# 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

UNI GRAZ

ITN STRONGnet

# 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 €

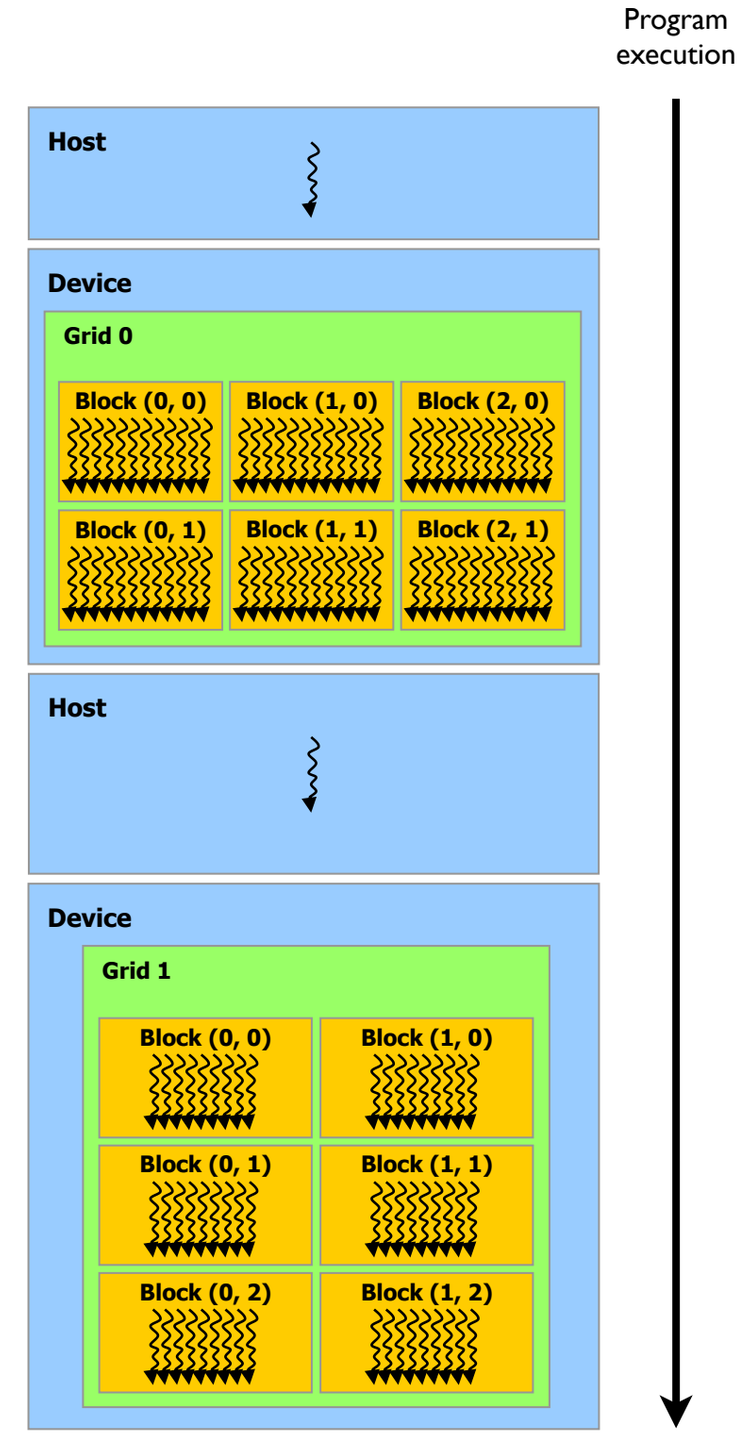
*Flops:* floating point operations per second

# 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

$$\partial_\mu A_\mu(x) = 0$$

translates to a large scale optimization problem, on the lattice

$$F^g_{\text{Landau}}[U] = \frac{1}{N_c N_d V}\mathfrak{Re}\sum_{\mu,x}\text{tr}\left[U^g_\mu(x)\right] \longrightarrow \text{max.}$$

until

$$\theta \equiv \frac{1}{N_c V}\sum_x \text{tr}\left[\Delta^g(x)\Delta^g(x)^\dagger\right]$$

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

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

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

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

with

$$K(x) := \sum_\mu \left( U_\mu(x) g(x+\hat{\mu})^\dagger + U_\mu(x-\hat{\mu})^\dagger g(x-\hat{\mu})^\dagger \right)$$

• 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.

