Slides and exercises

Check out the lecture notes and example code at

http://users.complexity-coventry.org/~weigel/GPU/

Any questions? Contact me at

Martin.Weigel@mail.com
Monte Carlo simulations

Ising model: GPU implementation

Continuous spins
Consider classical spin models with nn interactions, in particular

**Ising model**

\[ H = -J \sum_{\langle ij \rangle} s_i s_j + H \sum_i s_i, \quad s_i = \pm 1 \]

**Heisenberg model**

\[ H = -J \sum_{\langle ij \rangle} \vec{S}_i \cdot \vec{S}_j + \vec{H} \cdot \sum_i \vec{S}_i, \quad |\vec{S}_i| = 1 \]

**Edwards-Anderson spin glass**

\[ H = -\sum_{\langle ij \rangle} J_{ij} s_i s_j, \quad s_i = \pm 1 \]
Nucleation and phase ordering

$\varepsilon = 1.0$

$\varepsilon = 2.0$
Phase separation in lipid membranes

<table>
<thead>
<tr>
<th>DMPC/DSPC</th>
<th>20:80</th>
<th>20:80</th>
<th>20:80</th>
<th>50:50</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>322 K</td>
<td>321 K</td>
<td>317.5 K</td>
<td>310 K</td>
</tr>
</tbody>
</table>

- **DMPC**
- **DSPC**

**gel**

**fluid**
Markov-chain Monte Carlo

Basic Monte Carlo simulations as all-in-one device for estimating thermal expectation values:
Markov-chain Monte Carlo

Basic Monte Carlo simulations as all-in-one device for estimating thermal expectation values:

- Simple sampling, \( p_{\text{sim}} = \text{const} \): vanishing overlap of trial and target distributions

![Graph showing the probability distribution](image-url)
Markov-chain Monte Carlo

Basic Monte Carlo simulations as all-in-one device for estimating thermal expectation values:

- Simple sampling, $p_{\text{sim}} = \text{const}$: vanishing overlap of trial and target distributions
Markov-chain Monte Carlo

Basic Monte Carlo simulations as all-in-one device for estimating thermal expectation values:

- Simple sampling, $p_{\text{sim}} = \text{const}$: vanishing overlap of trial and target distributions
- Importance sampling $p_{\text{sim}} = p_{\text{eq}}$: trial and target distributions identical

![Graph showing $P(E)$ at $K = 0.4$ for simple and importance sampling](image-url)
Simulate Markov chain \( \{s_i\}_1 \to \{s_i\}_2 \to \ldots, \) such that

\[
\langle O \rangle = \lim_{N \to \infty} \frac{1}{N} \sum_{t=1}^{N} O(\{s_i\}_t).
\]

To achieve this, the transition matrix \( T(\{s_i\} \to \{s'_i\}) \) must satisfy:

(a) **Ergodicity:**

For each \( \{s_i\} \) and \( \{s'_i\} \), there exists \( n > 0 \), such that

\[
T^n(\{s_i\} \to \{s_i\}) > 0.
\]

(b) **Balance:**

\[
\sum_{\{s'_i\}} T(\{s'_i\} \to \{s_i\}) p_\beta(\{s'_i\}) = \sum_{\{s'_i\}} T(\{s_i\} \to \{s'_i\}) p_\beta(\{s_i\}) = p_\beta(\{s_i\})
\]

i.e., \( p_\beta \) is a stationary distribution of the chain.
In practise, these requirements are usually fulfilled by

(a) Choosing an ergodic set of moves (“all possible configurations can be generated”).

(b) Narrowing down the balance to a detailed balance condition,

\[
T(\{s'_i\} \rightarrow \{s_i\}) p_\beta(\{s'_i\}) = T(\{s_i\} \rightarrow \{s'_i\}) p_\beta(\{s_i\})
\]

for each pair of states. A possible form of \( T \) fulfilling this last condition is

\[
T(\{s_i\} \rightarrow \{s'_i\}) = \min \left( 1, \frac{p_\beta(\{s'_i\})}{p_\beta(\{s_i\})} \right)
\]

(Metropolis-Hastings algorithm).
Outline

1. Monte Carlo simulations

2. Ising model: GPU implementation

3. Continuous spins
Markov chain Monte Carlo simulation using single spin flips:

**Algorithm**

1. pick a random spin
2. calculate energy change
   \[ \Delta E = s_i \sum_{j \text{ nn } i} J_{ij} s_j \]
3. draw random number \( r \in [0, 1[ \)
4. accept flip if
   \[ r \leq \min[1, e^{-\beta \Delta E}] \]
NVIDIA architecture
NVIDIA architecture
NVIDIA architecture

