GPU computing

Part 1: General introduction

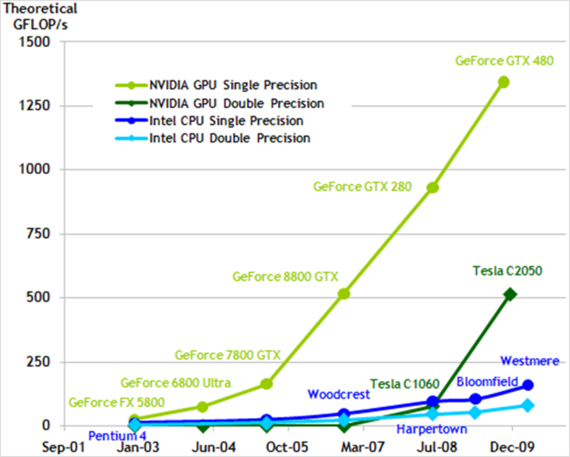
Ch. Hoelbling

Wuppertal University

Lattice Practices 2011

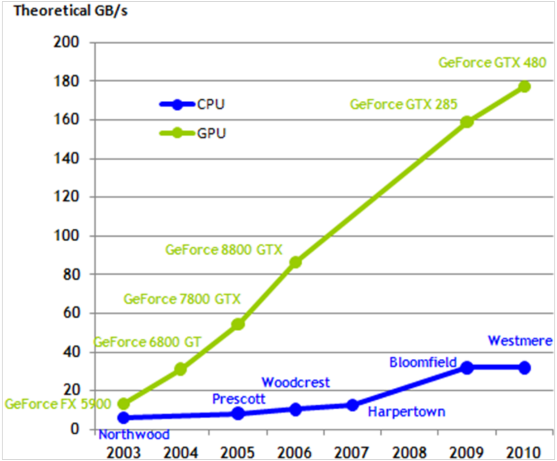
Outline

WHY GPUs?



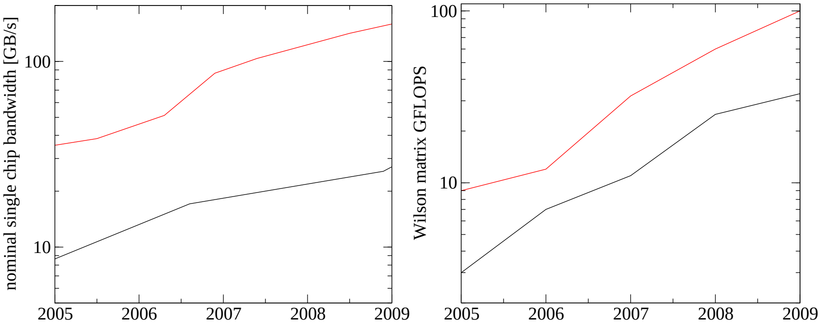
Far better peak performance than CPUs

WHY GPUs?



