Optimisation Myths and Facts as Seen in Statistical Physics

Massimo Bernaschi
Institute for Applied Computing
National Research Council
&
Computer Science Department
University "La Sapienza"
Rome - ITALY
m.bernaschi@iac.cnr.it

Presentation Outline

- Update by overrelaxation of Heisenberg spin-glass
- Survey on possible implementations for GPU
- Results on GPU and comparison with a high-end multicore CPU
- Multi-GPU: possible implementations and results
- Conclusions

The (3D) Heisenberg Spin Glass Model

A system whose energy is given by:

$$-\sum J_{[x,y,z],[x\pm1,y\pm1,z\pm1]}\sigma_{[x,y,z]}\sigma_{[x\pm1,y\pm1,z\pm1]}$$

where the sums run on nearest neighbors.

- $\sigma_{[x,y,z]}$ is a 3-component vector $[a,b,c]$
  - $[a,b,c] \in \mathbb{R}$
  - $||\sigma|| = \sqrt{a^2+b^2+c^2} = 1$
- $J_{[x,y,z],[x\pm1,y\pm1,z\pm1]} \in \mathbb{R}$ are scalar random variables with Gaussian distribution:
  - average value equal to 0
  - variance equal to 1
  - for $J > 0$ spins tend to align but for $J < 0$ spins tend to misalign

spins are frustrated... and simulations take long times!

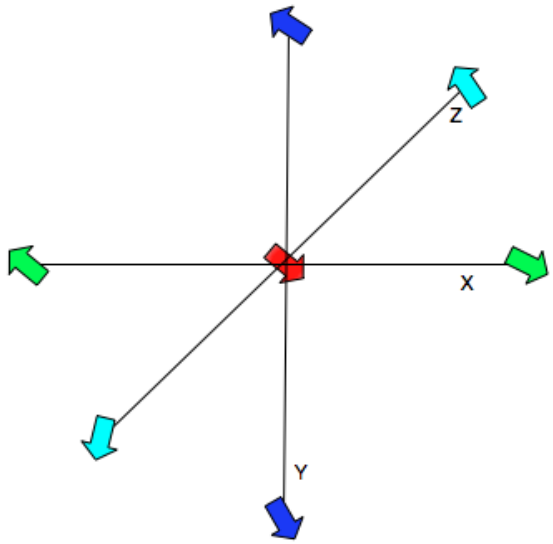
Update by Overrelaxation

The overrelaxation is the maximal move that leaves the system energy invariant:

```
foreach σ⃗ in (SPIN SYSTEM){
    H⃗_σ = 0⃗
     foreach n in (NEIGHBORS OF σ⃗){
         H⃗_σ = H⃗_σ + J_n σ⃗_n
     }
    σ⃗ = 2(H⃗_σ · σ⃗ / H⃗_σ · H⃗_σ) H⃗_σ − σ⃗
}
```

a set of scalar products that involve only simple floating point arithmetic operations.

In 3D, for a single spin:

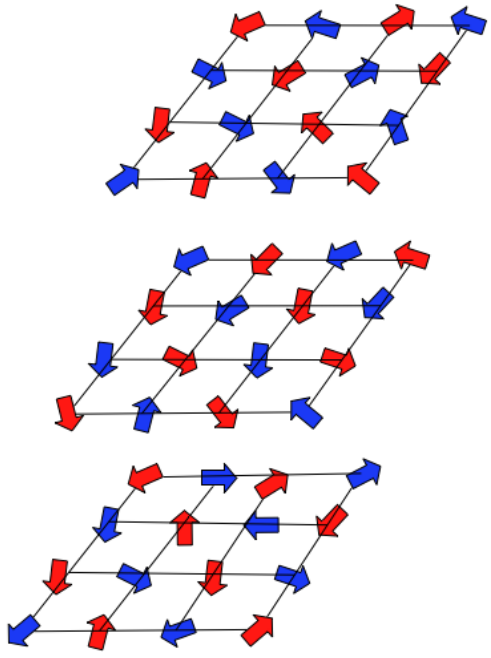


- 20 sums
- 3 differences
- 28 products
- one division ($\sim$ 10 Flops).

Check-board decomposition

Spins can be updated in parallel if they don't interact directly.

- Simple *(check-board)* decompositions of the domain can be used to guarantee the consistency of the update procedure.

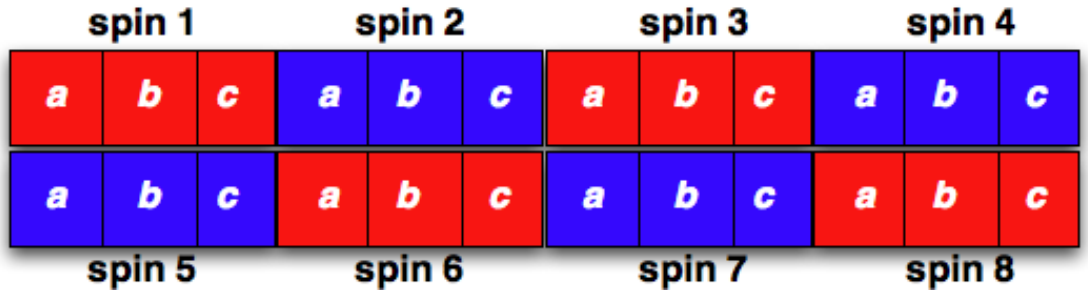


Data structures

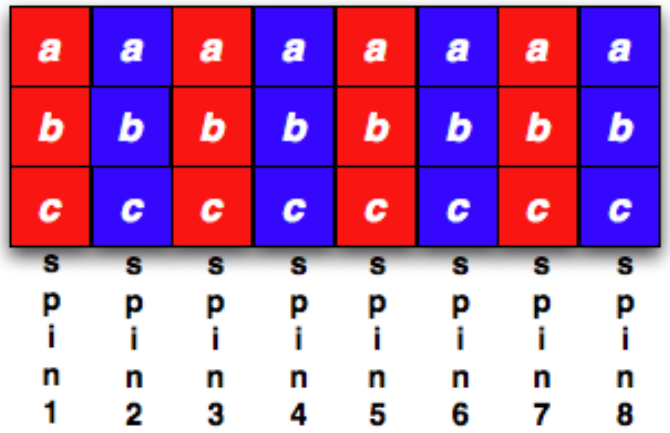
Spins: each spin is a 3-component ($[a, b, c]$) vector.

