Gauge Fixing in Lattice QCD on Multi-GPUs

Mario Schröck

in collaboration with Hannes Vogt (Uni Tübingen)

arXiv:1212.5221

Bjelasnica Mountain, Sarajevo
February 04, 2013
Outline

• Motivation

• Brief introduction to GPU computing with CUDA

• Lattice gauge fixing on the GPU
  • overrelaxation
  • simulated annealing

• Multi-GPUs and scaling

• Summary
Motivation

• First one Teraflops sustained performance in QCD in 2004 (P. Vranas):
• one Blue Gene/L rack
• price > 500,000 €

Flops: floating point operations per second
Motivation

• First one Teraflops sustained performance in QCD in 2004 (P. Vranas):
  • one Blue Gene/L rack
  • price > 500,000 €

Flops: floating point operations per second
Motivation

- First one Teraflops sustained performance in QCD in 2004 (P. Vranas):
  - one Blue Gene/L rack
  - price > 500,000 €

*Flops*: floating point operations per second
Motivation

- First one Teraflops sustained performance in QCD in 2004 (P. Vranas):
  - one Blue Gene/L rack
  - price > 500,000 €

- Desktop PC
  - 3 NVIDIA GTX 580 GPUs
  - one Teraflop sustained performance
  - price ~ 4,000 €

Flops: floating point operations per second
The CUDA Programming Model

- Program is executed on host system (CPU)
- host calls kernels that run on the device (GPU)
- each kernel starts many threads that perform the same work on different data
  - e.g. one thread per lattice site
Lattice gauge fixing

- Gauge freedom
  \[ g(x)U_\mu(x)g(x + \hat{\mu})^\dagger \]

- Gauge fixing, e.g. the Landau gauge condition
  \[ \partial_\mu A_\mu(x) = 0 \]

which translates to a large scale optimization problem, on the lattice

\[ F_{\text{Landau}}^g[U] = \frac{1}{N_cN_dV} \text{Re} \sum_{\mu,x} \text{tr} [U_\mu^g(x)] \rightarrow \text{max.} \]

until

\[ \theta = \frac{1}{N_cV} \sum_x \text{tr} [\Delta^g(x)\Delta^g(x)^\dagger] \]

is sufficiently small.
Relaxation: optimize locally

- the idea of the relaxation algorithm is to iterate over the lattice site by site and to maximize the local gauge functional, i.e., maximize

\[
 f^g_{\text{Landau}}(x) = \Re \text{ tr } [g(x)K(x)]
\]

with

\[
 K(x) := \sum_{\mu} \left( U_\mu(x)g(x + \hat{\mu})^\dagger + U_\mu(x - \hat{\mu})^\dagger g(x - \hat{\mu})^\dagger \right)
\]
Relaxation: optimize locally

- The idea of the relaxation algorithm is to iterate over the lattice site by site and to maximize the local gauge functional, i.e., maximize

\[ f^g_{\text{Landau}}(x) = \Re \text{ tr } [g(x)K(x)] \]

with

\[ K(x) := \sum \left( U_\mu(x)g(x + \hat{\mu})^\dagger + U_\mu(x - \hat{\mu})^\dagger g(x - \hat{\mu})^\dagger \right) \]

- For SU(2) the maximum is directly given by

\[ g(x) = K(x)^\dagger / \sqrt{\det K(x)^\dagger} \]

and for SU(3) we iterate through the SU(2) subgroups and thereby maximize the local gauge functional.
Variations of the relaxation update

- **overrelaxation**: replace $g(x)$ by $g^\omega(x)$, $\omega \in [1, 2]$
- **stochastic relaxation**: replace $g(x)$ by $g^2(x)$ with probability $p$
- **simulated annealing** (Kirkpatrick et al., Science 330 (1983)):
  - motivated by annealing of metal in condensed matter physics
  - SA temperature $T$ decreases with time (cooling)
  - a new randomly chosen gauge transformation is accepted with probability

$$P[g(x)] = \begin{cases} 
1 & \text{if } f^g(x) \geq f(x) \\
\exp\left(\frac{f^g(x) - f(x)}{T}\right) & \text{else}.
\end{cases}$$

- this allows for worsening of the function that is to be optimized at high temperatures, can escape from local maxima
- in the limit of infinite time SA is guaranteed to converge to the global maximum
Simulated annealing: temperature dependence

- Landau and maximally Abelian gauge (MAG) functionals as a function of the simulated annealing temperature.
Simulated annealing: towards the global max.

