Lattice QCD on GPUs

Christian Hoelbling

Bergische Universität Wuppertal

Programming of Heterogeneous Systems in Physics
Jena, October 7, 2011
CONTENT

1. Overview
2. History
3. Lattice QCD
4. GPUs for lattice QCD
5. Current Implementation
6. Conclusions
PHYSICS GOALS

Goal:
- Solve QCD (theory of strong nuclear force)

Method:
- Lattice QCD
  - Discretize QCD on a space-time lattice
  - Numerically compute the path integral

Challenge:
- Massive numerical problem
  - Relatively straightforward to parallelize, vectorize
  - Mostly sparse matrix inversions
  - Why not put it on GPU’s?
THEORY: FLOPS

The graph shows the theoretical GFLOP/s performance over time for different NVIDIA GPUs and Intel CPUs. Key points include:

- NVIDIA GPU Single Precision:
  - GeForce GTX 480
  - GeForce GTX 280
  - GeForce 8800 GTX
  - GeForce 7800 GTX
  - GeForce 6800 Ultra
  - GeForce FX 5800

- NVIDIA GPU Double Precision:
  - Tesla C2050
  - Westmere

- Intel CPU Single Precision:
  - Woodcrest
  - Bloomfield

- Intel CPU Double Precision:
  - Harpertown

The graph illustrates a significant improvement in performance from September 2001 to December 2009, with NVIDIA GPUs showing a much steeper increase compared to Intel CPUs.
THEORY: MEMORY BANDWIDTH

Theoretical GB/s

- **CPU**
- **GPU**

- GeForce FX 5900
- GeForce 6800 GT
- GeForce 7800 GTX
- GeForce 8800 GTX
- GeForce GTX 285
- GeForce GTX 480
- Westmere
- Bloomfield
- Harpertown
- Woodcrest
- Prescott
- Northwood

Year:
- 2003
- 2004
- 2005
- 2006
- 2007
- 2008
- 2009
- 2010
GPUs give a factor $\sim 3$ boost compared to CPUs

→ 2-4 years ahead
SOME GPU HISTORY

Until mid 1990s: Framebuffer + BitBlock shifter
- 2D operations only
- CPU does all floating point work

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

Programmable shaders (← we entered here)
- 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, \( \sim 350 \)€

CPU: Intel Xeon W5590 (Nehalem), \( \sim 1400 \)€

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, \ldots
- Number of program instructions limited: \( 2 \times 10^6 \)
- 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: 400 TFLOPS peak, \( \sim 60 \) TFLOPS sustained

August 2011:

- 50 cluster nodes
  - Core i7
  - \( 2 \times \) C2050
  - QDR infiniband

- 150 farm nodes
  - Core2 Quad
  - \( 2 \times \) C1060
  - Ethernet
Elements of the pipeline:

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

OpenGL, supported by GeForce6+ cards

- Each pixel computation: incoming textures $\rightarrow$ outgoing textures: same operation on each pixel $\rightarrow$ massively parallel
- We hooked up the pixel shader
GL PECULIARITIES

✗ Restrictions on input textures (number, addressing, read only)
✗ Restrictions on output textures (number, addressing, write only)
✗ Initially no render to texture
✗ No branching
✗ Initially no double float and no IEEE
✓ Native vectors allow arbitrary “swizzle”
   \[ x = y \cdot rbag + z \cdot agrb \]  much better than SSE!
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:

```c
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)

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

Transfer data CPU → GPU in OpenGL

```c
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

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

Same in C:

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

- **GPU 8800 GTX**
- **GPU 7900 GTX**
- **GPU 7800 GTX**
- **CPU P4 SSE**

![Graph showing performance comparison between different GPUs and a CPU. The x-axis represents the volume (V) ranging from $8^4$ to $16^3 64$, and the y-axis represents Gflops ranging from 0 to 35.](image)
Currently best theory of strong interaction: QCD

Matter-fields: Quarks Interaction: Gluons

\[
\mathcal{L}_{\text{QCD}} = \overline{\Psi} D\Psi - \frac{1}{2} \text{Tr} (F^{\mu\nu} F_{\mu\nu}) \\
D = i\gamma^\mu (\partial_\mu + igA_\mu) - m \\
F_{\mu\nu} = \partial_\mu A_\nu - \partial_\nu A_\mu - g [A_\mu, A_\nu]
\]