At least two alternatives (assuming C-like memory ordering).

- array of structures
  (good for data locality)



- structure of arrays
  (good for thread locality)



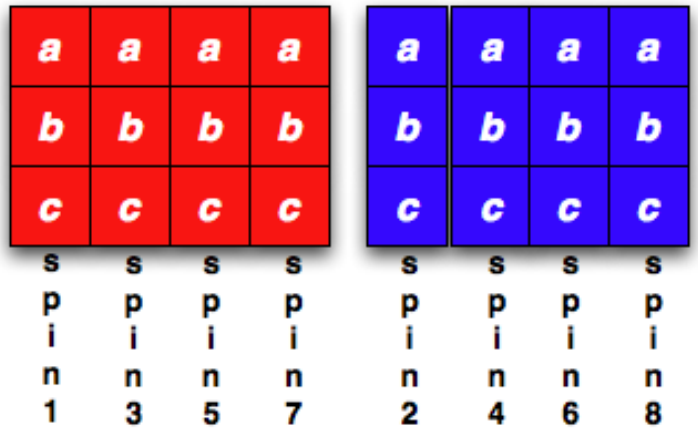
Couplings: each coupling is a scalar but there is a different coupling for each spin in each direction (x, y, z).
So we have the same possible alternatives:

Structure of Arrays (SoA) on GPU and Array of Structures (AoS) on CPU.

A survey of possible techniques: GPU global memory + registers (1)

A straightforward approach: each thread loads data directly from GPU global memory to GPU registers (no shared memory variables)

- Use the *structure of arrays* data layout with red and blue spins stored separately
- Two kernel invocations:
  one for red spins;
  one for blue spins.



- Very limited resource usage: 50 registers. No shared memory.
- Up to 640 threads per SM on Fermi GPUs.

A survey of possible techniques: GPU global memory + registers (2)

In spite of its simplicity this scheme produces a lot of data traffic:

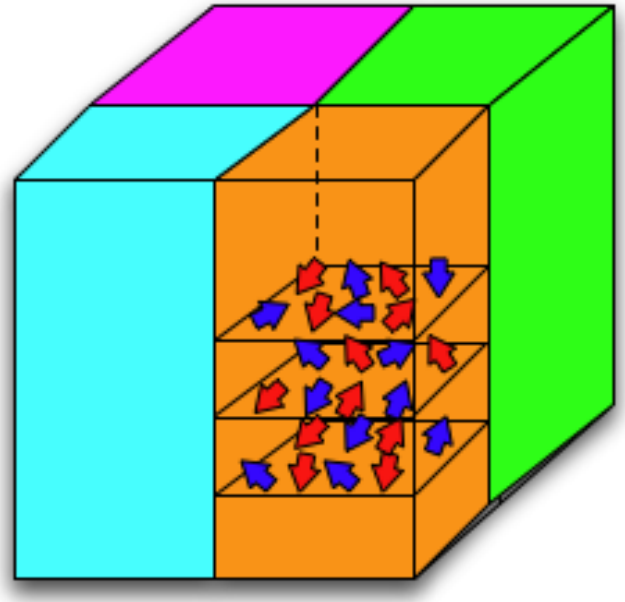
- each spin has 6 neighbors;
- each spin loaded exactly **7** times from the Global Memory (assuming periodic boundary conditions).
- one time, by one thread, for its update;
- six times (by six other threads) to contribute to the update of its neighbors.

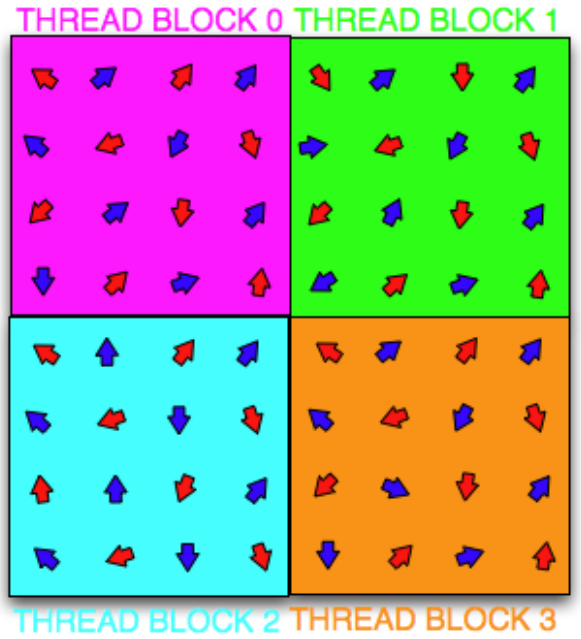
It looks like a situation where GPU *shared memory* could come to rescue...