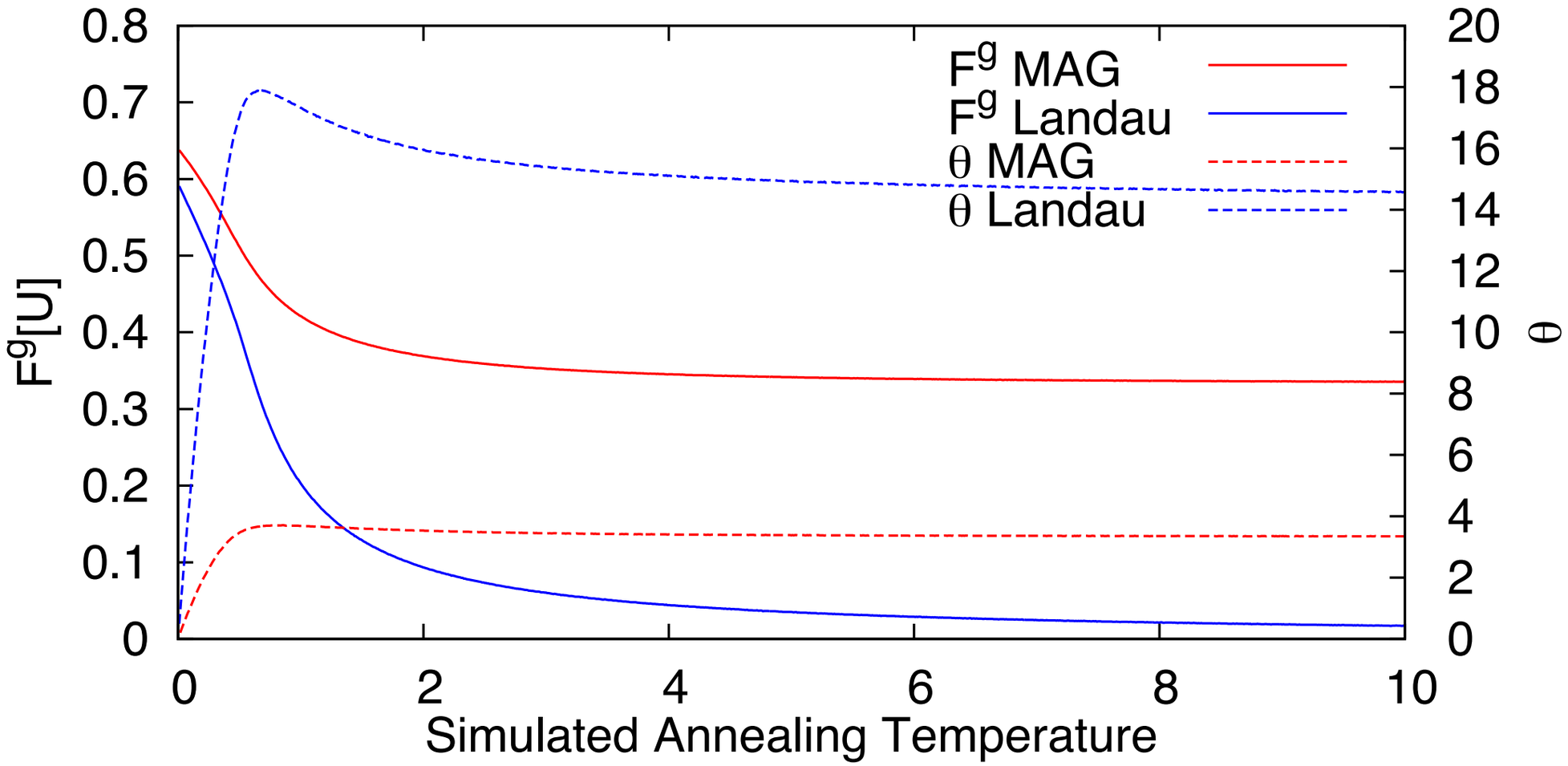
# Variations of the relaxation update

- 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 trafo is accepted with probability

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

- 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

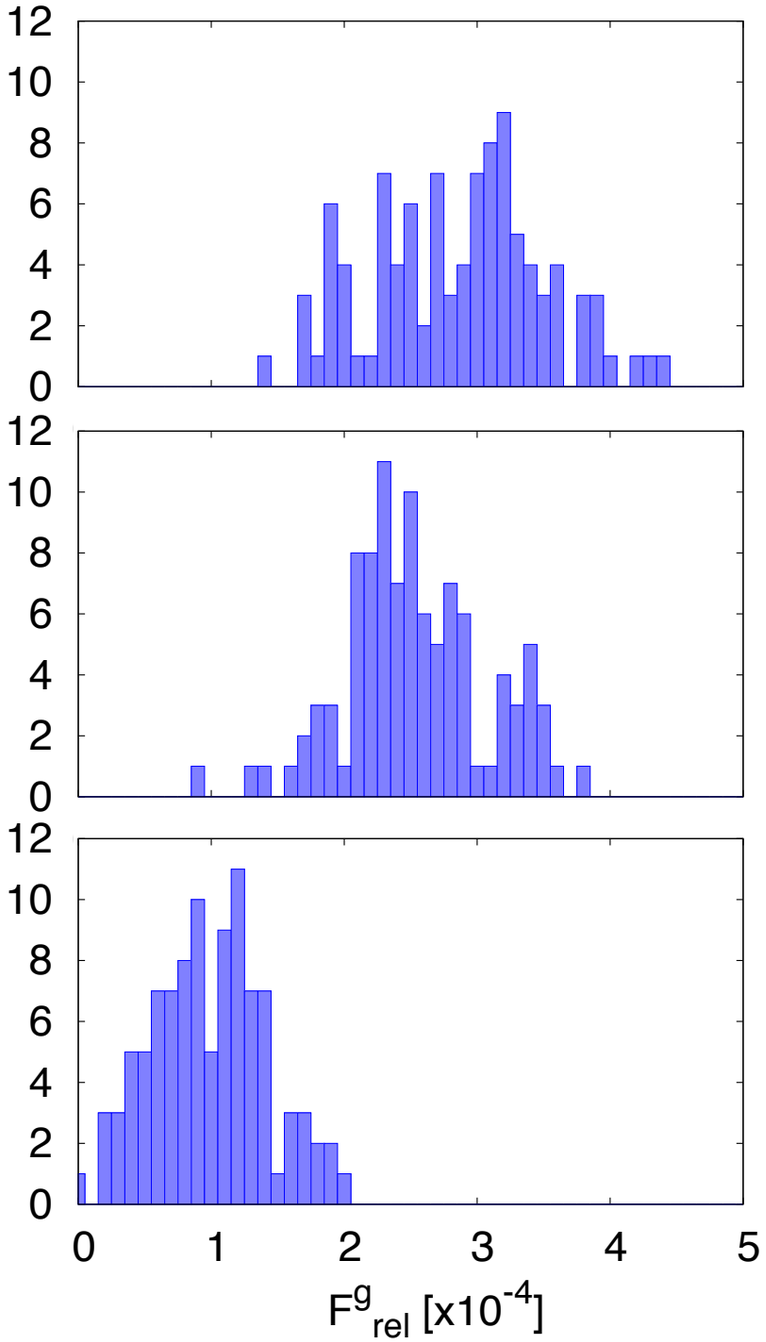
# Simulated annealing: temperature dependence