Even more important for QCD: memory bandwidth

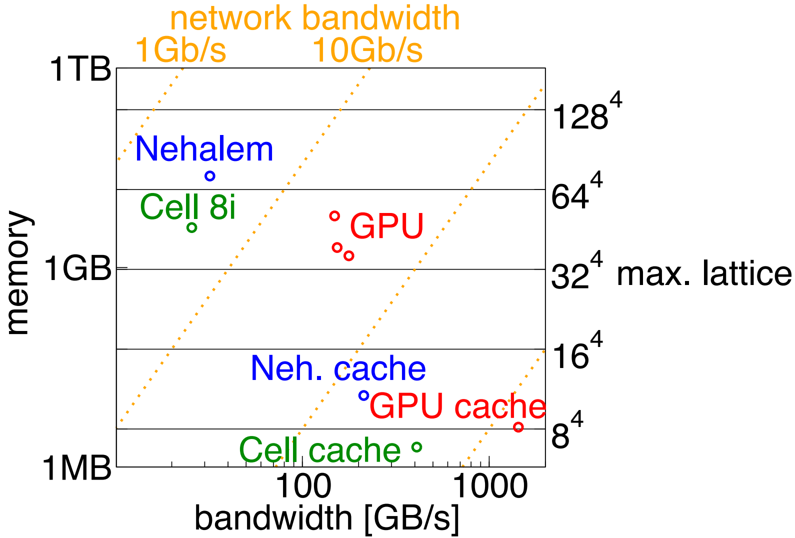
QCD CODE



GPUs give a factor $\sim$ 3 boost compared to CPUs

➛2-4 years ahead

RAM LANDSCAPE



How did GPUs become so capable?

Until mid 1990s: Framebuffer + BitBlock shifter

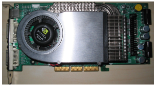
- 2D operations only
- CPU does all floating point work

Fixed pipline

- Hardware renders 3D polygons
- Simple texture to surface mapping

Programmable shaders

- Polygon and texture rendering programmable
- Floating point support



Fully general streaming processors

- Fully generic streaming processor
- All graphics rendering in software

TODAYS GPUs

Examples: GPU: NVidia GeForce GTX 480, ~350€

CPU: Intel Xeon W5590 (Nehalem), ~1400€

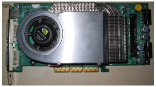
- Modern GPUs are streaming processors
  - Highly parallel: 512 cores 4 cores
  - Floating point optimized: 1345 GFLOPS 51.2 GFLOPS
  - Memory bandwidth optimized: 177 GB/s 32 GB/s
  - Cache and fast memory (configurable):
    2MB registers, 1MB L1+shared, L2, texture, constant 8MB L3

  More die space for number crunching
- Less die space for other stuff
  - No x86 translator, branch prediction, MMU, . . .
  - Number of program instructions limited: 2×10^6
  - Fixed amount of memory: 1.5GB up to 192GB

GPU clusters in Wuppertal

2004/5: 1×6800 Ultra

- First tests
- OpenGL 1.3, AGP Bus



2006: J/Ψ: 110×7900 GTX

- 1.6TFLOPS sustained (single)
- 256MB, OpenGL 2.0, PCIe 1.0

2007: uddn: 110×8800 GTX

- 3TFLOPS sustained (single)
- 512MB, CUDA 1.0, PCIe 1.0

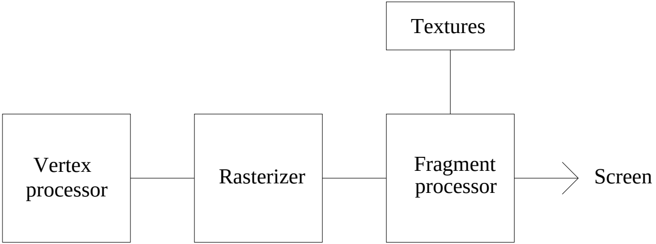
2008: uddn upgrade: 500×260 GTX

- 40TFLOPS sustained (single)
- 896MB, CUDA 1.3, PCIe 2.0

uddn: 350 TFLOPS peak, 40 TFLOPS sustained

Okt. 2008

GPU ARCHITECTURE

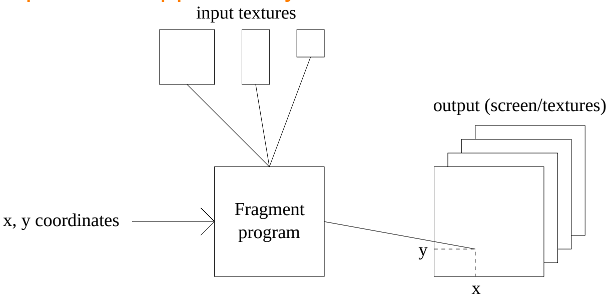


Elements of the pipeline:

- Vertex processor: basic transformations (e.g. rotations)
- Rasterizer: generate texture coordinates
- Fragment processor: shade a region using texture inputs

OLD PROGRAMING MODEL

OpenGL, supported by GeForce6+ cards



- Each pixel computation: incoming textures → outgoing textures: same operation on each pixel → massively parallel
- Like a stream: pipeline should be full → performance is better for large textures (large system size)

OPENGL BASICS

- Native data format:
  - 2D arrays (textures) (surfaces of polygons usually displayed on screen)
  - Each pixel (or fragment): 4 floating point numbers: RGBA colour channels (RGB, alpha)
  - Data types: half-float (16 bit), float (32 bit), more recently double (64 bit)
  - Everything done with floats (even indexing)

GENERAL SETUP

- Initialize graphics library (OpenGL)
- Set up rectangles (textures), that will contain data
- Write the shader program
  - Done in pseudo-assembler or Cg (C-like)
  - Compiled by driver during runtime
- Upload to GPU
- Run the shader (GL call)
- Download result textures

All this is done blindly (no printf in shader)

GL EXAMPLE

Real life example: $x_i = y_i + z_i \quad i = 1 \ldots 4nm$

Shader in Cg:

```
struct FragmentOut { float4 color0:COLOR0; };
FragmentOut example( in float2 myTexCoord:WPOS,
    uniform samplerRECT y, uniform samplerRECT z )
{
  FragmentOut c;
  c.color0 = texRECT( y, myTexCoord ) + texRECT( z, myTexCoord );
  return c;
}
```

EXAMPLE (ctd.)

Create texture in OpenGL (like malloc)

```
GLuint X;
glGenTextures( 1, &X);
glBindTexture( ..., X);
glTexParameteri(...);
glTexImage2D( ..., n, m, ..., 0 );   // also for y,z
```

Transfer data CPU → GPU in OpenGL

```
glBindTexture( ..., Y);
glFramebufferTexture2DEXT( ..., Y, ... );
glTexSubImage2D( ..., n, m, ..., y); // also for z
```

EXAMPLE (ctd.)

Do the actual computation, run the Cg shader

```
cgGLSetTextureParameter( ..., Y);
cgGLEnableTextureParameter( ..., "y" );    // also for z

glFramebufferTexture2DEXT( ..., X, 0 );
glDrawBuffer( ... );
cgGLBindProgram( ... );
glBegin( ... );
{
  glVertex2f( -n, -m);
  glVertex2f( n, -m);
  glVertex2f( n, m);
  glVertex2f( -n, m);
}
glEnd( );
```

EXAMPLE (ctd.)

Transfer data GPU → CPU in OpenGL

```
glFramebufferTexture2DEXT( ..., X, ... );
glReadBuffer( ... );
glReadPixels( ..., n, m, ..., x);
```

Same in C:

```
for( i = 0; i < 4 * n * m; i++ ) x[i] = y[i] + z[i];
```

GL PECULIARITIES

- Restriction on input textures:
  - ✗ maximum 32
  - ✗ read only
  - ✓ can be addressed indirectly (careful with performance!)
  - ✓ can have different sizes
- Restriction on output textures:
  - ✗ maximum 4 (only half Wilson vectors!)
  - ✗ write only
  - ✗ same sizes
  - ✗ location fixed per pixel
- Other features:
  - ✗ no branching
  - ✓ native vectors allow arbitrary “swizzle”

    `x=y.rbag+z.agrb` much better than SSE!

NEW PROGRAMMING MODEL

CUDA-C, supported by GeForce8+ cards

- General-purpose streaming language
  - Exact features supported depend on hardware!
- Minimal extension to C
  - Other language interfaces exist (C++, Fortran, OpenCL)
- Lifts OpenGL restrictions
  - Lots of possibilities to screw up performance

CUDA SOFTWARE STACK

3 components:

- Hardware driver
- API library and its runtime
- Some higher level math libraries CUFFT, CUBLAS