A survey of possible techniques: GPU shared memory (3-plane version) (1)

An approach originally proposed to solve the Laplace equation

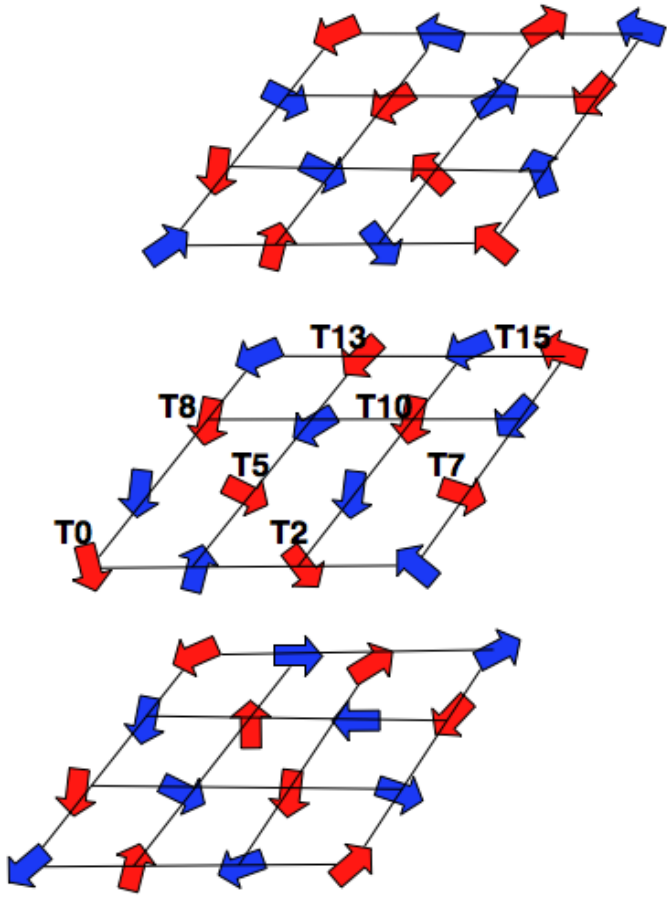
- The domain is divided in columns.
- A single block of threads is in charge of a column





A survey of possible techniques: GPU shared memory (3-plane version) (2)

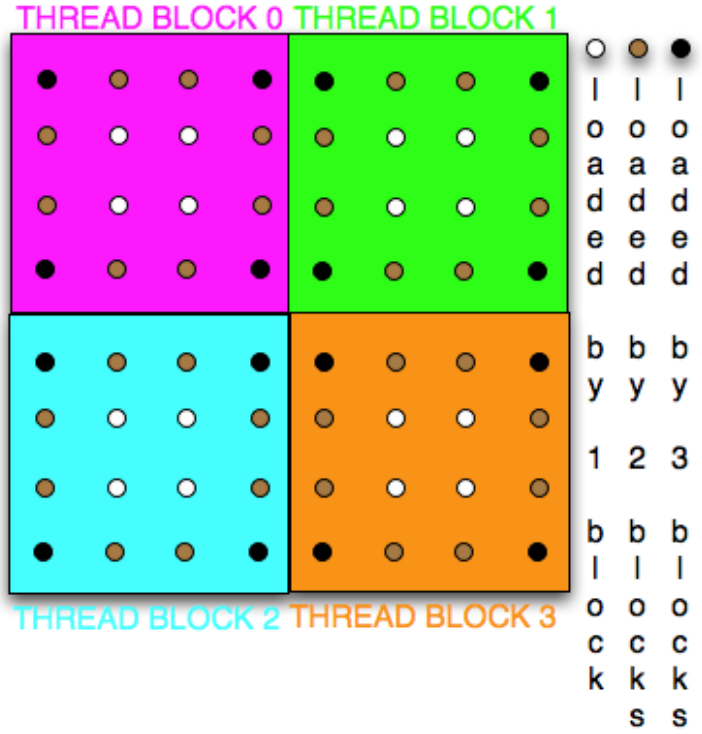
Once data are in shared memory, all red spins of the active plane can be updated concurrently then...



A survey of possible techniques: GPU shared memory (3-plane version) (3)

Data traffic between Global Memory and GPU processors is dramatically reduced!

- Only the spins on the boundaries of the subdomains need to be loaded more than once.
- A spin needs to be loaded three times, at most, instead of seven!



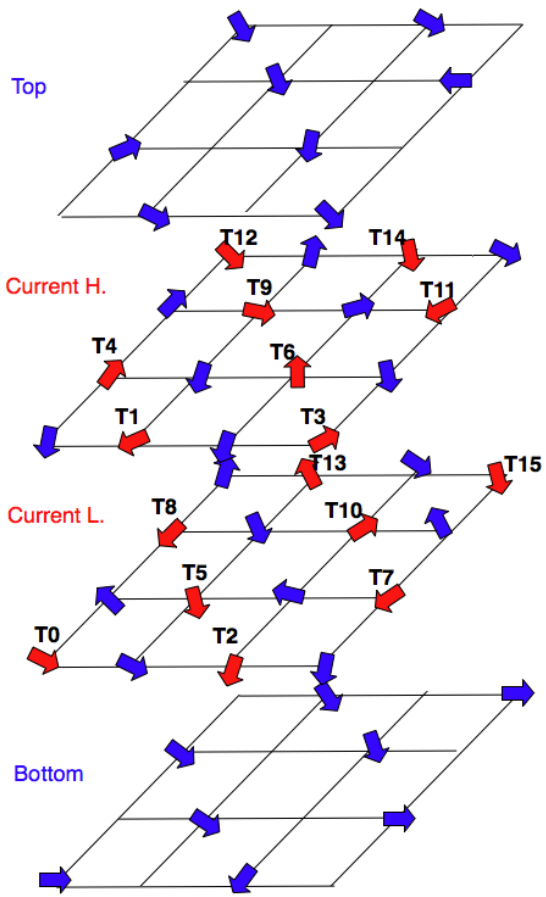
A survey of possible techniques: GPU shared memory (3-plane version) (4)

