ EU 1k.
- They have lots of compute cores, but a simpler architecture cf a standard CPU
- The "shader pipeline" can be used to do floating point calculations
- $\longrightarrow$ cheap scientific/technical computing

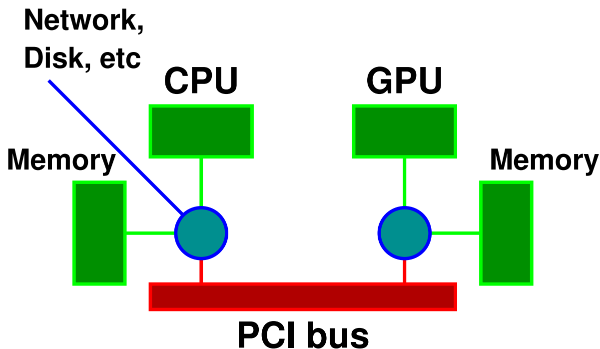
Will CUDA work on my PC/laptop?

- CUDA works on modern nVidia cards (Quadro, GeForce, Tesla)
- See
  `http://www.nvidia.com/object/cuda_learn_products.html`

nVidia's compiler - `nvcc`

- CUDA code must be compiled using `nvcc`
- `nvcc` generates both instructions for host and GPU (PTX instruction set), as well as instructions to send data back and forwards between them
- Standard CUDA install; `/usr/local/cuda/bin/nvcc`
- Shell executing compiled code needs dynamic linker path `LD_LIBRARY_PATH` environment variable set to include `/usr/local/cuda/lib`
- . . . and that's it!
- Probably can even get around this . . .

Simple overview



- GPU can't directly access main memory
- CPU can't directly access GPU memory
- Need to explicitly copy data
- No `printf`!

Very simple example - A CUDA “hello world”

```
#include <stdio.h>
int main()
{
  int i,n;
  struct cudaDeviceProp x;

  cudaGetDeviceCount(&n);
  printf("Found %d CUDA-enabled devices\n",n);

  for (i=0;i<n;i++)
  {
    cudaGetDeviceProperties(&x, i);
    printf("GPU %d <%s> has %d multi-processors \n",
        i, x.name, x.multiProcessorCount);
  }
}
```

Very simple example - A CUDA “hello world”

- Output on `pasanda` is:

```
Found 2 CUDA-enabled devices
GPU 0 <Tesla C1060> has 30 multi-processors
GPU 1 <Quadro NVS 290> has 2 multi-processors
```

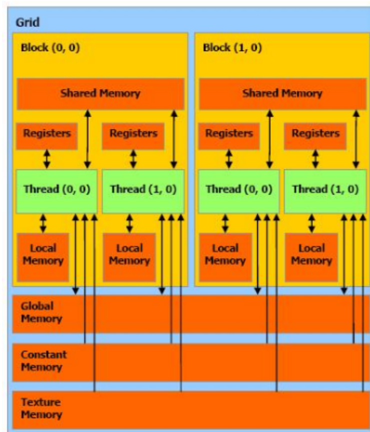
- A multi-processor has 8 thread processors

Writing some code (1) - specifying where code runs

- CUDA provides **function type qualifiers** (that are not in C/C++) to enable programmer to define where a function should run
- `__host__`: specifies the code should run on the host CPU (redundant on its own - it is the default)
- `__device__`: specifies the code should run on the GPU, and the function can only be called by code running on the GPU
- `__global__`: specifies the code should run on the GPU, but be called from the host - this is the **access point** to start multi-threaded codes running on the GPU
- Device can't execute code on the host!
- CUDA imposes some restrictions, such as device code is C-only (host code can be C++), device code can't be called recursively...

Writing some code (2) - launching a `__global__` function

- All calls to a `__global__` function must specify how many threaded copies are to launch and in what configuration.
- CUDA syntax: `<<< >>>`
- threads are grouped into thread blocks then into a grid of blocks
- This defines a memory heirarchy (probably important for performance)



Writing some code (3) - launching a `__global__` function

- Inside the `<<< >>>`, need at least two arguments (can be two more, that have default values)
- Call looks eg. like `my_func<<<bg, tb>>>(arg1, arg2)`
- `bg` specifies the dimensions of the `block grid` and `tb` specifies the dimensions of each `thread block`
- `bg` and `tb` are both of type `dim3` (a new datatype defined by CUDA; three unsigned ints where any unspecified component defaults to 1).
- `dim3` has struct-like access - members are `x`, `y` and `z`
- CUDA provides constructor: `dim3 mygrid(2,2);` sets `mygrid.x=2`, `mygrid.y=2` and `mygrid.z=1`
- 1d syntax allowed: `myfunc<<<5, 6>>>()` makes 5 blocks (in linear array) with 6 threads each and runs `myfunc` on them all.

Writing some code (4) - built-in variables on the GPU

- For code running on the GPU (`__device__` and `__global__`), some variables are predefined, which allow threads to be located inside their blocks and grids
- `dim3 gridDim` Dimensions of the grid.
- `uint3 blockIdx` location of this block in the grid.
- `dim3 blockDim` Dimensions of the blocks
- `uint3 threadIdx` location of this thread in the block.
- `int warpSize` number of threads in the warp?

Example 2 - vector adder

Start:

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

#define N 1000
#define NBLOCK  10
#define NTHREAD 10
```

- Now define the kernel to execute on the host

```
__global__
void adder(int n, float* a, float *b)
// a=a+b - thread code - add n numbers per thread
{
  int i,off = (N * blockIdx.x ) / NBLOCK +
    (threadIdx.x * N) / (NBLOCK * NTHREAD);

  for (i=off;i<off+n;i++)
  {
    a[i] = a[i] + b[i];
  }
}
```

Example 2 - vector adder (2)

- Call using

```
cudaMemcpy(gpu_a, host_a, sizeof(float) * n,
    cudaMemcpyHostToDevice);
cudaMemcpy(gpu_b, host_b, sizeof(float) * n,
    cudaMemcpyHostToDevice);

adder<<<NBLOCK, NTHREAD>>>(n / (NBLOCK * NTHREAD), gpu_a, gpu_b);

cudaMemcpy(host_c, gpu_a, sizeof(float) * n,
    cudaMemcpyDeviceToHost);
```

- Need the `cudaMemcpy`'s to push/pull the data on/off the GPU.