• 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

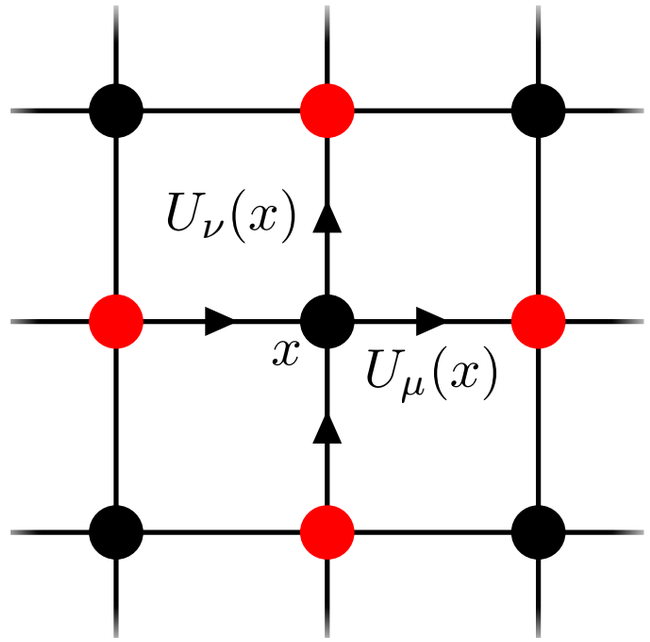
→ 3000 simulated annealing steps

→ 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 $g(x) \in \mathrm{SU}(2)$
        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**

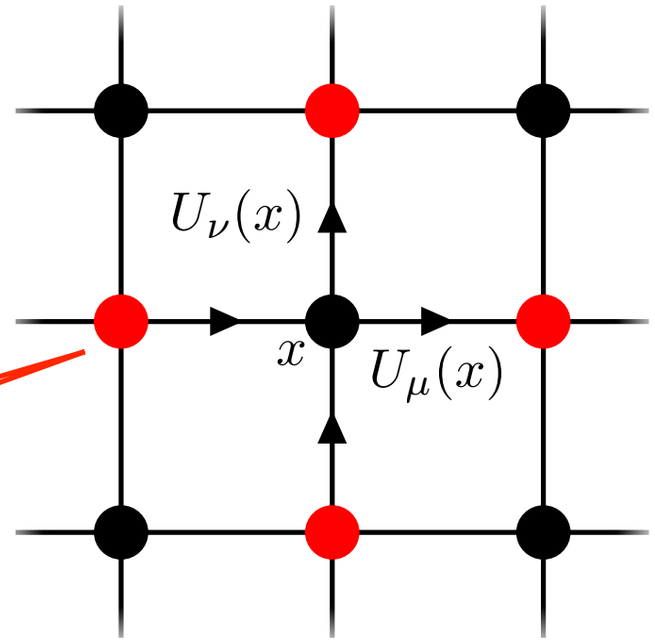


# The algorithm in detail

**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 $g(x) \in \mathrm{SU}(2)$
        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**

local optimization



# Run the algorithm on the device

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

**for all** SU(2) subgroups **do**
  local optimization: find $g(x) \in \mathrm{SU}(2)$
  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**

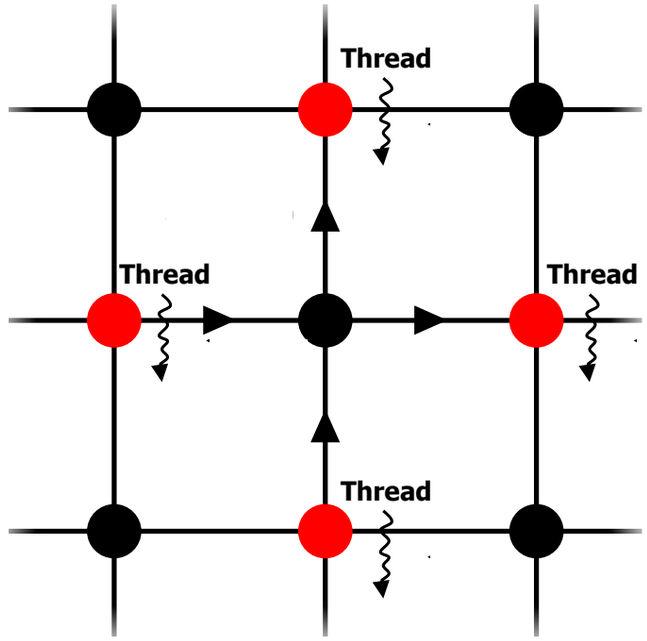
```
}
```

# Run the algorithm on the device

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

**for all** SU(2) subgroups **do**
  local optimization: find $g(x) \in \mathrm{SU}(2)$
  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**

```
}
```