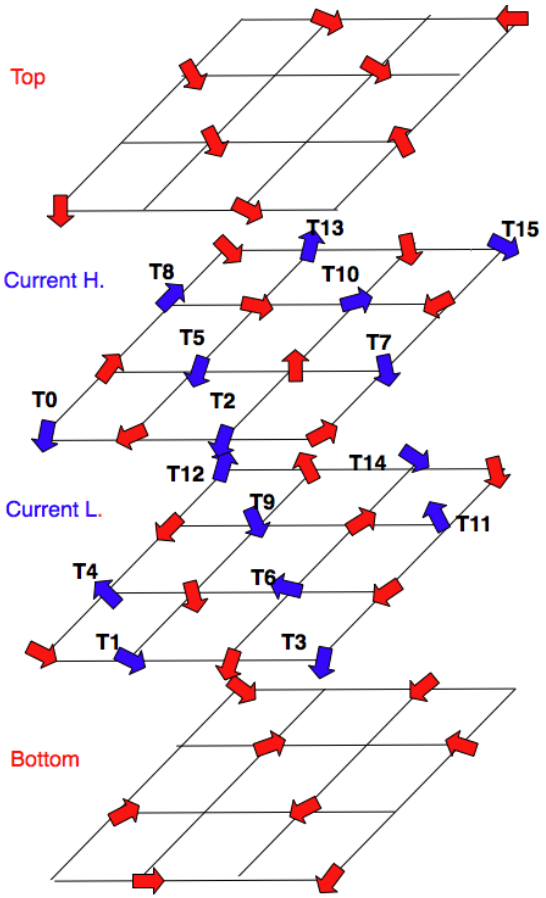
Resources usage:

- Shared memory (single precision):
  - 6 variables *per* spin: 24 bytes
    (3 spin components and the couplings along the 3 directions)
  - each thread loads 3 spins $->$ 72 bytes of shared memory.
- Registers: 62 (no local memory).
- On Fermi GPUs, having up to 48 Kbytes of shared memory, the number of threads is limited by the number of registers (32768).

Even more important... half of the threads are idle after data loading!
What is a possible solution?

A survey of possible techniques: GPU shared memory (4-plane version)





Working on two planes allows to keep all threads busy!

In the end **two** planes have been updated.

- 4 planes need to be resident in shared memory.
- More shared memory *per* thread is required.
- Fewer threads *per* SM: max **384** on Fermi GPUs
  (the global memory version could have up to **640** threads).

Further variants are possible by replacing plain (global) memory accesses with *texture fetches* operations:

- exploit *texture* caching capabilities;
- make easier to manage (periodic) boundary conditions.

Results

Source codes and test case available from

`http://www.iac.rm.cnr.it/~massimo/hsgfiles.tgz`

The test case is a system having 128^3 spins.

- spins and couplings are single precision variables;
- energy is accumulated in a double precision variable.

The performance measure is the time in nanoseconds required for a single spin update (lower is better)

$$T_{upd} = \frac{TotalTime \text{ (excluding I/O)}}{(\#\ of\ iterations)\times(\#\ of\ spins)}$$

GM: simple Global Memory version;

ShM: Shared Memory based (3 planes);

ShM4P: Shared Memory based (4 planes);

TEXT: simple Global Memory using *texture* operations for data fetching.

Platform	*# threads*	T_{upd}
Intel X5680 (SSE instr.)	1	$\sim$ 13.5 ns
Intel X5680 (SSE instr. + OpenMP)	8	$\sim$ 3.4 ns
Tesla C1060 GM	320	1.9 ns
Tesla C1060 ShM	128	2.5 ns
Tesla C1060 ShM4P	64	2.2 ns
Tesla C1060 TEXT	320	1.8 ns
GTX 480 (Fermi) GM	320	0.66 ns
GTX 480 (Fermi) ShM	256	1.3 ns
GTX 480 (Fermi) ShM4P	192	0.86 ns
GTX 480 (Fermi) TEXT	480	0.63 ns

Single vs. Double precision

Platform	*# threads*	$T_{upd}(s.p.)$	$T_{upd}(d.p.)$
GTX 480 (Fermi) GM	320	0.66 ns	1.3 ns
GTX 480 (Fermi) ShM4P	192/96	0.86 ns	2.2 ns
Intel X5680 3.33 Ghz (SSE instr.)	1	13.5 ns	26.6 ns
Intel X5680 3.33 Ghz (SSE instr.) (SSE instr. + OpenMP)	8	3.5 ns	6.8 ns

GM: simple Global Memory version;

ShM4P: Shared Memory based (4 planes).

The Error Correcting Code effect

Platform	*ECC on*	*ECC off*
C2050 (Fermi) GM	1.0 ns	0.84 ns
C2050 (Fermi) ShM	1.63 ns	1.66 ns
C2050 (Fermi) ShM4P	1.4 ns	1.3 ns
C2050 (Fermi) TEXT	1.0 ns	0.78 ns

The C2050 is slightly slower than the GTX 480 (14 instead of 15 SM and lower clock rate).