Compute model:

- all GPU calculations are encapsulated in dedicated functions (“kernels”)
- two-level hierarchy of a “grid” of thread “blocks”
- mixture of vector and parallel computer:
  - different threads execute the same code on different data (branching possible)
  - different blocks run independently
- threads on the same multiprocessor communicate via shared memory, different blocks are not meant to communicate
- coalescence of memory accesses
NVIDIA architecture

Memory layout:

- **Registers**: each multiprocessor is equipped with several thousand registers with local, zero-latency access.

- **Shared memory**: processors of a multiprocessor have access to a small amount (16–96 KB, depending on GPU generation) of on-chip, very small latency shared memory.

- **Global memory**: large amount (currently up to 16 GB) of memory on separate DRAM chips with access from every thread on each multiprocessor with a latency of several hundred clock cycles.

- **Constant and texture memory**: read-only memories of the same speed as global memory, but cached.

- **Host memory**: cannot be accessed from inside GPU functions, relatively slow transfers.
Design goals:

- a large degree of locality of the calculations, reducing the need for communication between threads
- a large coherence of calculations with a minimum occurrence of divergence of the execution paths of different threads
- a total number of threads significantly exceeding the number of available processing units
- a large overhead of arithmetic operations and shared memory accesses over global memory accesses
For reference, consider the following CPU code for simulating a 2D nearest-neighbor Ising model with the Metropolis algorithm.

```c
for(int i = 0; i < SWEEPS_GLOBAL; ++i) {
    for(int j = 0; j < SWEEPS_EMPTY*SWEEPS_LOCAL; ++j) {
        for(int x = 0; x < L; ++x) {
            for(int y = 0; y < L; ++y) {
                int ide = s[L*y+x]*(s[L*y+(x==0)?L-1:x-1]+s[L*y+(x==L-1)?0:x+1]+s
                        [L*((y==0)?L-1:y-1)+x]+s[L*((y==L-1)?0:y+1)+x]);
                if(ide <= 0 || fabs(RAN(ran)*4.656612e-10f) < boltz[ide]) {
                    s[L*y+x] = -s[L*y+x];
                }
            }
        }
    }
}
```
For reference, consider the following CPU code for simulating a 2D nearest-neighbor Ising model with the Metropolis algorithm.

```c
for(int i = 0; i < SWEEPS_GLOBAL; ++i) {
    for(int j = 0; j < SWEEPS_EMPTY*SWEEPS_LOCAL; ++j) {
        for(int x = 0; x < L; ++x) {
            for(int y = 0; y < L; ++y) {
                int ide = s[L*y+x]*(s[L*y+((x==0)?L-1:x-1)]+s[L*y+((x==L-1)?0:x+1)]+s
                  [L*((y==0)?L-1:y-1)+x]+s[L*((y==L-1)?0:y+1)+x]);
                if(ide <= 0 || fabs(RAN(ran)*4.656612e-10f) < boltz[ide]) {
                    s[L*y+x] = -s[L*y+x];
                }
            }
        }
    }
}
```

- array `s[]` and random number generator must be initialized before
- performs at around 11.6 ns per spin flip on an Intel Q9850 @3.0 GHz
For reference, consider the following CPU code for simulating a 2D nearest-neighbor Ising model with the Metropolis algorithm.

```
for(int i = 0; i < SWEEPS_GLOBAL; ++i) {
    for(int j = 0; j < SWEEPS_EMPTY*SWEEPS_LOCAL; ++j) {
        for(int y = 0; y < L; ++y) {
            for(int x = 0; x < L; ++x) {
                int ide = s[L*y+x]*( s[L*y+((x==0)?L-1:x-1)]+s[L*y+((x==L-1)?0:x+1)]+s
                    [L*((y==0)?L-1:y-1)+x]+s[L*((y==L-1)?0:y+1)+x]);
                if(ide <= 0 || fabs(RAN(ran)*4.656612e-10f) < boltz[ide]) {
                    s[L*y+x] = -s[L*y+x];
                }
            }
        }
    }
}
```

- simple optimization for cache locality improves performance to 7.66 ns per spin flip
Checkerboard decomposition

We need to perform updates on non- (or weakly) interacting sub-domains.
Checkerboard decomposition

We need to perform updates on non- (or weakly) interacting sub-domains. For bi-partite lattices and nearest-neighbor interactions, this leads to a checkerboard decomposition.
We need to perform updates on non- (or weakly) interacting sub-domains. For bi-partite lattices and nearest-neighbor interactions, this leads to a checkerboard decomposition.
We need to perform updates on non- (or weakly) interacting sub-domains. For bi-partite lattices and nearest-neighbor interactions, this leads to a checkerboard decomposition.

