cuLGT: Lattice Gauge Fixing on GPUs
Landau, Coulomb and Maximally Abelian gauge

10.09.2014, Hannes Vogt, Mario Schröck (INFN Roma Tre)
Motivation

- QCD action is gauge invariant
- Fixing a gauge is necessary in the continuum
- Gauge-variant two point functions play an important role in confinement scenarios

- On the lattice:
  - high dimensional optimization problem
  - many local maxima: Gribov copies
  - algorithms are nicely parallelizable

- cuLGT
  - CUDA-based code for gauge fixing and more
  - code is publicly available: www.culgt.com
Outline

- Gauge fixing on the lattice
- Standard optimizations
- More optimization: 8 threads per site
- Performance of cuLGT1
- Improvements: cuLGT2
Gauge fixing on the lattice: Landau gauge

- Action is invariant under gauge transformation of the links
  
  \[ U_\mu(x) \rightarrow U_\mu^g(x) = g(x)U_\mu(x)g^\dagger(x + \hat{\mu}) \]

- Landau gauge \((\partial_\mu A_\mu(x) = 0)\) is achieved by maximizing

  \[
  F^U[g] = \frac{1}{N_d N_c V} \sum_x \sum_\mu \text{Re} \text{tr} \left[ g(x)U_\mu(x)g^\dagger(x + \hat{\mu}) \right]
  \]
Gauge fixing on the lattice: Landau gauge

- Action is invariant under gauge transformation of the links

\[ U_\mu(x) \rightarrow U_\mu^g(x) = g(x)U_\mu(x)g^\dagger(x + \hat{\mu}) \]

- Landau gauge \((\partial_\mu A_\mu(x) = 0)\) is achieved by maximizing

\[
F^U[g] = \frac{1}{2N_dN_cV} \sum_x \text{Re} \text{ tr} [g(x)K(x)]
\]
Gauge fixing on the lattice: Landau gauge

- Action is invariant under gauge transformation of the links
  \[ U_\mu(x) \rightarrow U_\mu^g(x) = g(x)U_\mu(x)g^\dagger(x + \hat{\mu}) \]
- Landau gauge \((\partial_\mu A_\mu(x) = 0)\) is achieved by maximizing
  \[ F^U[g] = \frac{1}{2N_dN_cV} \sum_x \text{Re tr} [g(x)K(x)] \]
  - optimize locally:
    \[ g(x)K(x) = g(x) \sum_\mu \left[ U_\mu(x)g(x + \hat{\mu})^\dagger + U_\mu(x - \hat{\mu})^\dagger g(x - \hat{\mu})^\dagger \right] \]
  - SU(2): optimum for \(g(x) = K^\dagger(x)/\det(K^\dagger)\)
    for \(N_c > 2\) iterate over SU(2) subgroups
Gauge fixing on the lattice: Landau gauge

update:
- \( U_\mu(x) \rightarrow g(x) U_\mu(x) \)
- \( U_\mu(x - \hat{\mu}) \rightarrow U_\mu(x - \hat{\mu}) g(x) \dagger \)
- ...

iterate until:
\[
\theta \approx \max_x |\partial_\mu A_\mu(x)| < \epsilon
\]

- overrelaxation: \( g(x) \rightarrow g(x)^\omega, \omega \in [1, 2) \)
- simulated annealing: \( dP(g) = e^{-\frac{1}{T} \text{Re} tr g(x) K(x)} dg \)
Pseudocode

Algorithm 1

while precision $\theta$ not reached do
    for sublattice = even, odd do
        for all $x$ of sublattice do
            for all SU(2) subgroups do
                local optimization: find $g(x) \in$ SU(2)
                which is a function of $U_\mu(x), U_\mu(x - \hat{\mu})$
                for all $\mu$ do
                    apply $g(x)$ to $U_\mu(x), U_\mu(x - \hat{\mu})$
                end for
            end for
        end for
    end for
end while
Standard optimizations for GPUs

- reorder memory layout for coalescing:
  - memory split into even/odd sublattice

Standard optimizations for GPUs

- reorder memory layout for coalescing:
  - memory split into even/odd sublattice
  - site index running faster than color/Dirac indices

- code is memory bound:
  - store only parts of the \(N_c \times N_c\) matrix in global memory
  - reconstruct in registers using SU(\(N_c\)) properties
  - SU(3): 12 parameters instead of 18

- use texture cache to load links

- our standard setup:
  - GTX 580 (Fermi)
  - 32 \(4^4\) SU(3) lattice in SP
  - Landau gauge, Overrelaxation