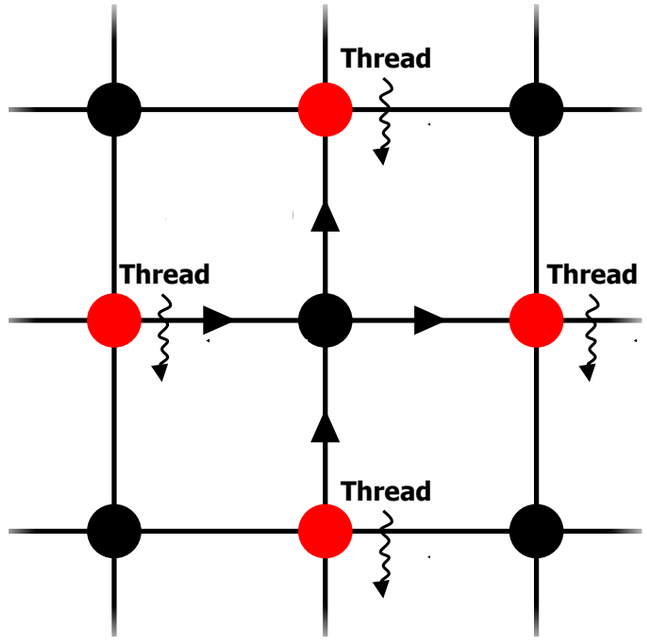


# Run the algorithm on the device

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

**for all** SU(2) subgroups **do**
  local optimization: find $g(x) \in \mathrm{SU}(2)$
  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

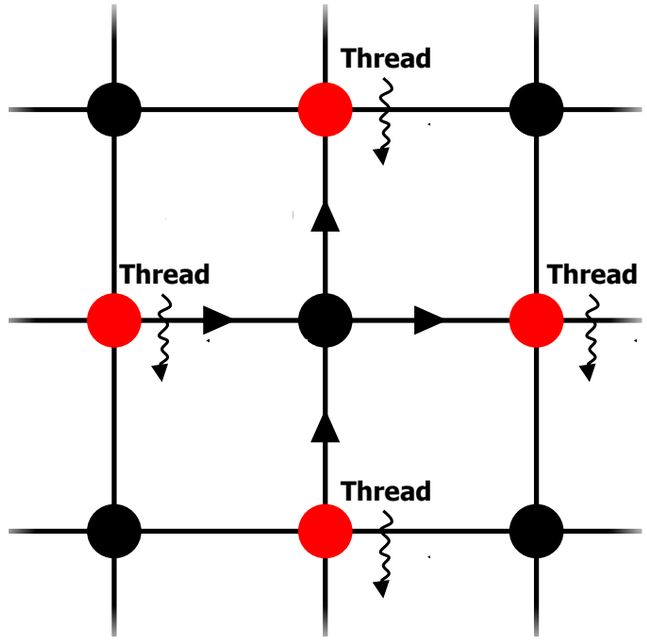
```
relaxKernel<<<V/32,32>>>(U);
```

# Run the algorithm on the device

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

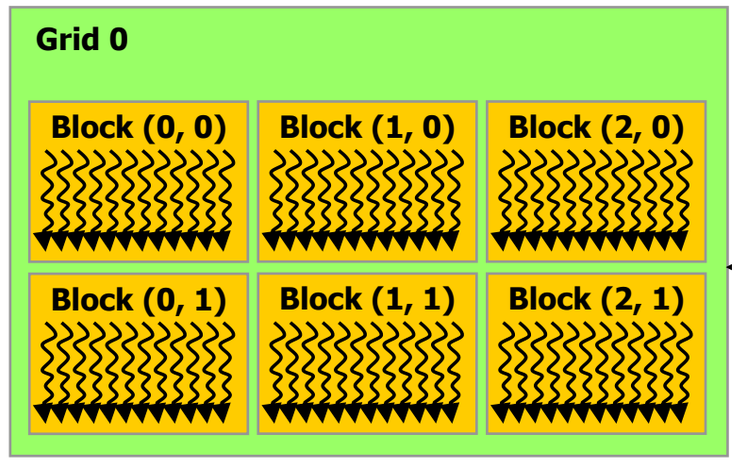
**for all** SU(2) subgroups **do**
  local optimization: find $g(x) \in \mathrm{SU}(2)$
  which is a function of $U_\mu(x)$, $U_\mu(x-\hat{\mu})$
  **for all** $\mu$ **do**
    apply $g(x)$ to $U_\mu(x)$, $U_\mu(x-\hat{\mu})$
  **end for**
**end for**

```
}
```




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

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




© NVIDIA

# 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

# 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 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

• 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 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