Generalizations for more general lattices and longer (but finite) range interactions are straightforward.
A straightforward minimal code translates the CPU version, using one thread to update each spin of a sub-lattice.

**GPU code v1 - driver**

```c
void simulate()
{
    ... declare variables ...

    for(int i = 0; i <= 2*DIM; ++i) boltz[i] = exp(-2*BETA*i);
    cudaMemcpyToSymbol("boltzD", &boltz, (2*DIM+1)*sizeof(float));

    ... setup random number generator ... initialize spins ...

    cudaMalloc((void**)&sD, N*sizeof(spint));
    cudaMemcpy(sD, s, N*sizeof(spint), cudaMemcpyHostToDevice);

    // simulation loops

    dim3 block(BLOCKL/2, BLOCKL); // e.g., BLOCKL = 16
    dim3 grid(GRIDL, GRIDL); // GRIDL = (L/BLOCKL)

    for(int i = 0; i < SWEEPS_GLOBAL; ++i) {
        for(int j = 0; j < SWEEPS_EMPTY; ++j) {
            metro_checkerboard_one<<<grid, block>>>(sD, ranvecD, 0);
            metro_checkerboard_one<<<grid, block>>>(sD, ranvecD, 1);
        }
    }

    cudaMemcpy(s, sD, N*sizeof(spint), cudaMemcpyDeviceToHost);
    cudaFree(sD);
}
```
A straightforward minimal code translates the CPU version, using one thread to update each spin of a sub-lattice.

**GPU code v1 - kernel**

```c
typedef int spin_t;

__global__ void metro_checkerboard_one(spin_t *s, int *ranvec, int offset) {
    int y = blockIdx.y*BLOCKL+threadIdx.y;
    int x = blockIdx.x*BLOCKL+((threadIdx.y+offset)%2)+2*threadIdx.x;
    int xm = (x == 0) ? L-1 : x -1, xp = (x == L-1) ? 0 : x+1;
    int ym = (y == 0) ? L-1 : y -1, yp = (y == L-1) ? 0 : y+1;
    int n = (blockIdx.y*blockDim.y+threadIdx.y)*(L /2)+blockIdx.x*blockDim.x+
        threadIdx.x;

    int ide = s(x,y)*(s(xp,y)+s(xm,y)+s(x,yp)+s(x,ym));
    if(ide <= 0 || fabs(RAN(ranvec[n])*4.656612e-10f) < boltzD[ide]) s(x,y) = -s(x,y);
}
```
We need to perform updates on non- (or weakly) interacting sub-domains. For bi-partite lattices and nearest-neighbor interactions, this leads to a checkerboard decomposition.

Generalizations for more general lattices and longer (but finite) range interactions are straightforward.
A straightforward minimal code translates the CPU version, using one thread to update each spin of a sub-lattice.

**GPU code v1 - kernel**

```c
typedef int spin_t;

__global__ void metro_checkerboard_one(spin_t *s, int *ranvec, int offset)
{
    int y = blockIdx.y*BLOCKL+threadIdx.y;
    int x = blockIdx.x*BLOCKL+((threadIdx.y+offset)%2)+2*threadIdx.x;
    int xm = (x == 0) ? L-1 : x-1 , xp = (x == L-1) ? 0 : x+1;
    int ym = (y == 0) ? L-1 : y-1 , yp = (y == L-1) ? 0 : y+1;
    int n = (blockIdx.y*blockDim.y+threadIdx.y)*(L/2)+blockIdx.x*blockDim.x+
    threadIdx.x;

    int ide = s(x,y)*(s(xp,y)+s(xm,y)+s(x,yp)+s(x,ym));
    if(ide <= 0 || fabs(RAN(ranvec[n])*4.656612e-10f) < boltzD[ide]) s(x,y) = -
    s(x,y);
}
```

- **offset** indicates sub-lattice to update
- periodic boundaries require separate treatment
- use the (cached) constant memory to look up Boltzmann factors
Improving the code

Compare the serial CPU code and the first GPU version:

- CPU code at 7.66 ns per spin flip on Intel Q9850
- GPU code at 0.84 ns per spin flip on Tesla C1060
- \(~\) factor 10, very typical of “naive” implementation

How to improve on this?