Distribution of the gauge functional values of 100 gauge copies: relative deviation from the maximum found

- without simulated annealing
- 3000 simulated annealing steps
- 10000 simulated annealing steps
The algorithm in detail

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 $g(x) \in \text{SU}(2)$
        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 while
The algorithm in detail

Algorithm 1

\[
\textbf{while} \quad \text{precision} \ \theta \ \text{not reached} \quad \textbf{do} \\
\quad \text{for} \quad \text{sublattice} = \text{even, odd} \quad \textbf{do} \\
\quad \quad \text{for all} \quad x \quad \text{of sublattice} \quad \textbf{do} \\
\quad \quad \quad \text{for all} \quad \text{SU}(2) \quad \text{subgroups} \quad \textbf{do} \\
\quad \quad \quad \quad \text{local optimization: find} \quad g(x) \in \text{SU}(2) \\
\quad \quad \quad \quad \quad \text{which is a function of} \quad U_\mu(x), \ U_\mu(x - \hat{\mu}) \\
\quad \quad \quad \quad \quad \text{for all} \quad \mu \quad \textbf{do} \\
\quad \quad \quad \quad \quad \quad \text{apply} \quad g(x) \quad \text{to} \quad U_\mu(x), \ U_\mu(x - \hat{\mu}) \\
\quad \quad \quad \quad \quad \textbf{end for} \\
\quad \quad \textbf{end for} \\
\quad \textbf{end for} \\
\textbf{end for} \\
\textbf{end while}
\]
Run the algorithm on the device

```c
__global__ relaxKernel( Real* U )
{
    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
}
```
Run the algorithm on the device

```c
__global__ relaxKernel( Real* U )
{
    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
}
```
Run the algorithm on the device

```c
__global__ relaxKernel( Real* U )
{
    for all SU(2) subgroups do
        local optimization: find \( g(x) \in SU(2) \)
            which is a function of \( U_\mu(x), U_\mu(x - \mu) \)
        for all \( \mu \) do
            apply \( g(x) \) to \( U_\mu(x), U_\mu(x - \mu) \)
        end for
    end for
}
```

- instead of looping over all lattice size
- start the kernel for a grid of thread blocks

`relaxKernel<<<V/32,32>>>(U);`
Run the algorithm on the device

```c
__global__ relaxKernel( Real* U )
{
    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
}
```

- instead of looping over all lattice size
- start the kernel for a grid of thread blocks

```
relaxKernel<<<V/32,32>>>(U);
```

Figure 1: Block Diagram of Code Execution on Multi-GPUs
Coalesced memory accesses

• a warp (group of 32 threads) reads blocks of 128kB from global memory at once:

• we want to serve all 32 threads (think of sites) with that read access

• normal storage layout has the whole SU(3) matrix as one block in memory, we need a modified memory pattern:
Coalesced memory accesses

- a warp (group of 32 threads) reads blocks of 128kB from global memory at once:

- we want to serve all 32 threads (think of sites) with that read access

- normal storage layout has the whole SU(3) matrix as one block in memory, we need a modified memory pattern:

  - **StandardPattern** (natural layout): $t, x, y, z, \mu, i, j, c$
  
  - **GpuPattern**: $\mu, i, j, c, p, [t, x, y, z]_p$
  
  - **TimesliceGpuPattern**: $t, \mu, i, j, c, p, [x, y, z]_p$

- $x, y, t, z$ is the space-time index, $\mu$ the Dirac index, $i$ and $j$ row and column index of the matrix and $p$ is parity.
Optimizations

- most lattice QCD kernels are bound by the bandwidth to global memory instead of by the theoretical peak performance (GFlops)
Optimizations

• most lattice QCD kernels are bound by the bandwidth to global memory instead of by the theoretical peak performance (GFlops)

• transfer only 12 parameter of each SU(3) matrix and recalculate the matrix when needed (virtually for free!)
Optimizations

• most lattice QCD kernels are bound by the bandwidth to global memory instead of by the theoretical peak performance (GFlops)

• transfer only 12 parameter of each SU(3) matrix and recalculate the matrix when needed (virtually for free!)