Path integral = Partition function

\[
I = \int D[\overline{\Psi}, \Psi, A] e^{-\int_{\mathbb{R}^3 \times [0,t]} \mathcal{L}_{\text{QCD}}} \\
Z = \sum_i e^{-\frac{E_i}{k_B T}} = \sum_i e^{-E_i t} , \quad t = \frac{1}{k_B T}
\]
LATTICE DISCRETIZATION

- UV cutoff: space-time lattice
- Hypercubic, spacing $a$
- Momentum cutoff $p_{\mu} < \frac{2\pi}{a}$
- Continuum theory: $a \to 0$

- Anti-commuting $\psi(x)$ quark fields live on the sites
- Gluon fields $U_{\mu}(x) = U(x, x + e_{\mu})$ live on links

$$U(x, y) = \exp(ig \int_{x}^{y} dz_{\mu} A_{\mu}(z)) \in SU(3)$$

'Reverse' link: $U_{\mu}^\dagger(x + e_{\mu}) = U(x + e_{\mu}, x)$
GAUGE INVARIANT OBJECTS

Closed gluon loops:

\[ \text{Tr} \left( U_{\mu_0}(x) U_{\mu_1}(x + e_\mu) \ldots U_{\mu_n}(x) \right) \]

\( \bar{q} q \) connected by gluon lines:

\[ \bar{\psi}(x) U_{\mu_0}(x) \ldots U_{\mu_n}(y) \psi(y) \]
FERMION INTEGRATION

Full lattice QCD action:

\[ S = S_G + S_F \]

where the fermionic part in general is a Grassmann bilinear

\[ S_F = \bar{\Psi} M \Psi \]

discretized derivative

which can be integrated out formally

\[ Z = \int \prod_x \prod_{\mu} [dU_{\mu}(x)][d\bar{\psi}(x)][d\psi(x)] e^{-S_G - S_F} \]

\[ = \int \prod_x \prod_{\mu} [dU_{\mu}(x)] \det(M[U]) e^{-S_G} \]
FERMIONIC OBSERVABLES

\[ \mathcal{O}(x, y) = \mathcal{O}^\dagger(y)\mathcal{O}(x) = \left( \bar{\psi}^u \psi^d \right)_y \left( \bar{\psi}^d \psi^u \right)_x \]

fermionic operator

\[
\langle 0 | T(\mathcal{O}(x, y)) | 0 \rangle = \frac{1}{\mathcal{Z}} \int D\bar{\psi} D\psi DU \left( \bar{\psi}^u \psi^d \right)_y \left( \bar{\psi}^d \psi^u \right)_x e^{-S_G - S_F} \\
= \frac{1}{\mathcal{Z}} \int DU \text{Tr}_{c,s} \left( M^{-1,u}_{x,y}(U) M^{-1,d}_{y,x}(U) \right) \det(M[U]) e^{-S_G}
\]
We want to compute stochastically (on a finite lattice)

$$\langle 0|\mathcal{O}|0 \rangle = \frac{\sum_U \mathcal{O}_U e^{-S_U}}{\sum_U e^{-S_U}} \quad \text{with} \quad S_U = S_G - \ln \det(M[U])$$

Importance sampling:
- produce configuration $U_i$ weighted by $e^{-S_U}$
- Compute the sum $\langle 0|\mathcal{O}|0 \rangle = \frac{1}{N} \sum_i \mathcal{O}_i$

Simplest method: Metropolis algorithm
Choose an initial configuration $U_0$, then loop:
1. Generate $U_{k+1}$ from $U_k$ with a small random change
2. Measure the change $\Delta S$ in the action
3. If $\Delta S \leq 0$, keep $U_{k+1}$
4. If $\Delta S > 0$, keep $U_{k+1}$ with a probability of $e^{-\Delta S}$
NUMERICAL INTEGRATION

For each step we need to compute:

$$\exp(-\Delta S_g) \frac{\det(M(U_{k+1}))}{\det(M(U_k))}$$

- **Gauge part**: trace of $3 \times 3$ matrices
  - Local update possible
  - Low computational cost
- **Fermionic part**: det of large, usually sparse matrices
  - Only global update possible
  - Size of matrix: $\sim 10^6 \times 10^6 - 10^9 \times 10^9$
    - Dimension of $M$: $3$ (color) $\times$ $4$ (spin) $\times$ $N_x \times N_y \times N_z \times N_t$
    - Number of lattice points: $N_\mu = L_\mu / a$
      - $a \lesssim 0.1$ fm to resolve a proton
      - $N_\mu \gg 1$ fm to fit a proton
NUMERICAL SETUP