A good tool to get ideas is the “CUDA compute visual profiler” part of the CUDA toolkit starting from version 4.0 and/or read:

- CUDA C Programming Guide
- CUDA C Best Practices Guide
Improving the code

Compare the serial CPU code and the first GPU version:

- CPU code at 7.66 ns per spin flip on Intel Q9850
- GPU code at 0.84 ns per spin flip on Tesla C1060
- \( \sim \) factor 10, very typical of “naive” implementation

How to improve on this?

- good tool to get ideas is the “CUDA compute visual profiler”
- part of the CUDA toolkit starting from version 4.0
Improving the code

Compare the serial CPU code and the first GPU version:

- CPU code at 7.66 ns per spin flip on Intel Q9850
- GPU code at 0.84 ns per spin flip on Tesla C1060
- \( \sim \text{factor 10}, \) very typical of “naive” implementation

How to improve on this?

- good tool to get ideas is the “CUDA compute visual profiler”
- part of the CUDA toolkit starting from version 4.0
- and/or read:
  - CUDA C Programming Guide
  - CUDA C Best Practices Guide
Compute visual profiler

- Kernel time = 88.06% of total GPU time
- Memory copy time = 6.3% of total GPU time
- Kernel taking maximum time = `mstore_checked_one` (80.9% of total GPU time)
- Memory copy taking maximum time = `memcpyHost` (10.4% of total GPU time)
- Total overlap time in GPU = 4.8% of total GPU time

**Him(s)**

- Double click on the kernel name in the Summary Table to analyze the kernel
- Analyze kernel `mstore_checked_one`
- Consider using page-locked memory to attain higher bandwidth between host and device memory. Overuse of pinned memory should be avoided as it may reduce overall system performance.
- Refer to the "Page-locked Host Memory" section in the "CUDA C Runtime" chapter of the CUDA C Programming Guide for more details.
Ising model: GPU implementation

Compute visual profiler

![Compute visual profiler](image)

Summary profiling information for the kernel:
- Number of calls: 0
- Minimum GPU time: 0
- Maximum GPU time: 154.48
- Average GPU time: 160.59
- GPU time (us): 88.66
- Global size: (84 48 1)
- Block size: (8 16 1)

Limiting Factor
- Achieved instruction rate: 1.90 (Balanced instruction rate: 3.79)
- Achieved occupancy: 0.69 (Theoretical occupancy: 0.67)
- IPC: 4.11 (Maximum IPC: 3.2)
- Achieved global memory throughput: 76.74 (Peak global memory throughput: 177.41)

Hints:
- The achieved instructions per byte ratio for the kernel is less than the balanced instruction per byte ratio for the device. Hence, the kernel is likely bandwidth limited. For details, click on Memory Throughput Analysis.

Factors that may affect analysis:
- The counters of type SM are collected only for 4 multiprocessors in the chip and the values may be extrapolated to get the behavior of entire GPU assuming equal work distribution. This may result in some inaccuracy in the analysis in some cases.
- The counters for some derived stats are collected in different runs of application. This may cause some inaccuracy in the derived statistics as the blocks scheduled on each multiprocessor may...
CUDA C Best Practices Guide: “Perhaps the single most important performance consideration in programming for the CUDA architecture is the coalescing of global memory accesses. Global memory loads and stores by threads of a warp (of a half warp for devices of compute capability 1.x) are coalesced by the device into as few as one transaction when certain access requirements are met.”
Memory coalescence

CUDA C Best Practices Guide: “Perhaps the single most important performance consideration in programming for the CUDA architecture is the coalescing of global memory accesses. Global memory loads and stores by threads of a warp (of a half warp for devices of compute capability 1.x) are coalesced by the device into as few as one transaction when certain access requirements are met.”