Intel X5680 "Westmere" (3.33 Ghz). Intel compiler: v. 11.1

# of *cores*	T_{upd} (s.p.) no SSE	T_{upd} (s.p.)	T_{upd} (d.p.)
1	32.6 ns	13.6 ns	26.6 ns
2	17.8 ns	7.1 ns (96%)	15.6 ns
4	8.5 ns	4.7 ns (72%)	9.0 ns
6	6.3 ns	3.7 ns (61%)	7.4 ns
8	4.6	3.3 ns (51%)	6.8 ns
10	4.2	3.3 ns (41%)	7.0 ns
12	3.9	3.4 ns (33%)	7.0 ns

System size 128^3. Very minor differences for 256^3.

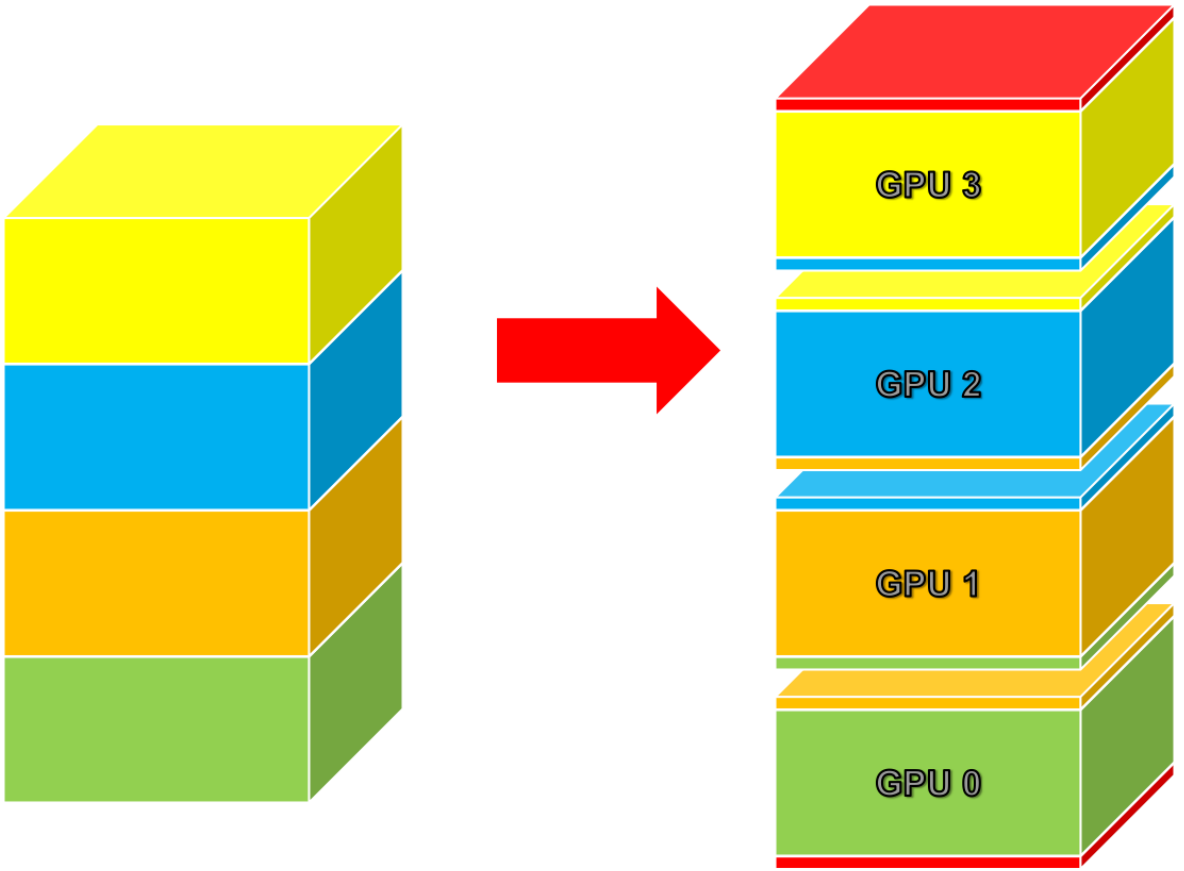
Parallelization based on OpenMP. To "convince" the compiler that the innermost loop can be safely vectorized at source code level...

```
REAL * __restrict__ anspx=an+spxoff,
     * __restrict__ ansmx=an+smxoff;
REAL * __restrict__ anspy=an+spyoff,
     * __restrict__ ansmy=an+smyoff;
REAL * __restrict__ anspz=an+spzoff,
     * __restrict__ ansmz=an+smzoff;
REAL * __restrict__ bnspx=bn+spxoff,
     * __restrict__ bnsmx=bn+smxoff;
REAL * __restrict__ bnspy=bn+spyoff,
     * __restrict__ bnsmy=bn+smyoff;
REAL * __restrict__ bnspz=bn+spzoff,
     * __restrict__ bnsmz=bn+smzoff;
REAL * __restrict__ cnspx=cn+spxoff,
     * __restrict__ cnsmx=cn+smxoff;
REAL * __restrict__ cnspy=cn+spyoff,
     * __restrict__ cnsmy=cn+smyoff;
REAL * __restrict__ cnspz=cn+spzoff,
     * __restrict__ cnsmz=cn+smzoff;
REAL * __restrict__ JpxWO=Jpx+yzoff,
     * __restrict__ JmxWO=Jmx+yzoff;
REAL * __restrict__ JpyWO=Jpy+yzoff,
     * __restrict__ JmyWO=Jmy+yzoff;
REAL * __restrict__ JpzWO=Jpz+yzoff,
     * __restrict__ JmzWO=Jmz+yzoff;
REAL * __restrict__ at=a+yzoff,
     * __restrict__ bt=b+yzoff,
     * __restrict__ ct=c+yzoff;
REAL * __restrict__ ta=t_a+yzoff,
     * __restrict__ tb=t_b+yzoff,
     * __restrict__ tc=t_c+yzoff;
```