• the max. number of registers per thread are not sufficient: the result are register spills to local memory
Optimizations

- most lattice QCD kernels are bound by the bandwidth to global memory instead of by the theoretical peak performance (GFlops)

- transfer only 12 parameters of each SU(3) matrix and recalculate the matrix when needed (virtually for free!)

- the max. number of registers per thread are not sufficient: the result are register spills to local memory

- we avoid this by assigning eight threads to each lattice site: each of the eight threads takes care of one of the eight neighbor links and therefore the number of registers is sufficient
Optimizations

- most lattice QCD kernels are bound by the bandwidth to global memory instead of by the theoretical peak performance (GFlops)

- transfer only 12 parameters of each SU(3) matrix and recalculate the matrix when needed (virtually for free!)

- the max. number of registers per thread are not sufficient: the result are register spills to local memory

- we avoid this by assigning eight threads to each lattice site: each of the eight threads takes care of one of the eight neighbor links and therefore the number of registers is sufficient

- more tuning: setting launch bounds to the kernels, prefer L1 cache over shared memory, compiler flag for non-caching loads, use_fast_math flag
Performance on different devices

- DP
- MP
- SP
Performance on different devices

- GTX 480
- GTX 580
- Tesla C2070
- Quadro 4000

DP, MP, SP

Sustained GFlops
Comparison to the CPU

- we compare our performance to FermiQCD run on a Intel Xeon Six-Core CPU X5650 ("Westmere") @ 2.67GHz with MPI

- FermiQCD
- Tesla C2070
- GTX 580
Comparison to the CPU

- we compare our performance to FermiQCD run on a Intel Xeon Six-Core CPU X5650 (“Westmere”) @ 2.67GHz with MPI

![Bar chart comparing FermiQCD, Tesla C2070, and GTX 580 to FermiQCD]
Multi-GPUs
Multi-GPUs

- we split the lattice along the temporal direction and distribute it to multiple GPUs
Multi-GPUs

- we split the lattice along the temporal direction and distribute it to multiple GPUs

During each step of the iteration, the data at the boundaries have to be exchanged. On each device:

- send $U_0(t_{\text{max}})$ from device $i$ to $i+1$
- perform the update
- send back
Data exchange between devices

• to be more precise, each device has carry out the following instructions:

1. `cudaMemcpyDeviceToHost` of $U_0(t_{\text{max}})$ (inactive parity)
2. `MPI_Send` of $U_0(t_{\text{max}})$ to process with $i+1$ and `MPI_Recv` of $U_0(t_{\text{min}}-1)$ from process with $i-1$
3. `cudaMemcpyHostToDevice` of $U_0(t_{\text{min}}-1)$
4. update $U_\mu(t_{\text{min}})$ (active parity)
5. `cudaMemcpyDeviceToHost` of $U_0(t_{\text{min}}-1)$ (inactive parity)
6. `MPI_Send` of $U_0(t_{\text{min}}-1)$ to process with $i$ and `MPI_Recv` of $U_0(t_{\text{max}})$ from process with $i+1$
7. `cudaMemcpyDeviceToHost` of $U_0(t_{\text{max}})$
Data exchange between devices

• to be more precise, each device has carry out the following instructions:

1. `cudaMemcpyDeviceToHost` of $U_0(t_{\text{max}})$ (inactive parity)
2. `MPI_Send` of $U_0(t_{\text{max}})$ to process with $i+1$ and `MPI_Recv` of $U_0(t_{\text{min}} - 1)$ from process with $i - 1$
3. `cudaMemcpyHostToDevice` of $U_0(t_{\text{min}} - 1)$
4. update $U_\mu(t_{\text{min}})$ (active parity)
5. `cudaMemcpyDeviceToHost` of $U_0(t_{\text{min}} - 1)$ (inactive parity)
6. `MPI_Send` of $U_0(t_{\text{min}} - 1)$ to process with $i$ and `MPI_Recv` of $U_0(t_{\text{max}})$ from process with $i + 1$
7. `cudaMemcpyDeviceToHost` of $U_0(t_{\text{max}})$