Global update: MD evolution in pseudofermion fields $\Phi$:

$$\det(M(U)) = \int D\Phi^\dagger D\Phi e^{\Phi^\dagger M^{-1}(U)\Phi}$$

Numerical problem: Inversion of the fermion matrix $M$

Properties of $M$:

- Large, sparse matrix inversion
- Krylov-space methods (conjugate gradient et. al.)
- Critical:
  - Fast matrix-vector multiplication
  - Fast linear algebra
MEMORY REQUIREMENTS

- Gauge update (relatively minor): $\sim 4$ flop/byte
  
  For each link (---) multiply staples (□□) and add them

- Pseudofermion update: $\sim 1$ flop/byte
  
  For each odd site (○) multiply links (---) on neighboring even sites (●) and add them

- Linear algebra: $<1$ flop/byte

- Typical GPU: $\sim 10$ flop/byte
Overview

History

Lattice QCD

GPUs for lattice QCD

Current Implementation

Conclusions

RELATIVE COST

One cpu core

Costs [eur/trajectory]

full trajectory [h]

other
link update
smearing
gauge force
mixed CG
multi CG

Physics is here

pion mass [MeV]

One cpu core

other
link update
smearing
gauge force
mixed CG
multi CG

Physics is here

30/47 Christian Hoelbling (Wuppertal)

Lattice QCD on GPUs
MEMORY CONSIDERATIONS

**Goal:**
- Maximize code performance

**Challenges:**
- Good performance for low flop/byte ratios
  - Reuse memory
  - Computations are essentially for free
- Problem is memory BW limited

**Strategies:**
- Compressed storage of gauge fields
- Try to reuse gauge fields
MAXIMIZING MEMORY BANDWIDTH

CPU:
- Linear access or intelligent prefetching

Tesla:
- Use textures for input (cached)
- Linear write to separate result buffer

Fermi:
- Use coalesced linear reads (e.g. float4)
- Extreme register pressure: use shared memory
- Best if shared memory is commonly used
EFFICIENT SHARED MEMORY USE

- Load links (—) into shared memory
- Do odd (○) → even (●)
- Do even (●) → odd (○)
- Concurrent: link only loaded once

Further improvements:

- Multiple RHSs concurrently: reuse link (currently 16 - half warp)
- Reuse cached vector components (sites): blocked site ordering
  Currently use 8 cubic neighbors at once (8 × 16 thread blocks)
- Combine linear algebra and inner products
  e.g. Combine result and search direction update in CG
How to best store gauge links:

- Gauge links are SU(3) matrices: 18 real numbers
- One row/column can be reconstructed by unitarity
  - 12 real numbers instead of 18
  - Reconstruct missing elements on the fly
  - Trade FLOPs for memory BW
- More aggressive: Gell-Mann basis $U = \exp(\sum_{i=1}^{8} c_i \lambda_i)$
  - 8 real numbers instead of 18
  - Matrix exponential on the fly
  - Danger: can be unstable for certain parameters
- We store 16 components (aligned memory access)
CUDA CC 1.0 BENCHMARK

Single-precision multishift solver

Gigaflops

Number of lattice sites

Geforce 470 GTX
Geforce 275 GTX
Geforce 260 GTX
Tesla C1060
Geforce 8800 GTX
COMPARISON TO CPU

✓ Massive speedup in CG solver and gauge force
✗ Other components become relevant
FURTHER OPTIMIZATION

Goal:
- Put everything on GPU

Challenges:
- More complex code parts (smearing, link updates)
  - Trigonometry, matrix exponentials
  - Double precision often required
- More work, less gain: still necessary

Things that must remain on host:
- Final crosscheck (GPU errors)
- Global sums, barriers and memory synchronization
Overview

History

Lattice QCD

GPUs for lattice QCD

Current Implementation

Conclusions

CC 1.3 code, Nvidia 260 GTX + AMD quad-core

mixed precision

other

link update

smearing

gauge force

mixed CG

multi CG

full trajectory [min]
pion mass [MeV]