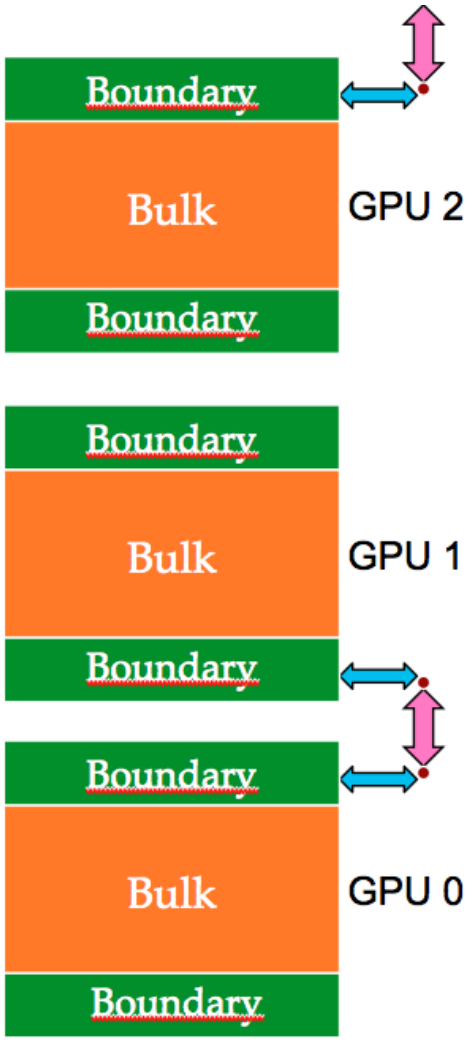
```
REAL as, bs, cs, factor;
int ix;
for(ix=istart; ix<iend; ix++) {
    as=anspx[ix]*JpxWO[ix]+
    anspy[ix]*JpyWO[ix]+
    anspz[ix]*JpzWO[ix]+
    ansmx[ix]*JmxWO[ix]+
    ansmy[ix]*JmyWO[ix]+
    ansmz[ix]*JmzWO[ix];

    bs=bnspx[ix]*JpxWO[ix]+
    bnspy[ix]*JpyWO[ix]+
    bnspz[ix]*JpzWO[ix]+
    bnsmx[ix]*JmxWO[ix]+
    bnsmy[ix]*JmyWO[ix]+
    bnsmz[ix]*JmzWO[ix];

    cs=cnspx[ix]*JpxWO[ix]+
    cnspy[ix]*JpyWO[ix]+
    cnspz[ix]*JpzWO[ix]+
    cnsmx[ix]*JmxWO[ix]+
    cnsmy[ix]*JmyWO[ix]+
    cnsmz[ix]*JmzWO[ix]+h;

    factor=two*(as*at[ix]+bs*bt[ix]+cs*ct[ix])/
                (as*as+bs*bs+cs*cs);
    ta[ix]=as*factor-at[ix];
    tb[ix]=bs*factor-bt[ix];
    tc[ix]=cs*factor-ct[ix];
}
```

