Multi-GPU simulation of spin (glass) systems
The (3D) Heisenberg Spin Glass Model
The (3D) Heisenberg Spin Glass Model

A physical 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 (3D) Heisenberg Spin Glass Model

A physical 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

- \( \sigma_{[x,y,z]} \) is a 3-component vector \([a,b,c]\)
  - \([a,b,c] \in \mathbb{R}\)
  - \(||\sigma|| = \sqrt{a^2 + b^2 + c^2} = 1||\)
The (3D) Heisenberg Spin Glass Model

A physical system whose energy is given by:

\[ -\sum J[x,y,z], [x\pm 1, y\pm 1, z\pm 1]\sigma[x,y,z]\sigma[x\pm 1, y\pm 1, z\pm 1] \]

where

- \( \sigma[x,y,z] \) is a 3-component vector \([a, b, c]\)
  - \([a, b, c] \in \mathbb{R}\)
  - \(||\sigma|| = \sqrt{a^2 + b^2 + c^2} = 1\)

- \( J[x,y,z], [x\pm 1, y\pm 1, z\pm 1] \in \mathbb{R} \) 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
The (3D) Heisenberg Spin Glass Model

A physical system whose energy is given by:

$$- \sum \sigma[x,y,z][x \pm 1, y \pm 1, z \pm 1] \sigma[x, y, z] \sigma[x \pm 1, y \pm 1, z \pm 1]$$

where

- $\sigma[x,y,z]$ is a 3-component vector $[a, b, c]$
  - $[a, b, c] \in \mathbb{R}$
  - $||\sigma|| = \sqrt{a^2 + b^2 + c^2} = 1$

- $J[x,y,z],[x \pm 1, y \pm 1, z \pm 1] \in \mathbb{R}$ 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 become frustrated...
Our measure of performances will be the time required for the update of a single spin.

For *benchmarking* purposes we implemented the *Overrelaxation* technique, a general method mainly used as *accelerator* of other algorithms.

For each spin, the overrelaxation is the maximal move that leaves the energy invariant $\sigma_{new} = 2(\overrightarrow{H_\sigma} \cdot \overrightarrow{\sigma}_{old}/\overrightarrow{H_\sigma} \cdot \overrightarrow{H_\sigma})\overrightarrow{H_\sigma} - \overrightarrow{\sigma}_{old}$

$$\overrightarrow{H_\sigma} = J_{[x+1,y,z]} \overrightarrow{\sigma}_{[x+1,y,z]} + J_{[x-1,y,z]} \overrightarrow{\sigma}_{[x-1,y,z]} +$$
$$J_{[x,y+1,z]} \overrightarrow{\sigma}_{[x,y+1,z]} + J_{[x,y-1,z]} \overrightarrow{\sigma}_{[x,y-1,z]} +$$
$$J_{[x,y,z+1]} \overrightarrow{\sigma}_{[x,y,z+1]} + J_{[x,y,z-1]} \overrightarrow{\sigma}_{[x,y,z-1]}.$$
Update by Overrelaxation
Update by Overrelaxation

\[
\text{foreach } \sigma \text{ in (SPIN SYSTEM)} \{
\]
Update by Overrelaxation

foreach \( \vec{\sigma} \) in (SPIN SYSTEM)\
{\
\[ \overrightarrow{H}_{\sigma} = \overrightarrow{0} \]

foreach \( n \) in (NEIGHBORS OF \( \vec{\sigma} \))\
{\
\[ \overrightarrow{H}_{\sigma} = \overrightarrow{H}_{\sigma} + J_{ni} \vec{\sigma}_n \]
}

foreach $\vec{\sigma}$ in (SPIN SYSTEM) {
    $\overrightarrow{H}_\sigma = \overrightarrow{0}$

    foreach $n$ in (NEIGHBORS OF $\vec{\sigma}$) {
        $\overrightarrow{H}_\sigma = \overrightarrow{H}_\sigma + J_n \vec{\sigma}_n$
    }

    $\vec{\sigma} = 2(\overrightarrow{H}_\sigma \cdot \vec{\sigma} / \overrightarrow{H}_\sigma \cdot \overrightarrow{H}_\sigma) \overrightarrow{H}_\sigma - \vec{\sigma}$
}