In Fermi and Kepler:

- configurable cache memory of 64 KB, which can be set up as
  - 16 KB L1 cache plus 48 KB of shared memory, or
  - 48 KB L1 cache plus 16 KB of shared memory
  - 32 KB L1 cache plus 32 KB of shared memory (Kepler only)
Memory coalescence

CUDA C Best Practices Guide: “Perhaps the single most important performance consideration in programming for the CUDA architecture is the coalescing of global memory accesses. Global memory loads and stores by threads of a warp (of a half warp for devices of compute capability 1.x) are coalesced by the device into as few as one transaction when certain access requirements are met.”

In Fermi and Kepler:

- configurable cache memory of 64 KB, which can be set up as
  - 16 KB L1 cache plus 48 KB of shared memory, or
  - 48 KB L1 cache plus 16 KB of shared memory
  - 32 KB L1 cache plus 32 KB of shared memory (Kepler only)

- global memory accesses are per default cached in L1 and L2, however caching in L1 can be switched off (-Xptxas -dlcm=cg)
Memory coalescence

CUDA C Best Practices Guide: “Perhaps the single most important performance consideration in programming for the CUDA architecture is the coalescing of global memory accesses. Global memory loads and stores by threads of a warp (of a half warp for devices of compute capability 1.x) are coalesced by the device into as few as one transaction when certain access requirements are met.”

In Fermi and Kepler:

- configurable cache memory of 64 KB, which can be set up as
  - 16 KB L1 cache plus 48 KB of shared memory, or
  - 48 KB L1 cache plus 16 KB of shared memory
  - 32 KB L1 cache plus 32 KB of shared memory (Kepler only)

- global memory accesses are per default cached in L1 and L2, however caching in L1 can be switched off (-xptxas -dlcm=cg)

- cache lines in L1 are 128 bytes, cache lines in L2 32 bytes
Memory coalescence

Access pattern

- cached load
- warp requests aligned to 32 bytes
- accessing consecutive 4-byte words
- addresses lie in one cache line
- efficiency:
  - warp needs 128 bytes
  - 128 bytes are transferred on a cache miss
  - bus utilization 100%

addresses from a warp

Memory addresses

0 32 64 96 128 160 192 224 256 288 320 352 384 416 448
Memory coalescence

Access pattern

- L2 only load
- warp requests aligned to 32 bytes
- accessing consecutive 4-byte words
- addresses lie in 4 adjacent segments
- efficiency:
  - warp needs 128 bytes
  - 128 bytes are transferred on a cache miss
  - bus utilization 100%

addresses from a warp

0  32  64  96  128  160  192  224  256  288  320  352  384  416  448

Memory addresses
Memory coalescence

Access pattern

- cached load
- warp requests aligned to 32 bytes
- accessing permuted 4-byte words
- addresses lie in one cache line
- efficiency:
  - warp needs 128 bytes
  - 128 bytes are transferred on a cache miss
  - bus utilization 100%

addresses from a warp

```
0  32  64  96  128  160  192  224  256  288  320  352  384  416  448
```

Memory addresses
Memory coalescence

Access pattern

- L2 only load
- warp requests aligned to 32 bytes
- accessing permuted 4-byte words
- addresses lie in 4 adjacent segments
- efficiency:
  - warp needs 128 bytes
  - 128 bytes are transferred on a cache miss
  - bus utilization 100%
Memory coalescence

Access pattern

- cached load
- warp requests misaligned to 32 bytes
- accessing consecutive 4-byte words
- addresses fall in two adjacent cache lines
- efficiency:
  - warp needs 128 bytes
  - 256 bytes are transferred on a cache miss
  - bus utilization 50%
Memory coalescence

**Access pattern**
- L2 only load
- warp requests misaligned to 32 bytes
- accessing consecutive 4-byte words
- addresses fall in at most 5 segments
- efficiency:
  - warp needs 128 bytes
  - 160 bytes are transferred on cache misses
  - bus utilization at least 80% (100% for some patterns)
Memory coalescence

**Access pattern**

- cached load
- warp requests are 32 scattered 4-byte words
- addresses fall in $N$ cache lines
- efficiency:
  - warp needs 128 bytes
  - $N \times 128$ bytes are transferred on cache misses
  - bus utilization $1/N$
Memory coalescence

**Access pattern**

- L2 only load
- warp requests are 32 scattered 4-byte words
- addresses fall in $N$ segments
- efficiency:
  - warp needs 128 bytes
  - $N \times 32$ bytes are transferred on cache misses
  - bus utilization $4/N$
We need to perform updates on non- (or weakly) interacting sub-domains. For bi-partite lattices and nearest-neighbor interactions, this leads to a **checkerboard decomposition**.

