A beginner's guide to programming GPUs with CUDA

Mike Peardon

School of Mathematics
Trinity College Dublin

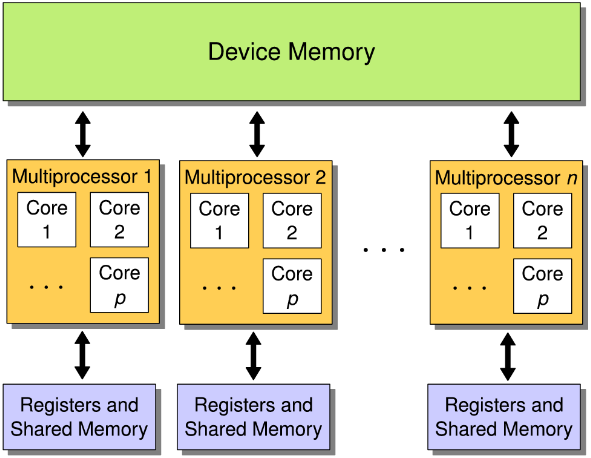
April 24, 2009

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

What is a GPU? (2)



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

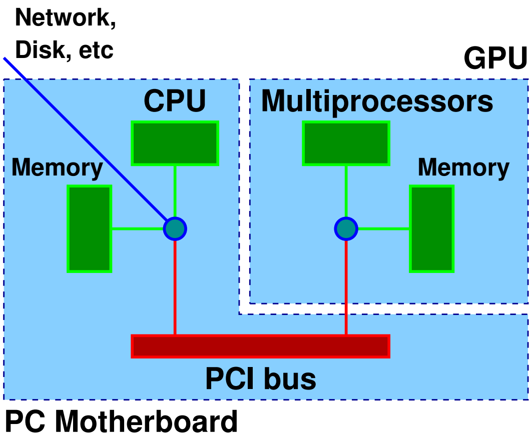
Will CUDA work on my PC/laptop?

- 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`

Simple overview

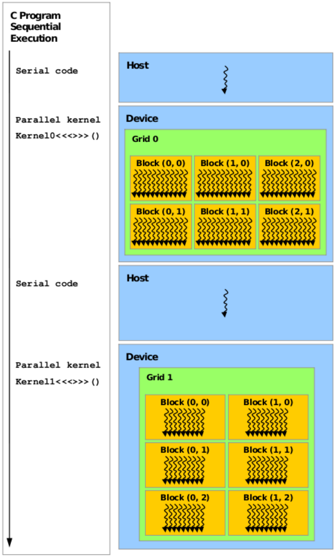


- GPU can't directly access main memory
- CPU can't directly access GPU memory
- Need to explicitly copy data
- No `printf`!

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...

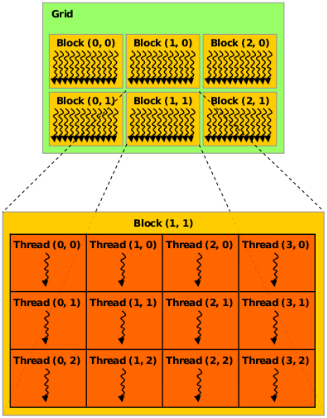
Code execution



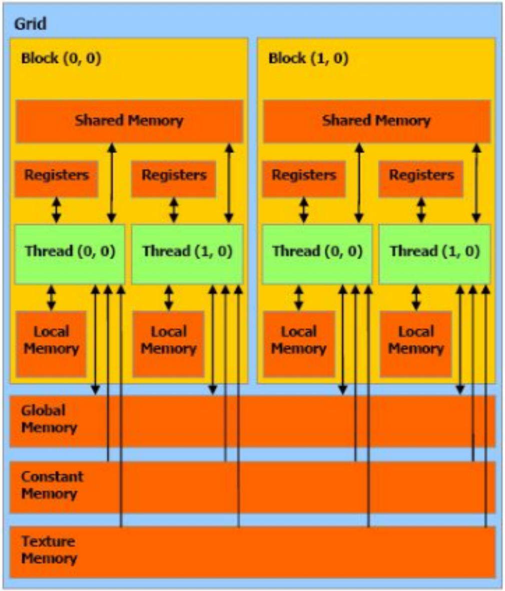
Serial code executes on the host while parallel code executes on the device.

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 (important for performance)



The thread/block/grid model



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?

Writing some code (5) - where variables are stored

- For code running on the GPU (`__device__` and `__global__`), the memory used to hold a variable can be specified.
- `__device__`: the variable resides in the GPU's global memory and is defined while the code runs.
- `__constant__`: the variable resides in the constant memory space of the GPU and is defined while the code runs.
- `__shared__`: the variable resides in the shared memory of the thread block and has the same lifespan as the block. block.