Update by Overrelaxation
Update by Overrelaxation

\[
\text{foreach } \vec{\sigma} \text{ in (SPIN SYSTEM)} \{
\quad \vec{H}_\sigma = \vec{0}
\]

\[
\text{foreach } n \text{ in (NEIGHBORS OF } \vec{\sigma}) \{
\quad \vec{H}_\sigma = \vec{H}_\sigma + J_n \vec{\sigma}_n
\}
\]

\[
\vec{\sigma} = 2(\vec{H}_\sigma \cdot \vec{\sigma} / \vec{H}_\sigma \cdot \vec{H}_\sigma)\vec{H}_\sigma - \vec{\sigma}
\]

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

In 3D, for a single spin, there are \( \sim 60 \) floating point operations:
foreach $\sigma$ in (SPIN SYSTEM) {
  $\vec{H}_\sigma = \vec{0}$
  foreach n in (NEIGHBORS OF $\vec{\sigma}$) {
    $\vec{H}_\sigma = \vec{H}_\sigma + J_n \vec{\sigma}_n$
  }
  $\vec{\sigma} = 2(\vec{H}_\sigma \cdot \vec{\sigma} / \vec{H}_\sigma \cdot \vec{H}_\sigma) \vec{H}_\sigma - \vec{\sigma}$
}

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

In 3D, for a single spin, there are $\sim 60$ floating point operations:

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

A simple *(check-board)* decomposition of the domain guarantees 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).
**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)
Data structures

Spins: each spin is a 3-component \([a, b, c]\) vector.
At least two alternatives (assuming \(C\)-like memory ordering).

- structure of arrays
  (good for thread locality)
Spins: each spin is a 3-component \([a, b, c]\) vector.
At least two alternatives (assuming \(C\)-like memory ordering).

- structure of arrays
  (good for thread locality)

- On GPU we choose the SoA data structure
A possible implementation technique: GPU global memory + registers (1)

A straightforward approach: each thread loads data directly from GPU global memory to GPU registers (no shared memory variables)
A possible implementation technique: 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.
A possible implementation technique: 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.
Multi-GPU simulation of spin (glass) systems

Simple domain decomposition along one axis
0. 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;
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);
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 (possibly) by using MPI;
               − copies data to the boundaries from the CPU to the GPU;

   stream two  continues to update the bulk;
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 (possibly) by using MPI;
               — copies data to the boundaries from the CPU to the GPU;

   stream two  continues to update the bulk;

4. go to step 2.
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 (possibly) by using MPI;
   - copies data to the boundaries from the CPU to the GPU;
   stream two continues to update the bulk;

4. go to step 2.

The CPU acts as a communication-coprocessor of the GPU!
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.

<table>
<thead>
<tr>
<th># of GPUs</th>
<th>$T_{upd}$ naive</th>
<th>Efficiency</th>
<th>$T_{upd}$ with overlap (w.o.)</th>
<th>Efficiency (w.o.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.66 ns</td>
<td>N.A.</td>
<td>0.66 ns</td>
<td>N.A.</td>
</tr>
<tr>
<td>2</td>
<td>0.45 ns</td>
<td>73%</td>
<td>0.36 ns</td>
<td>91%</td>
</tr>
<tr>
<td>4</td>
<td>0.30 ns</td>
<td>55%</td>
<td>0.19 ns</td>
<td>86%</td>
</tr>
<tr>
<td>8</td>
<td>0.22 ns</td>
<td>37%</td>
<td>0.16 ns</td>
<td>51%</td>
</tr>
</tbody>
</table>

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.
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!
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 and asynchronous copies it is still possible to overlap communication and computation.
• 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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>5.02</td>
<td>63%*</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5.91</td>
<td>74%</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>6.80</td>
<td>85%</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>7.70</td>
<td>96%</td>
</tr>
</tbody>
</table>


(* It was 51% for the $256^3$ test case).
Timings with up to 32 (2070) GPU for a $512^3$ system (CINECA cluster)
Timings with two different threads configurations (CINECA cluster)