Generalizations for more general lattices and longer (but finite) range interactions are straightforward.
Coalescence for checkerboard accesses

Re-arrange data for better coalescence: *crinkling* transformation

Re-arrange data for better coalescence: \textit{crinkling} transformation

This corresponds to the mapping

$$(x, y) \mapsto (x, \{(x + y) \text{ mod } 2 \times L + y\}/2)$$

Arrange spins in the crinkled fashion in memory, leading to coalesced accesses to global memory:

```c
__global__ void metro_checkerboard_two(spin_t *s, int *ranvec, int offset)
{
    int n = blockDim.x*blockIdx.x + threadIdx.x;
    int cur = blockDim.x*blockIdx.x + threadIdx.x + offset*(N/2);
    int north = cur + (1-2*offset)*(N/2);
    int east = ((north+1)%L) ? north + 1 : north-L+1;
    int west = (north%L) ? north - 1 : north+L-1;
    int south = (n - (1-2*offset)*L + N/2)%(N/2) + (1-offset)*(N/2);

    int ide = s[cur]*(s[west]+s[north]+s[east]+s[south]);
    if(ide <= 0 || fabs(RAN(ranvec[n])*4.656612e-10f) < boltzD[ide]) s[cur] = -s[cur];
}
```
GPU simulation: second version

Arrange spins in the crinkled fashion in memory, leading to coalesced accesses to global memory:

### GPU code v2 - kernel

```c
__global__ void metro_checkerboard_two(spin_t *s, int *ranvec, int offset)
{
    int n = blockDim.x*blockIdx.x + threadIdx.x;
    int cur = blockDim.x*blockIdx.x + threadIdx.x + offset*(N/2);
    int north = cur + (1-2*offset)*(N/2);
    int east = ((north+1)%L) ? north + 1 : north-L+1;
    int west = (north%L) ? north - 1 : north+L-1;
    int south = (n - (1-2*offset)*L + N/2)%(N/2) + (1-offset)*(N/2);

    int ide = s[cur]*(s[west]+s[north]+s[east]+s[south]);
    if(ide <= 0 || fabs(RAN(ranvec[n])*4.656612e-10f) < boltzD[ide]) s[cur] = -s[cur];
}
```

- accesses to center spins are completely coalesced
- accesses to neighbors reduced to two segments
- at the expense of somewhat more complicated index arithmetic
GPU simulation: second version

- Ising model: GPU implementation
- GPU simulation: second version

M. Weigel (Coventry/Mainz)
Ising model: GPU implementation

GPU simulation: second version

<table>
<thead>
<tr>
<th>Limiting Factor</th>
<th>GPU Timestamp (us)</th>
<th>GPU Time (us)</th>
<th>Instructions issued Type:SM Run#</th>
<th>active warps Type:SM Run:11</th>
<th>active cycles Type:SM Run:12</th>
<th>12 read requests Type:FB</th>
<th>12 read texture requests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4882</td>
<td>82.064</td>
<td>90098</td>
<td>221388</td>
<td>56297</td>
<td>295720</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>4166</td>
<td>79.808</td>
<td>89978</td>
<td>211038</td>
<td>56088</td>
<td>297324</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>4840</td>
<td>60.122</td>
<td>92166</td>
<td>2129101</td>
<td>55611</td>
<td>295530</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>4930</td>
<td>79.576</td>
<td>89591</td>
<td>215012</td>
<td>54704</td>
<td>297640</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>5012</td>
<td>80.032</td>
<td>92338</td>
<td>2136768</td>
<td>55865</td>
<td>298732</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>5092</td>
<td>79.988</td>
<td>89468</td>
<td>2170091</td>
<td>55895</td>
<td>299482</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>5174</td>
<td>80.256</td>
<td>90016</td>
<td>2160665</td>
<td>55244</td>
<td>296004</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>5256</td>
<td>79.576</td>
<td>87277</td>
<td>2177102</td>
<td>55105</td>
<td>297704</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>5338</td>
<td>80.332</td>
<td>91086</td>
<td>2114519</td>
<td>55585</td>
<td>296752</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>5420</td>
<td>79.104</td>
<td>88726</td>
<td>2169846</td>
<td>55044</td>
<td>297656</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>5490</td>
<td>60.064</td>
<td>99172</td>
<td>2135765</td>
<td>55161</td>
<td>294600</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>5560</td>
<td>79.322</td>
<td>90460</td>
<td>2126528</td>
<td>55285</td>
<td>297628</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>5660</td>
<td>79.776</td>
<td>94610</td>
<td>2111170</td>
<td>55378</td>
<td>295630</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>5742</td>
<td>79.488</td>
<td>88045</td>
<td>2140768</td>
<td>54857</td>
<td>297752</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>5822</td>
<td>79.936</td>
<td>90810</td>
<td>2102979</td>
<td>54879</td>
<td>299632</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>5904</td>
<td>79.328</td>
<td>90305</td>
<td>2116269</td>
<td>55155</td>
<td>296864</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>5984</td>
<td>79.84</td>
<td>91403</td>
<td>2173881</td>
<td>55450</td>
<td>294688</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>6064</td>
<td>79.952</td>
<td>89066</td>
<td>2123891</td>
<td>54722</td>
<td>297444</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>6148</td>
<td>79.936</td>
<td>90876</td>
<td>2157385</td>
<td>55620</td>
<td>296420</td>
<td>30</td>
</tr>
</tbody>
</table>
GPU simulation: second version
Use texture for look-up of Boltzmann factors:
Ising model: GPU implementation