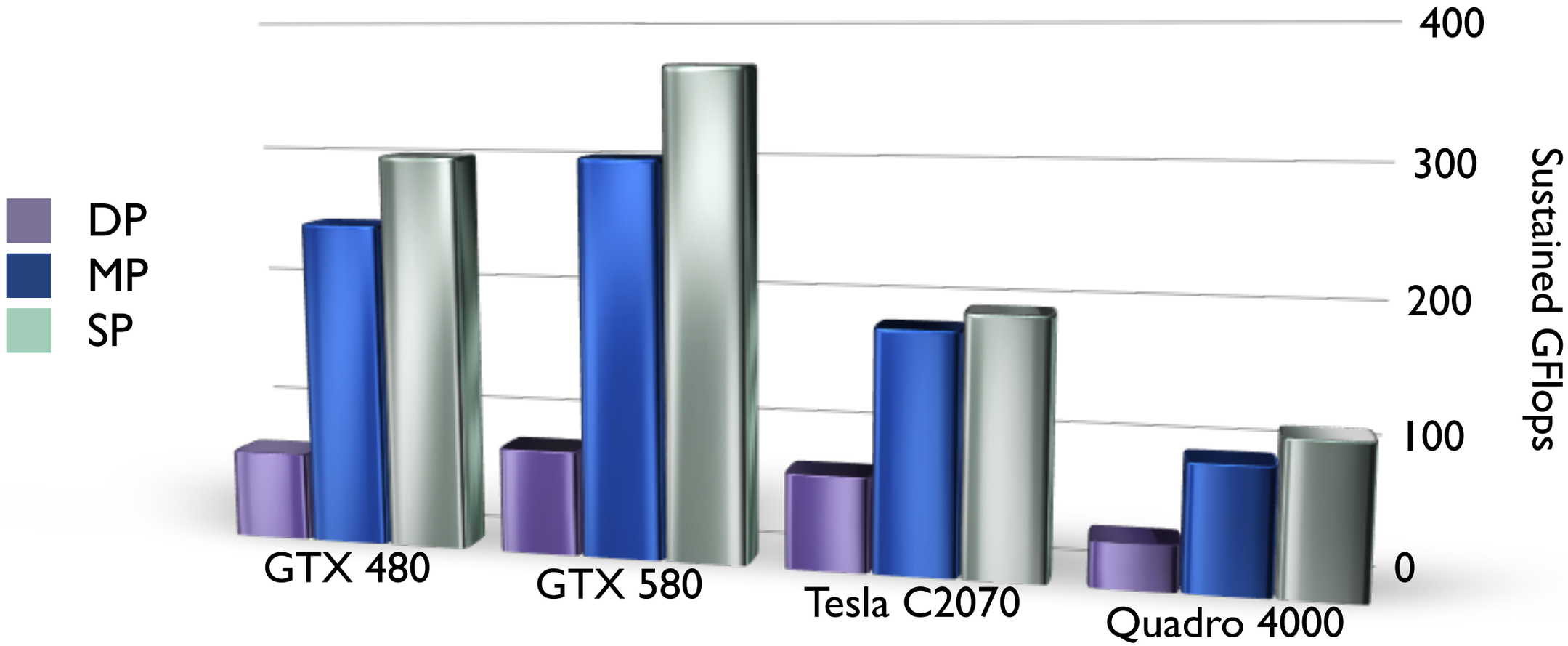
• we avoid this by assigning eight threads to each lattice site: each of the eight threads takes care of one of the eight neighbor links and therefore the number of registers is sufficient

