An introduction to GPU programming

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford-Man Institute of Quantitative Finance

Oxford eResearch Centre

Overview

- motivation & objectives
- hardware view
- software view
- CUDA programming

Motivation

Current Intel quad-core Xeon:

- 4 cores
- 10-40 GFflops in SP; 10-20 GFlops in DP
- 20-30GB/s memory bandwidth

Current NVIDIA Tesla GPU

- 240 cores
- 1 Tflops in SP; 125 GFlops in DP
- 100GB/s memory bandwidth

1 Tesla GPU is usually 5-10$\times$ faster than two quad-core Xeons, at roughly same cost and power consumption

Motivation

Next generation Intel CPUs:

- Westmere-EP: 6 core
- Nehalem-EX: 8 core
- Sandy Bridge: 4-10 cores, and SSE vectur unit replaced by AVX with 8 `float` or 4 `double`

Next generation NVIDIA Fermi GPU:

- 512 SP cores, 256 DP cores
- 1.5 Tflops in SP; 800 GFlops in DP
- L1/L2 cache on GPU
- 200GB/s memory bandwidth

Motivation

Other GPUs?

- AMD:
  - similar to NVIDIA for graphics/games applications
  - not very focussed on computing side, but supporting new OpenCL standard
- IBM Cell:
  - proved to be challenging to program
  - no further development for scientific computing?
- Intel Larrabee:
  - 16-32 cores each with a vector unit
  - project has suffered major delays, first-generation product has been cancelled

Motivation

My predictions?

- GPUs will continue to evolve, with increasing cores and features to make them more easily programmed
- CPUs will also continue to evolve with increasing cores and vector lengths will increase to compete with GPUs
- GPUs will stay ahead because:
  - better bandwidth on graphics card than in motherboard socket
  - GPUs aimed at high-end market, whereas CPUs aimed more at low-end mobile mass market

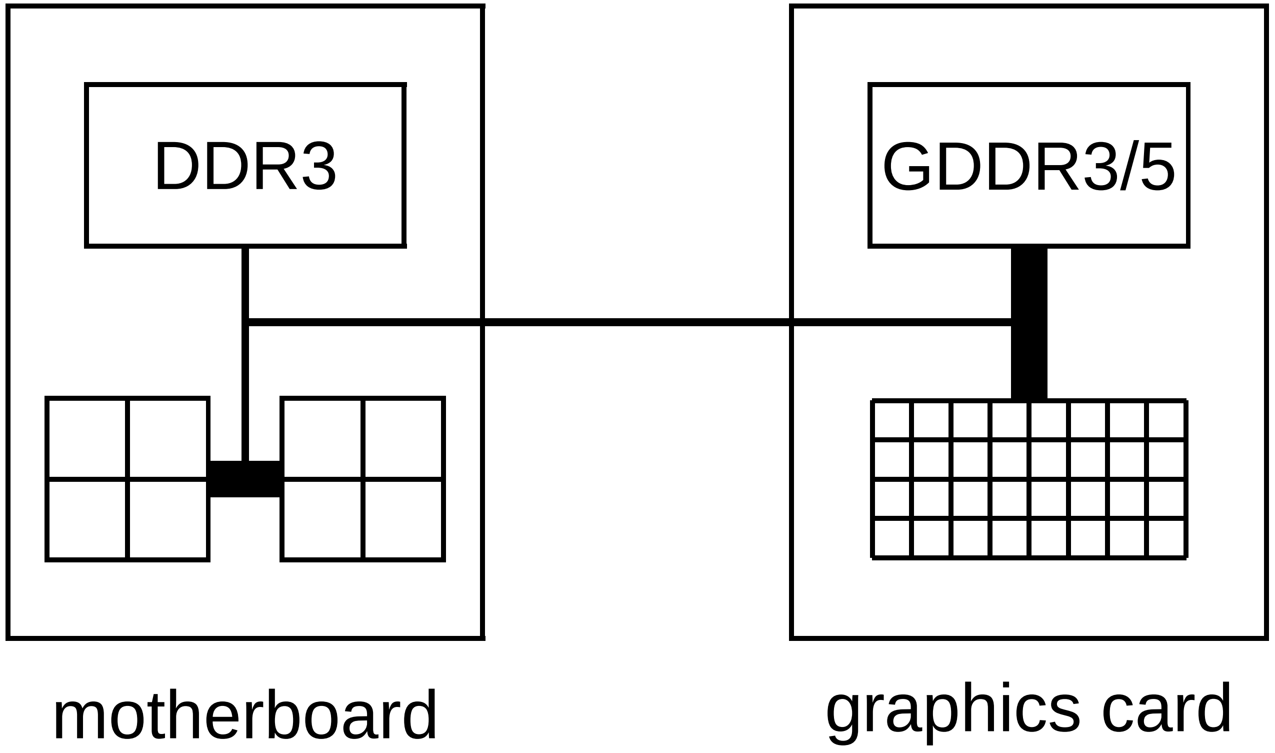
Objectives

- an overview understanding of GPU hardware and software
- hands-on experience of CUDA programming
  (very relevant to emerging OpenCL open standard)
- learn about some key challenges, and how to approach the GPU implementation of a new application
- learn about resources for future learning

I hope to convince you it’s not difficult!