• the copies between host and device are very slow

• idea: overlap the data exchange with calculations in the inner part of the domain

• i.e., each of the six exchange steps is “buffered” with the asynchronous update of some of the other time-slices on the corresponding device.
Data exchange vs. inner calculations

• compare the time for the update of one time-slice to the time for a device-to-host (D2) copy and a host-to-device copy (H2D):
Data exchange vs. inner calculations

- compare the time for the update of one time-slice to the time for a device-to-host (D2) copy and a host-to-device copy (H2D):

<table>
<thead>
<tr>
<th>$N_s^3$</th>
<th>D2H [$\mu$s]</th>
<th>H2D [$\mu$s]</th>
<th>kernel [$\mu$s]</th>
<th>D2H/kernel</th>
<th>H2D/kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0.0398</td>
<td>0.0368</td>
<td>0.0209</td>
<td>1.90</td>
<td>1.76</td>
</tr>
<tr>
<td>32</td>
<td>0.2543</td>
<td>0.2276</td>
<td>0.1443</td>
<td>1.76</td>
<td>1.58</td>
</tr>
<tr>
<td>64</td>
<td>1.2510</td>
<td>1.1830</td>
<td>1.0489</td>
<td>1.19</td>
<td>1.13</td>
</tr>
<tr>
<td>128</td>
<td>8.9597</td>
<td>8.7169</td>
<td>8.3041</td>
<td>1.08</td>
<td>1.05</td>
</tr>
</tbody>
</table>
Data exchange vs. inner calculations

• compare the time for the update of one time-slice to the time for a device-to-host (D2) copy and a host-to-device copy (H2D):

<table>
<thead>
<tr>
<th>$N^3_s$</th>
<th>D2H [$\mu$s]</th>
<th>H2D [$\mu$s]</th>
<th>kernel [$\mu$s]</th>
<th>D2H/kernel</th>
<th>H2D/kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>0.0398</td>
<td>0.0368</td>
<td>0.0209</td>
<td>1.90</td>
<td>1.76</td>
</tr>
<tr>
<td>32</td>
<td>0.2543</td>
<td>0.2276</td>
<td>0.1443</td>
<td>1.76</td>
<td>1.58</td>
</tr>
<tr>
<td>64</td>
<td>1.2510</td>
<td>1.1830</td>
<td>1.0489</td>
<td>1.19</td>
<td>1.13</td>
</tr>
<tr>
<td>128</td>
<td>8.9597</td>
<td>8.7169</td>
<td>8.3041</td>
<td>1.08</td>
<td>1.05</td>
</tr>
</tbody>
</table>

• we conclude that two time-slices per copy step (the six steps from before) are enough to hide the time that is spent for communications
Weak scaling

- keep the lattice volume per GPU fixed (here $64^3 \times 32$ and $48^4$)
Weak scaling

- keep the lattice volume per GPU fixed (here $64^3 \times 32$ and $48^4$)
Strong scaling

- keep the total lattice volume fixed (here $64^3 \times 256, 128, 96$)
Strong scaling

• keep the total lattice volume fixed (here $64^3 \times 256, 128, 96$)
Summary and outlook

• the combination of overrelaxation, stochastic relaxation and simulated annealing is well suited to fix the gauge on the lattice to, e.g., Coulomb, Landau or the maximally Abelian gauge

• GPUs offer a very good price to performance ratio

• the adoption of multi-GPUs overcomes the memory constraint of single GPUs

• we showed linear scaling on 16 GPUs on lattices of size $64^3 \times 256$ and larger

• we are currently applying our code to
  • the calculation of propagators in the MA/U(1)xU(1) gauge in SU(3)
  • collecting high statistics for the distribution and number of Gribov copies in compact U(1) with more than one billion gauge copies per gauge orbit

• our code is available for download under

  www.cuLGT.com
Thank you!