• 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



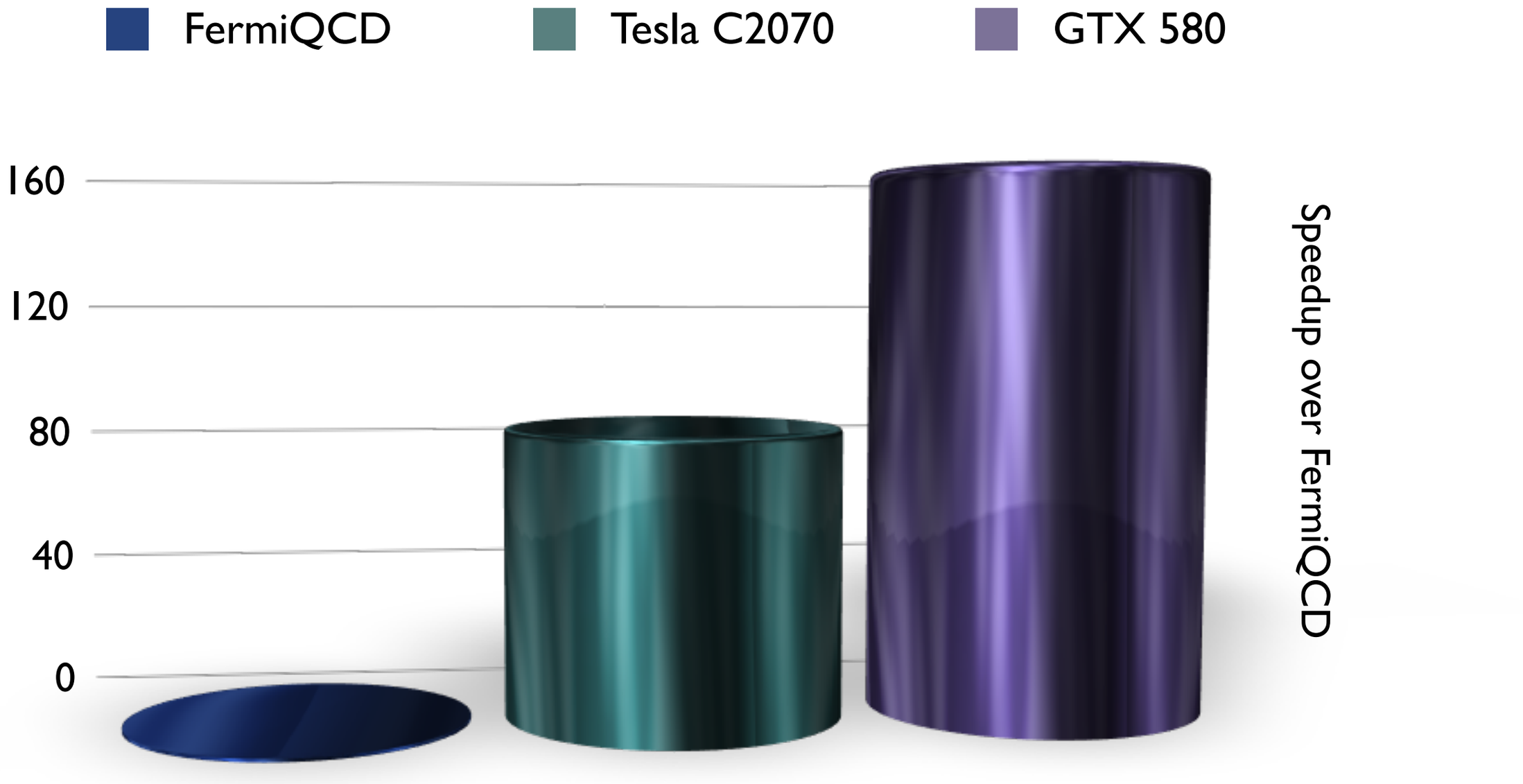
# Comparison to the CPU

• 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

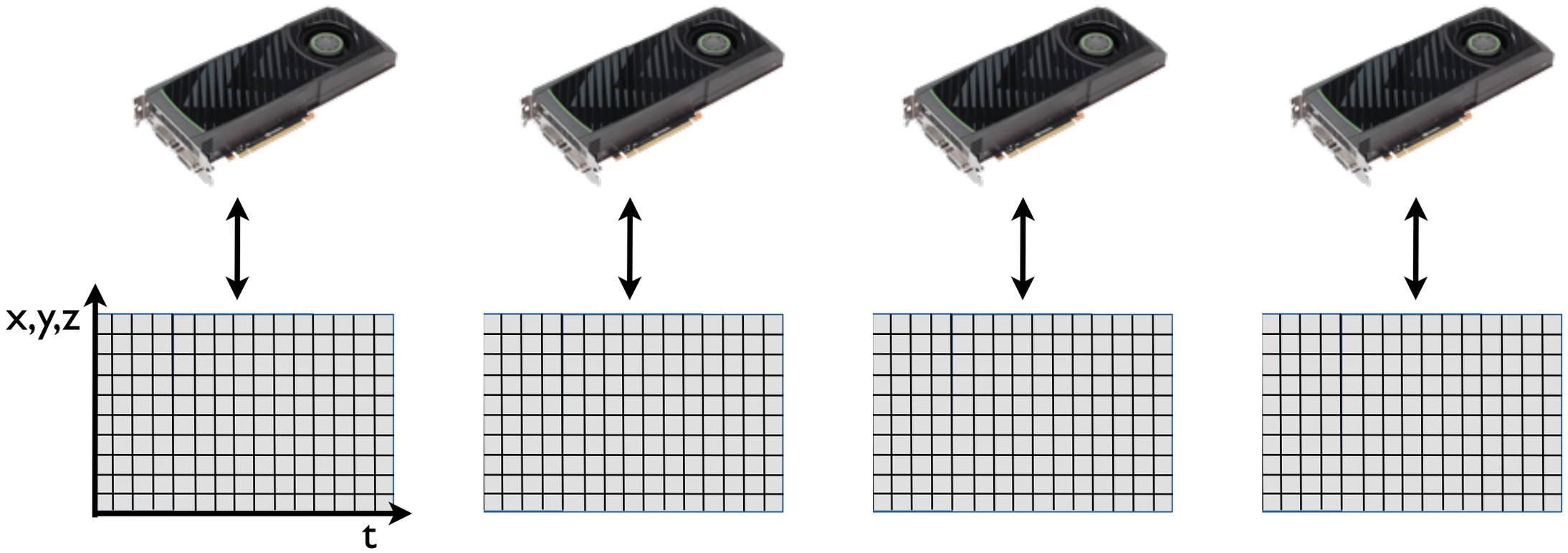
• we compare our performance to FermiQCD run on a Intel Xeon Six-Core CPU X5650 (“Westmere”) @ 2.67GHz with MPI