Hardware view

At the top-level, a PCIe graphics card with a many-core GPU and high-speed graphics “device” memory sits inside a standard PC/server with one or two multicore CPUs:



Hardware view

At the GPU level:

- basic building block is a “streaming multiprocessor” with
  - 8 cores, each with 2048 registers
  - 16KB of shared memory
  - 8KB cache for constants held in device memory
  - 8KB cache for textures held in device memory
- different chips have different numbers of these SMs:

product	SMs	bandwidth	memory
GTX260	27	110 GB/s	1-2 GB
GTX285	30	160 GB/s	1-2 GB
Tesla M1060	30	102 GB/s	4 GB

Hardware view

Key hardware feature is that the 8 cores in a multiprocessor are SIMT (Single Instruction Multiple Threads) cores:

- all 8 cores execute the same instructions simultaneously, but with different data
- similar to vector computing on CRAY supercomputers
- minimum of 4 threads per core, so end up with a minimum of 32 threads all doing the same thing at (almost) the same time
- natural for graphics processing and much scientific computing
- SIMT is also a natural choice for many-core chips to simplify each core

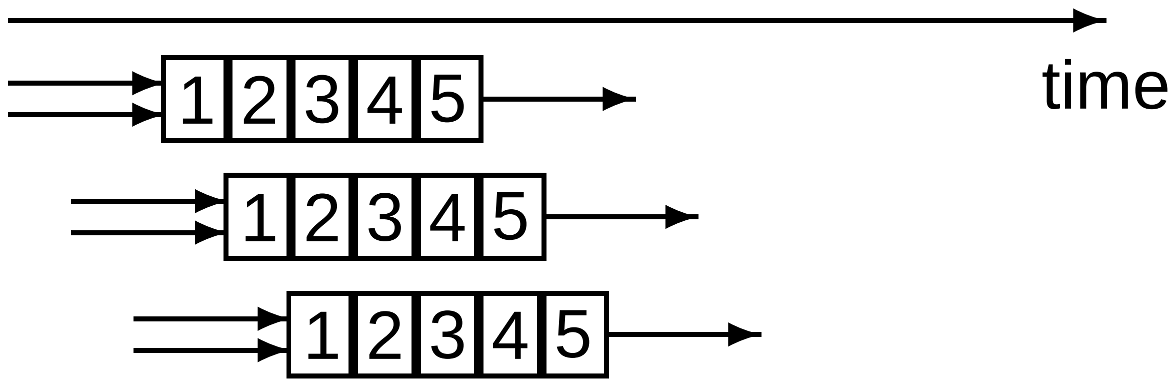
Multithreading

Lots of active threads is the key to high performance:

- no "context switching"; each thread has its own registers, which limits the number of active threads
- threads execute in "warps" of 32 threads per multiprocessor (4 per core) – execution alternates between "active" warps, with warps becoming temporarily "inactive" when waiting for data

Multithreading

- for each thread, one operation completes long before the next starts – avoids the complexity of pipeline overlaps which can limit the performance of modern processors



- memory access from device memory has a delay of 400-600 cycles; with 40 threads this is equivalent to 10-15 operations and can be managed by the compiler

Software view

At the top level, we have a master process which runs on the CPU and performs the following steps:

1. initialises card
2. allocates memory in host and on device
3. copies data from host to device memory
4. launches multiple copies of execution “kernel” on device
5. copies data from device memory to host
6. repeats 3-5 as needed
7. de-allocates all memory and terminates

Software view

At a lower level, within the GPU:

- each copy of the execution kernel executes on an SM
- if the number of copies exceeds the number of SMs, then more than one will run at a time on each SM if there are enough registers and shared memory, and the others will wait in a queue and execute later
- all threads within one copy can access local shared memory but can't see what the other copies are doing (even if they are on the same SM)
- there are no guarantees on the order in which the copies will execute

CUDA programming

CUDA is NVIDIA's program development environment:

- based on C with some extensions
- C++ support increasing steadily
- FORTRAN support provided by PGI compiler
- basis for OpenCL standard pushed by Apple and supported by AMD, Intel and IBM
- lots of example code and good documentation
  – 2-4 week learning curve for those with experience of OpenMP and MPI programming
- large user community on NVIDIA forums

CUDA programming

At the host code level, there are library routines for:

- memory allocation on graphics card
- data transfer to/from device memory
  - constants
  - texture arrays (useful for lookup tables)
  - ordinary data
- error-checking
- timing

There is also a special syntax for launching multiple copies of the kernel process on the GPU.