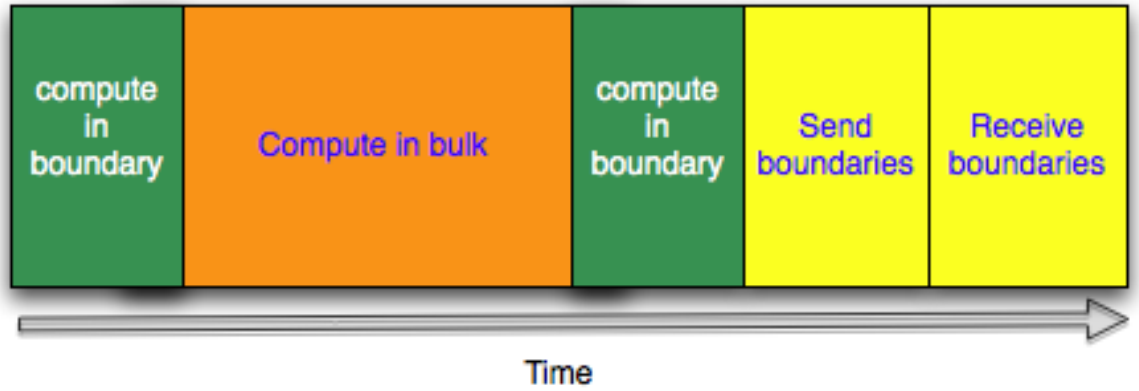
Multi-GPU

Simple domain decomposition along one axis



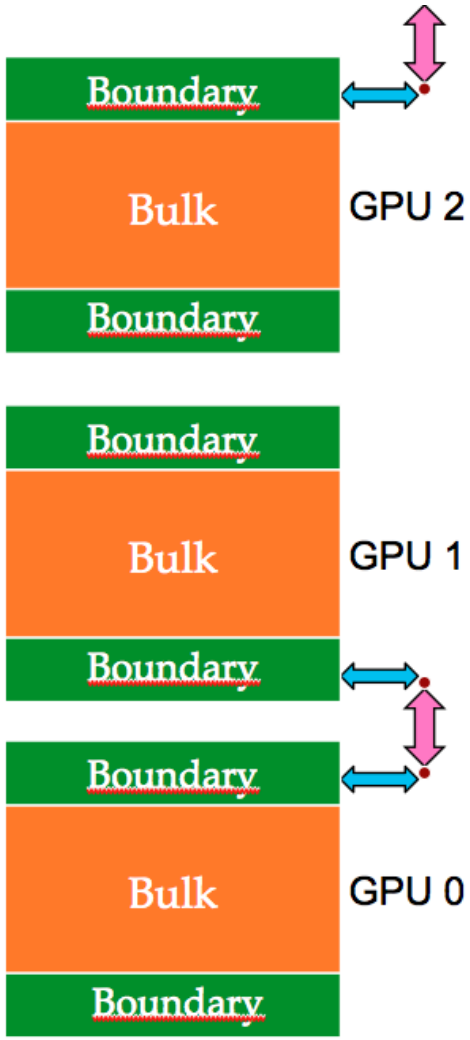
Multi-GPU: a first simple approach



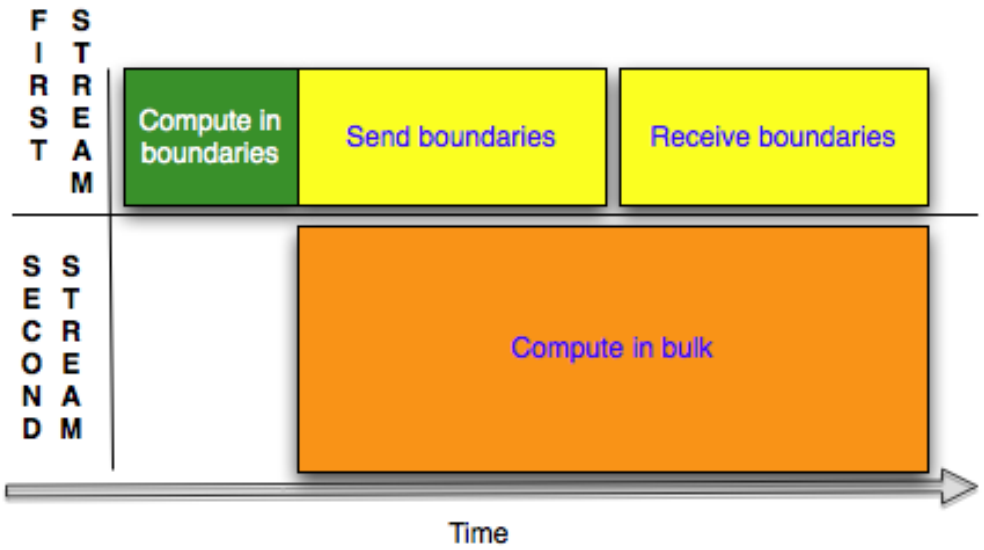


1. compute in boundaries;
2. compute in bulk;
3. exchange data (by using CPUs!);
4. goto to 1.

Multi-GPU: Overlap of communication and computation



- Compute the overrelaxation for the *bulk* and, at the same time, exchange data for the *boundaries*. Requires:
  - Cuda Streams
  - Asynchronous Memory Copy operations



1. Create 2 streams: one for the boundaries, one for the bulk;
2. concurrently:

stream one starts to update the boundaries;

stream two starts to update the bulk
(actually the latter starts only when the SM is available);

3. concurrently:

stream one copies data in the boundaries from the GPU to the CPU;
- exchanges data among CPUs by using MPI;
- copies data to the boundaries from the CPU to the GPU;

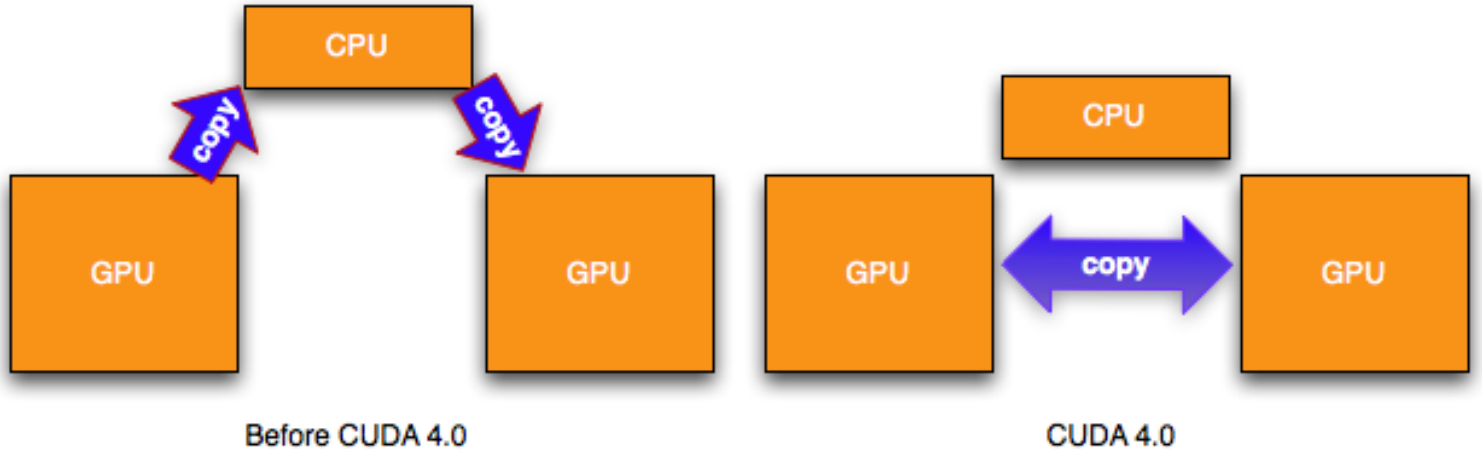
stream two continues to update the bulk;

4. go to 2.

The CPU acts as a MPI-coprocessor of the GPU!