Example - vector adder

Start:

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

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

- 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 - 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.

arXiv:0810.5365 Barros *et. al.*

"Blasting through lattice calculations using CUDA"

- An implementation of an important compute kernel for lattice QCD - the Wilson-Dirac operator - this is a sparse linear operator that represents the kinetic energy operator in a discrete version of the quantum field theory of relativistic quarks (interacting with gluons).
- Usually, performance is limited by memory bandwidth (and inter-processor communications).
- Data is stored in the GPU's memory
- "Atom" of data is the spinor of a field on one site. This is 12 complex numbers (3 colours for 4 spins).
- They use the `float4` CUDA data primitive, which packs four floating point numbers efficiently. An array of 6 `float4` types then holds one lattice size of the quark field.

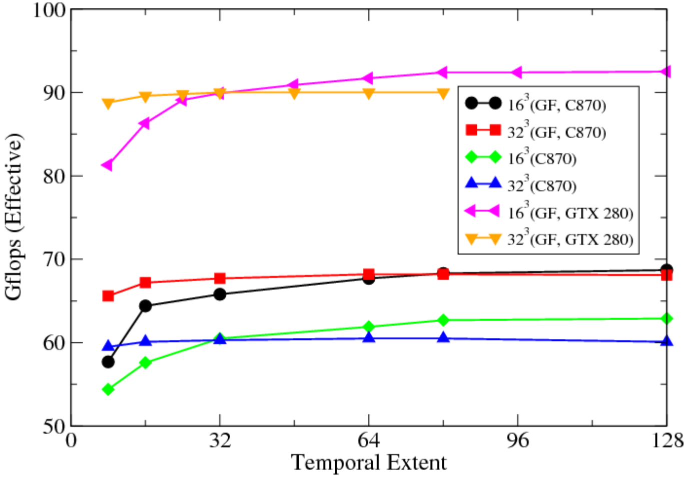
arXiv:0810.5365 Barros *et. al.* (2)

Performance issues:

1. 16 threads can read 16 contiguous memory elements very efficiently - their implementation of 6 arrays for the spinor allows this contiguous access
2. GPUs do not have caches; rather they have a small but fast shared memory. Access is managed by software instructions.
3. The GPU has a very efficient *thread manager* which can schedule multiple threads to run withing the cores of a multi-processor. Best performance comes when the number of threads is (much) more than the number of cores.
4. The local shared memory space is 16k - not enough! Barros et al also use the registers on the multiprocessors (8,192 of them). Unfortunately, this means they have to hand-unroll all their loops!

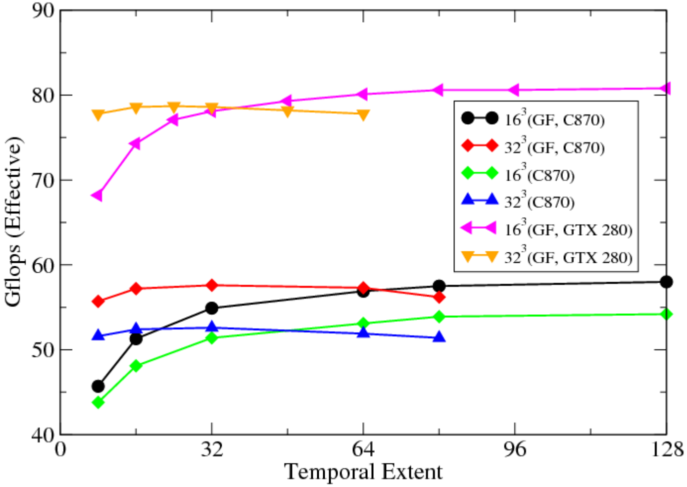
arXiv:0810.5365 Barros *et. al.* (3)

Performance: (even-odd) Wilson operator



arXiv:0810.5365 Barros *et. al.* (4)

Performance: Conjugate Gradient solver:



Conclusions

- The GPU offers a very impressive architecture for scientific computing on a single chip.
- Peak performance now is close to 1 TFlop for less than EU 1,000
- CUDA is an extension to C that allows multi-threaded software to execute on modern nVidia GPUs. There are alternatives for other manufacturer's hardware and proposed architecture-independent schemes (like OpenCL)
- Efficient use of the hardware is challenging; threads must be scheduled efficiently and synchronisation is slow. Memory access must be defined very carefully.
- The (near) future will be very interesting...