# Multi-GPUs

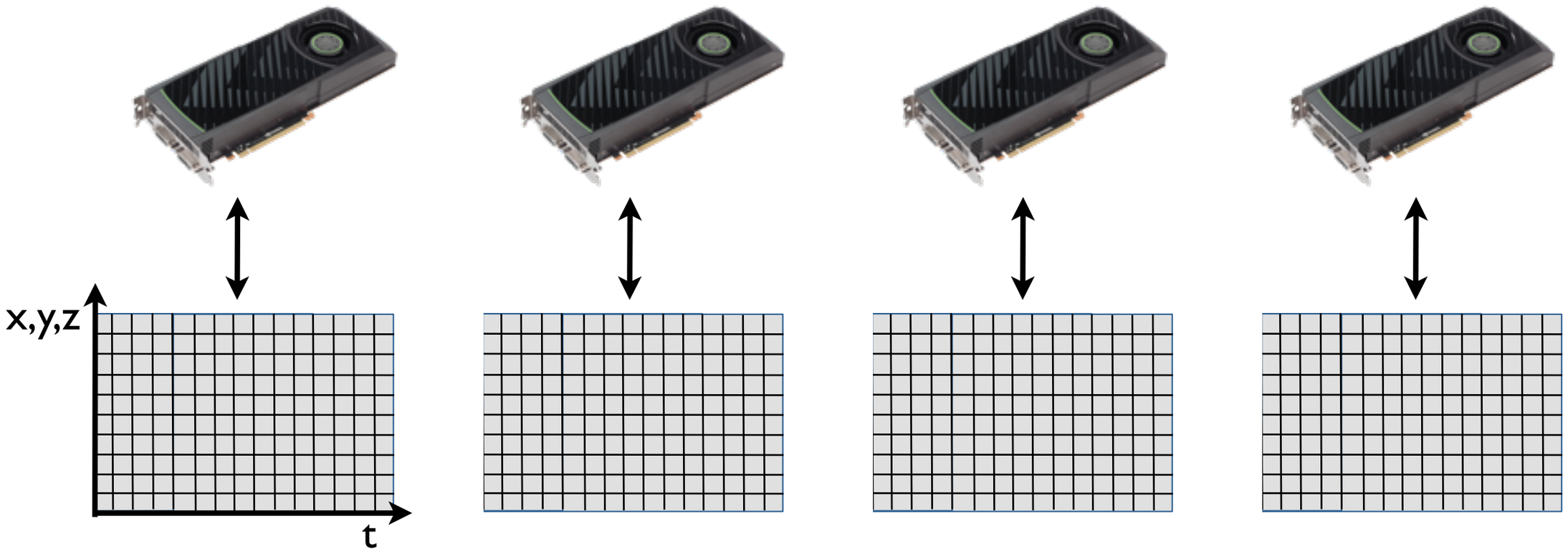
# Multi-GPUs

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

# Data exchange between devices

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

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

- 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):

| $N_s^3$ | D2H $[\mu s]$ | H2D $[\mu s]$ | kernel $[\mu s]$ | D2H/kernel | H2D/kernel |
|---|---|---|---|---|---|
| 16 | 0.0398 | 0.0368 | 0.0209 | 1.90 | 1.76 |
| 32 | 0.2543 | 0.2276 | 0.1443 | 1.76 | 1.58 |
| 64 | 1.2510 | 1.1830 | 1.0489 | 1.19 | 1.13 |
| 128 | 8.9597 | 8.7169 | 8.3041 | 1.08 | 1.05 |

# 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):

| $N_s^3$ | D2H $[\mu s]$ | H2D $[\mu s]$ | kernel $[\mu s]$ | D2H/kernel | H2D/kernel |
|---|---|---|---|---|---|
| 16 | 0.0398 | 0.0368 | 0.0209 | 1.90 | 1.76 |
| 32 | 0.2543 | 0.2276 | 0.1443 | 1.76 | 1.58 |
| 64 | 1.2510 | 1.1830 | 1.0489 | 1.19 | 1.13 |
| 128 | 8.9597 | 8.7169 | 8.3041 | 1.08 | 1.05 |

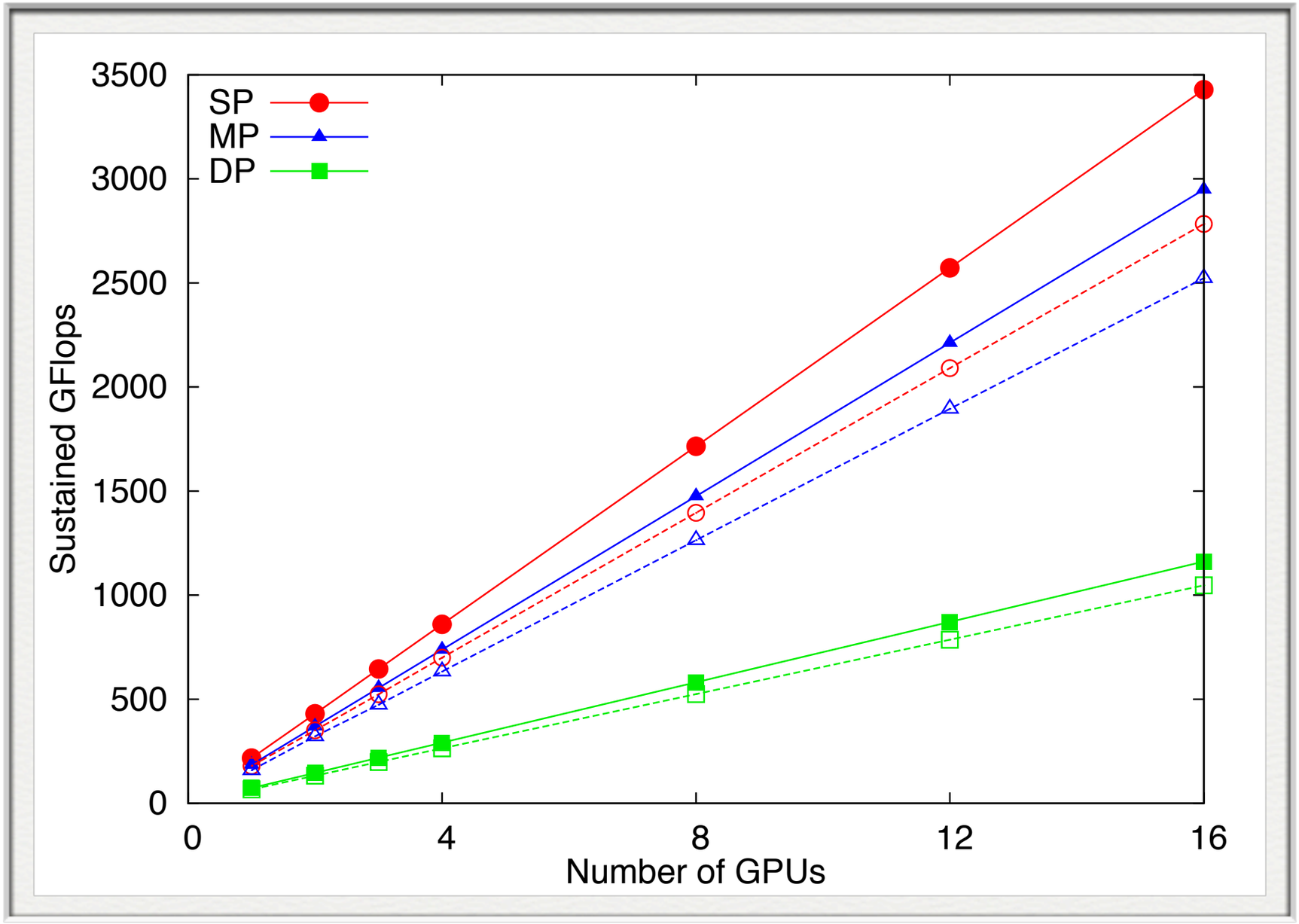
• we conclude that two time-slices per copy step (the six steps from before) are enough to hide the time that is spent for communications