Standard optimizations for GPUs

- reorder memory layout for coalescing:
  - memory split into even/odd sublattice
  - site index running faster than color/Dirac indices
- code is memory bound:
  - store only parts of the $N_c \times N_c$ matrix in global memory
  - reconstruct in registers using SU($N_c$) properties
  - SU(3): 12 parameters instead of 18
Standard optimizations for GPUs

- reorder memory layout for coalescing:
  - memory split into even/odd sublattice
  - site index running faster than color/Dirac indices
- code is memory bound:
  - store only parts of the $N_c \times N_c$ matrix in global memory
  - reconstruct in registers using SU($N_c$) properties
  - SU(3): 12 parameters instead of 18
- use texture cache to load links

\[ \text{M. A. Clark et al.,} \]
\[ \text{Comput. Phys. Commun. 181 (2010)} \]
Standard optimizations for GPUs

- reorder memory layout for coalescing:
  - memory split into even/odd sublattice
  - site index running faster than color/Dirac indices
- code is memory bound:
  - store only parts of the $N_c \times N_c$ matrix in global memory
  - reconstruct in registers using SU($N_c$) properties
  - SU(3): 12 parameters instead of 18
- use texture cache to load links

- our standard setup:
  - GTX 580 (Fermi)
  - $32^4$ SU(3) lattice in SP
  - Landau gauge, Overrelaxation
Performance: Overrelaxation

![Graph showing performance comparison between CPU and first version.](image)

Sustained GFlops vs. Throughput [% of max.]

- (CPU)
- first version

8 | Hannes Vogt - cuLGT: Lattice Gauge Fixing on GPUs
© 2014 University of Tuebingen
Performance limit: register spilling

- at each site: 8 links * 18 parameters = 144 floats
- Fermi: only 63 32-bit registers/kernel
- kernel signature: 63 registers, 512 Byte stack frame, 964 Byte spill stores, 1232 Byte spill loads

Solution:
- 8 threads per site (1 thread per link)
  - only 18 floats (registers) for the link
  - communication via shared memory
    - in practice: 34 registers (no register spilling)
    - optimized for higher occupancy: 32 register
Performance: Overrelaxation

![Graph showing performance comparison between CPU, first version, and cuLGT1 with sustained GFlops on the y-axis and throughput [% of max.] on the x-axis.]

(CPU) first version cuLGT1

Sustained GFlops

Throughput [% of max.]
Performance: Overrelaxation

Sustained GFlops vs. $N_t$

- SP
- MP
- DP

$N_s^3 = 24^3$
$N_s^3 = 20^3$
$N_s^3 = 16^3$
Conservation of unitarity
cuLGT2

Problems of cuLGT1

- structural problems
  - code evolved over time
  - readability/maintainability vs. performance
  - not well modularized
  - compile time constants
  - two codes for SU(2) and SU(3)

- performance obstacles
  - parameterization of links hardcoded
  - not easy to switch to 4 threads/site
  - no autotune possibility

⇒ time for a complete makeover
cuLGT2: Link parameterization

- in cuLGT1:
  - large array of float/double variables
  - `GlobalLink<Pattern>` \( \text{link}(x, \mu) \sim \text{array} \)
  - `LocalLink` \( \text{link} \sim 3 \times 3 \text{ complex matrix} \)
- links \( U_\mu(x) \) are optimally coalesced, neighbours \( U_\mu(x - \hat{\mu}) \) not!
  - float4/double2 would be better for non-coalesced loads

- solution: new layer of abstraction
  - parameterization: choose datatype & number of elements
  - examples for SU(3):
    - \( 12 \times \text{float} \)
    - \( 3 \times \text{float4} \)
    - \( 9 \times \text{Complex<float>} \)
    - \( 18 \times \text{float} \)
cuLGT2: an example

typedef LocalLink<SUNRealFull<3,float>> LOCALLINK;

typedef SU3Vector4<float> PARAMTYPE;
typedef GPUPattern<SITETYPE,PARAMTYPE> PATTERNTYPE;
typedef GlobalLink<PATTERNTYPE,USETEXTURE> GLOBALLINK;

__global__ polyakovLoop( float4* U, float* polyakov, LatDim<4> dim )
{
    SITETYPE site( blockIdx.x * blockDim.x + threadIdx.x );
    LOCALLINK linkProduct; linkProduct.identity();
    for( int t = 0; t < dim.getDimension( TDIR ); t++ )
    {
        GLOBALLINK glob( U, site, TDIR );
        LOCALLINK link = glob;
        linkProduct *= link;
        site.setNeighbour( TDIR );
    }
    polyakov[site.getIndex()] = linkProduct.retrace();
}
cuLGT2: an example

typedef LocalLink<SUNRealFull<3,float> > LOCALLINK;

typedef SU3Vector4<float> PARAMTYPE;
typedef GPUPattern<SITETYPE,PARAMTYPE> PATTERNTYPE;
typedef GlobalLink<PATTERNTYPE,USETEXTURE> GLOBALLINK;