CC 1.3 code, Nvidia 470 GTX

mixed precision

other

link update

smearing

gauge force

mixed CG

multi CG

full trajectory [min]
pion mass [MeV]

CC 1.3 code, Nvidia 260 GTX + AMD quad-core

IBM Cell BE 3.2 GHz (QPACE)

double precision

other

link update

smearing

gauge force

mixed CG

multi CG

full trajectory [min]
pion mass [MeV]

IBM Cell BE 3.2 GHz (QPACE)

bandwidth: one cell = 1/4 GPU

mixed precision

mixed precision

mixed precision

mixed precision

other

link update

smearing

gauge force

mixed CG

multi CG

full trajectory [min]
pion mass [MeV]

mixed precision

mixed precision

mixed precision

mixed precision

other

link update

smearing

gauge force

mixed CG

multi CG

full trajectory [min]
pion mass [MeV]
Use mixed precision inverters

- Invert to single precision
- Compute residue in double
- Invert remaining vector
- Can be refined (restart, target precision etc.)
- > 95% single operations
- Almost half memory bandwidth
Communicate boundaries:

\[ 8 \times N^3 \text{ for } N^4 \]

Surface/volume:

\[ S/V = 8/N \]

Need \( \sim 30 \text{Gb/s} \) network speed for maximal local lattice size
NETWORK

- 50 cluster nodes
  - Core i7
  - $2 \times$ C2050
  - QDR $\times$ 4 infiniband

- 36 port switches

- Real MPI bandwidth
  - $\sim$ 21 Gb/s on node
  - $\sim$ 16 Gb/s off node

- Real MPI latency
  - $\sim$ 27$\mu$s on node
  - $\sim$ 33$\mu$s off node

- Bandwidth-latency break even at $N \sim 16$
SCALING

2xGTX470+inf-QDRx4, CC 1.3 code

GFLOPS vs #GPUs graph with data points for different lattice sizes and number of GPUs.
RELIABILITY

- **Software**: need failsafe code
  - Matrix inversion: final residue check on CPU
  - Convergence in spite of intermittent error
  - Restart when error detected
  - Control part on CPU

- **Hardware**: Very different for different generations
  - GTX 7900: 80% fail after a year
  - GTX 260: 10% fail after a year
  - C1060: 2% fail after 3 months
  - GTX 4x0: “Thermi” better use TESLA
  - Underclock for increased lifetime
  - Don’t rely on ECC (it fails)
  - GPUs demand a good host! (especially mainboard)
  - Take care of warranty in contract!
CODE INVESTMENT

- We had 3 rewrites from scratch since 2005
  - OpenGL+CG
  - CUDA cc 1 (textures)
  - CUDA cc 2 (fermi)
- In addition many updates
  - render to texture, IEEE, double, multi-GPU, device-to-device, in-line ptx, ...
- Moving target: no hope for future-proof code
  - This is good! No development → no speedup
- Why not openCL?
  - CUDA currently faster
  - Similar (low level) syntax
  - nvcc compiles openCL via intermediate CUDA
  - Vendor lock in - ATI-openCL is getting better
SUMMARY

- GPUs are useful for lattice QCD
  - Large, relatively simple problem
  - Error-catching possible (cards do fail)
  - Memory bandwidth limited problem
- GPUs are now mainstream in lattice QCD
  - Many large installations worldwide
  - Most HW/SW talks at lattice conference about GPUs
  - Free library: QUDA
- Limitations
  - Need large local lattice
  - Parallelization difficult
  - Few registers/cache

✔ Very useful in farming mode (certain problems)

✖ Not useful for single, large problems
OUTLOOK

Will future GPUs be competitive for lattice QCD?

- Competition: highly scaleable machines (e.g. bluegene)
  - ✓ Small local lattice
  - ✓ Everything runs in-cache
  - ✓ Fast neighbor communication
  - ➔ Just add more processors for weak scaling

- Future GPUs need either
  - ✏ substantially larger memory bandwidth or
  - ✏ substantially larger caches or
  - ✏ substantially faster communication

to stay competitive
ONE PHYSICS RESULT

QCD crossover temperature

T[MeV]

Δl,s

Nt=8
Nt=10
Nt=12
Nt=16

p4 Nt=8
Asqtad Nt=8
Asqtad Nt=12
Hisq Nt=8