# 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 $48^4$ )

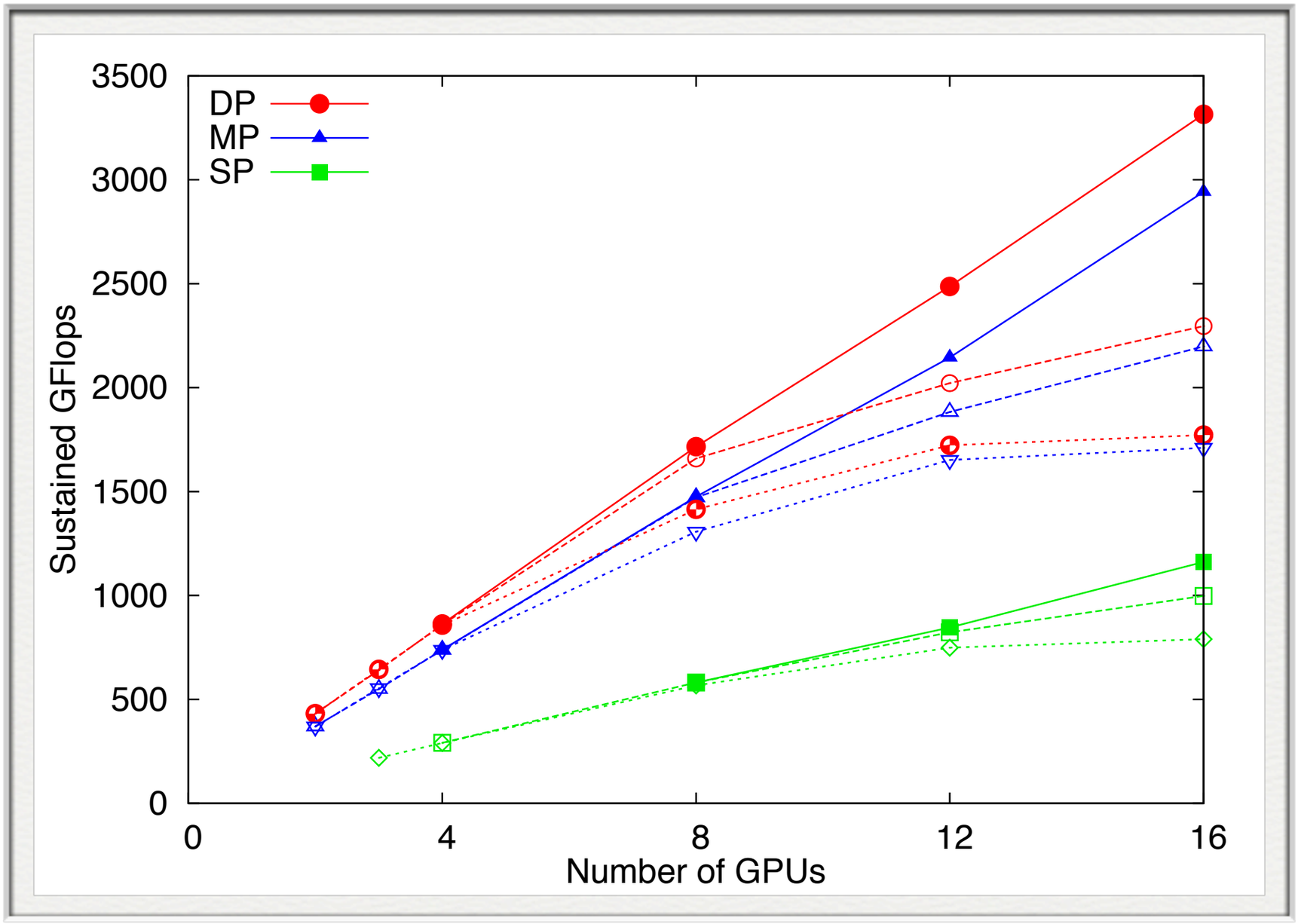


# Strong scaling

- 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 then one billion gauge copies per gauge orbit

- our code is available for download under

www.cuLGT.com

# Thank you!