CUDA programming

In its simplest form it looks like:

```
kernel_routine<<<gridDim, blockDim>>>(args);
```

where

- `gridDim` is the number of copies of the kernel (the “grid” size”)
- `blockDim` is the number of threads within each copy (the “block” size)
- `args` is a limited number of arguments, usually mainly pointers to arrays in graphics memory

The more general form allows `gridDim` and `blockDim` to be 2D or 3D to simplify application programs

CUDA programming

At the lower level, when one copy of the kernel is started on a multiprocessor it is executed by a number of threads, each of which knows about:

- some variables passed as arguments
- pointers to arrays in device memory (also arguments)
- global constants in device memory
- shared memory and private registers/local variables
- some special variables:
    - `gridDim` size (or dimensions) of grid of blocks
    - `blockIdx` index (or 2D/3D indices)of block
    - `blockDim` size (or dimensions) of each block
    - `threadIdx` index (or 2D/3D indices) of thread

CUDA programming

The kernel code looks fairly normal once you get used to two things:

- code is written from the point of view of a single thread
  - quite different to OpenMP multithreading
  - similar to MPI, where you use the MPI “rank” to identify the MPI process
  - all local variables are private to that thread
- need to think about where each variable lives
  - any operation involving data in the device memory forces its transfer to/from registers in the GPU
  - there’s no cache (currently) so a second operation with the same data will force a second transfer
  - usually better to copy the value into a local register variable

Host code

```
int main(int argc, char **argv) {
  float *h_x, *d_x;          //  h=host, d=device
  int   nblocks=2, nthreads=8, nsize=2*8;

  h_x = (float *)malloc(nsize*sizeof(float));
  cudaMalloc((void **)&d_x, nsize*sizeof(float))

  my_first_kernel<<<nblocks,nthreads>>>(d_x);

  cudaMemcpy(h_x,d_x,nsize*sizeof(float),
                 cudaMemcpyDeviceToHost);

  for (int n=0; n<nsize; n++)
    printf(" n,  x  =  %d  %f \n",n,h_x[n]);

  cudaFree(d_x); free(h_x);
}
```

Kernel code

```
#include <cutil_inline.h>

__global__ void my_first_kernel(float *x)
{
  int tid = threadIdx.x + blockDim.x*blockIdx.x;

  x[tid] = threadIdx.x;
}
```

- `__global__` identifier says it’s a kernel function
- each thread sets one element of x array
- within each block of threads, `threadIdx.x` ranges from 0 to `blockDim.x-1`, so each thread has a unique value for `tid`

CUDA distribution

3 components:

- graphics driver
- toolkit (nvcc compiler and libraries)
- SDK code samples and utilities

Everything is available for Windows, Linux and OS X.

I'll focus on Linux, but there is good integration for Visual Studio 2008.

CUDA Makefile

Two choices:

- use `nvcc` within a standard Makefile
- use the special Makefile template provided in the SDK

I use the SDK Makefile because it provides useful options:

- `make emu=1`
  uses an emulation library for debugging on a CPU
- `make dbg=1`
  activates run-time error checking (see Practical 1)

I would use a standard Makefile for more complex cases:

- producing a GPU library
- when needing different compiler flags for different files

Practical 1

- start from code shown above (but with comments)
- try out the various Makefile options
- modify code to add two vectors together (including sending them over from the host to the device)
- if time permits, look at CUDA SDK examples

Practical 1

Things to note:

- memory allocation
```
cudaMalloc((void **)&d_x, nbytes);
```
- data copying
```
cudaMemcpy(h_x,d_x,nbytes,
                  cudaMemcpyDeviceToHost);
```
- kernel routine is declared by `__global__` prefix, and is written from point of view of a single thread

Practical 1

Second version of the code is very similar to first, but uses CUDA SDK toolkit for various safety checks – gives useful feedback in the event of errors.

- check for error return codes:
  `cutilSafeCall( ... );`

- check for failure messages:
  `cutilCheckMsg( ... );`

Webpages

NVIDIA's CUDA homepage:
`www.nvidia.com/object/cuda_home.html`

Wikipedia overviews of NVIDIA GPUs:
`en.wikipedia.org/wiki/GeForce_200_Series`
`en.wikipedia.org/wiki/Nvidia_Tesla`

GPU computing community website:
`www.gpucomputing.net`

LIBOR test code:
`www.maths.ox.ac.uk/~gilesm/hpc/`