__global__ polyakovLoop(...)
{
    int main()
    {
        LatDim<4> dim( Nt, Nx, Ny, Nz );
        GaugeConfiguration<PATTERN> config( dim );
        config.allocateMemory();
        // allocate dPlaquette, load the configuration to host memory
        config.copyToDevice();

        GLOBALLINK.bindTexture( config.getDevicePointer(), config.getSize() );
polyakovLoop<<<GRIDSIZE,BLOCKSIZE>>>( config.getDevicePointer(),
            dPolyakov, dim );

        reduction( dPolyakov );
    }
cuLGT2: what do we have now?

- Flexible datastructure
  - easily switch from SU(3) to SU(2)
  - easily try different optimized parameterizations
- Modularized:
  - MILC interface
- Autotune utility selects optimal setup for different architectures
  - 4 or 8 threads per site
  - different block sizes
- Unit tests

- not yet: MultiGPU

- Performance
Performance: Overrelaxation

![Graph showing performance boosts for CPU, cuLGT1, and cuLGT2 over successive versions.](image)
cuLGT2: Performance on different GPUs

![Graph showing performance comparison of different GPUs (GTX 580, GTX 680, GTX Titan, Tesla K20s) for single precision (SP), multi precision (MP), and double precision (DP) calculations.](image-url)
Summary: What does cuLGT offer?

- **gauge types:**
  - Landau gauge
  - Coulomb gauge
  - Maximally Abelian gauge

- **algorithms:**
  - Overrelaxation
  - Simulated Annealing

- **groups:**
  - SU(2)
  - SU(3)
  - SU(N) for Landau/Coulomb gauge easy to implement

- **MultiGPU:** only Landau gauge

- **integration in other frameworks:** MILC
Thank you.

Contact:

Faculty of Science
Institute for Theoretical Physics
Auf der Morgenstelle 14
72076 Tübingen

hannes.vogt@uni-tuebingen.de
8 threads/site: sketch

<table>
<thead>
<tr>
<th>LinkID</th>
<th>ThreadID</th>
<th>$U_t(0)$</th>
<th>$U_t(1)$</th>
<th>$U_t(2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0...31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>32...63</td>
<td>$U_t(0 - \hat{t})$</td>
<td>$U_t(1 - \hat{t})$</td>
<td>$U_t(2 - \hat{t})$</td>
</tr>
<tr>
<td>2</td>
<td>64...95</td>
<td>$U_x(0)$</td>
<td>$U_x(1)$</td>
<td>$U_x(2)$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>224...255</td>
<td>$U_z(0 - \hat{z})$</td>
<td>$U_z(1 - \hat{z})$</td>
<td>$U_z(2 - \hat{z})$</td>
</tr>
</tbody>
</table>
8 threads/site: Pseudocode

Algorithm 2 eight-threads-per-site kernel

__shared__ g
loadAndReconstructLink( x, μ ) → U
for all SU(2) subgroups do
    getSubgroup( U, i ) → u
    atomicAdd( K, u )
__syncthreads()
if LINK_ID == 0 then
    g† / det(g†) → g
end if
__syncthreads()
update(U,g)
end for
saveLink( U )
MultiGPU

• split the lattice along the time direction:
  - $N_t/nprocs$ timeslices per GPUs
  - temporal links of the neighbouring timeslice at boundary
• data layout: timeslices split in parity
• asynchronous memory transfer (MPI)
  - update inner timeslices
  - asynchronously copy boundary timeslice, update when finished
  - (only inactive parity needs to be copied)
• $\sim 8$ timeslices per GPU to hide data exchange
MultiGPU: strong scaling

on Tesla C2070: $64^3 \times N_t$ with $N_t = 256, 128, 96$ (top to bottom)
Simulated Annealing

- used to find the global maximum
- propose a gauge transformation in the heatbath of surrounding links $dP(g) = e^{-\frac{1}{T}\text{Re} tr g(x) K(x)} dg$

The relative deviation from the maximal gauge functional.

l.h.s.: 10000, r.h.s. no simulated annealing steps.
Conservation of the plaquette

![Graph showing the conservation of the plaquette](chart)

**Number of Iterations**

**SP**, **SP reproj. 100**, **MP**, **MP reproj. 100**, **DP**