Multi-GPU: Peer-to-Peer Memory Copy

- Starting on CUDA 4.0, memory copies can be performed between two different GPU connected to the same PCI-e root complex.
- If *peer-to-peer* access is enabled then the copy operation no longer needs to be staged through the CPU and is therefore faster!



- By using streams it remains possible to overlap communication and computation.

MPI based multi-GPU results

- CPU: Dual Intel Xeon(R)X5550 @ 2.67GHz. GPU: S2050.
- QDR Infiniband connection among nodes.
- MPI intra node communication *via* shared memory.

# of *GPU*	T_{upd} naive	T_{upd} with overlap	Efficiency with overlap
1	0.66 ns	0.66 ns	N.A.
2	0.45 ns	0.36 ns	91%
4	0.3 ns	0.19 ns	86%
8	0.22 ns	0.16 ns	51%

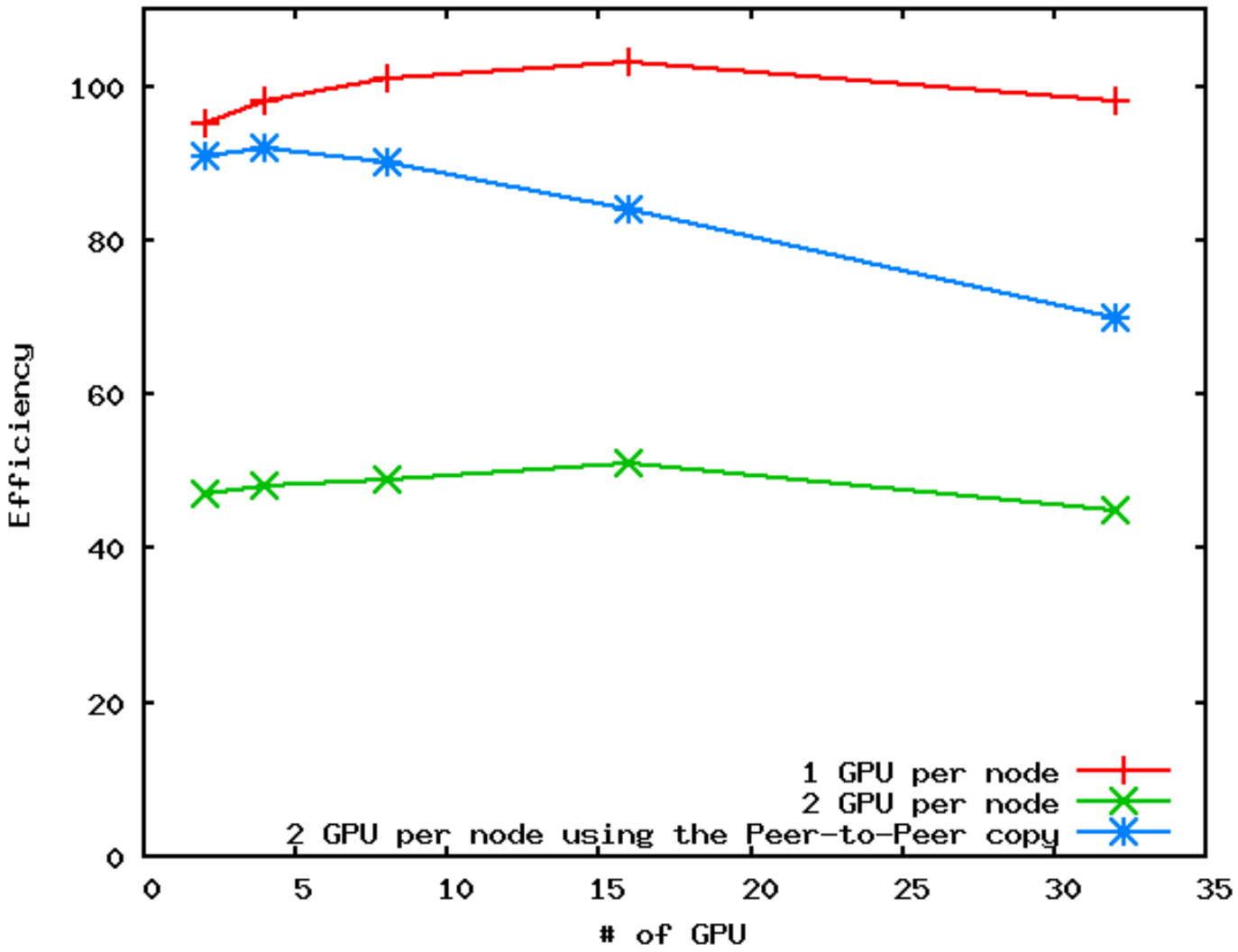
System size 256^3. Single Precision.

MPI, P2P and Hybrid multi-GPU results

- 8 2050 GPU with the same Hw configuration;
- Reference time from the 256^3 test case
  - 512^3 does not fit in a single GPU.

MPI	P2P	Speedup	Efficiency
8	1	5.02	63%
4	2	5.91	74%
2	4	6.80	85%
1	8	7.70	96%

System size 512^3. Single Precision.



Efficiency with up to 32 (2070) GPU for a 512^3 system.

Conclusions

Disclaimer: What follows applies to the Heisenberg spin glass model and very likely to other spins systems.
Any inference for other computational problems is at your own risk!

- GPU implementations outperform any CPU implementation.
- Vector instructions are absolutely required to get higher performances on multi-core CPUs.
- High end multi-core CPUs may show poor scalability.
- Multi-GPU programming is no more complex than hybrid OpenMP/vector programming but...
  - both Peer-to-Peer copy and MPI explicit message passing must be overlapped with GPU kernels execution.

Thanks!