GPU simulation: further improvements

- Use texture for look-up of Boltzmann factors:

**GPU code v3 - kernel**

```c
__global__ void metro_checkerboard_three(spin_t *s, int *ranvec, int offset)
{
    int n = blockDim.x*blockIdx.x + threadIdx.x;
    int cur = blockDim.x*blockIdx.x + threadIdx.x + offset*(N/2);
    int north = cur + (1-2*offset)*(N/2);
    int east = ((north+1)%L) ? north + 1 : north-L+1;
    int west = (north%L) ? north - 1 : north+L-1;
    int south = (n - (1-2*offset)*L + N/2)%(N/2) + (1-offset)*(N/2);

    int ide = s[cur]*(s[west]+s[north]+s[east]+s[south]);
    if(fabs(RAN(ranvec[n])*4.656612e-10f) < tex1Dfetch(boltzT, ide+2*DIM)) s[cur] = -s[cur];
}
```

Reduce memory bandwidth pressure by storing spins in narrower variables, e.g., char

M. Weigel (Coventry/Mainz)
GPU simulation: further improvements

- Use texture for look-up of Boltzmann factors:

```
__global__ void metro_checkerboard_three(spin_t *s, int *ranvec, int offset)
{
    int n = blockDim.x*blockIdx.x + threadIdx.x;
    int cur = blockDim.x*blockIdx.x + threadIdx.x + offset*(N/2);
    int north = cur + (1-2*offset)*(N/2);
    int east = ((north+1)%L) ? north + 1 : north-L+1;
    int west = (north%L) ? north - 1 : north+L-1;
    int south = (n - (1-2*offset)*L + N/2)%(N/2) + (1-offset)*(N/2);

    int ide = s[cur]*(s[west]+s[north]+s[east]+s[south]);
    if(fabs(RAN(ranvec[n])*4.656612e-10f) < tex1Dfetch(boltzT, ide+2*DIM)) s[cur] = -s[cur];
}
```

- Reduce memory bandwidth pressure by storing spins in narrower variables, e.g., `char`s:
Use texture for look-up of Boltzmann factors:

```c
__global__ void metro_checkerboard_three(spin_t *s, int *ranvec, int offset)
{
    int n = blockDim.x*blockIdx.x + threadIdx.x;
    int cur = blockDim.x*blockIdx.x + threadIdx.x + offset*(N/2);
    int north = cur + (1-2*offset)*(N/2);
    int east = ((north+1)%L) ? north + 1 : north-L+1;
    int west = (north%L) ? north - 1 : north+L-1;
    int south = (n - (1-2*offset)*L + N/2)%(N/2) + (1-offset)*(N/2);

    int ide = s[cur]*(s[west]+s[north]+s[east]+s[south]);
    if(fabs(RAN(ranvec[n])*4.656612e-10f) < tex1Dfetch(boltzT, ide+2*DIM)) s[cur] = -s[cur];
}
```

Reduce memory bandwidth pressure by storing spins in narrower variables, e.g., `char`:

```c
typedef char spin_t;
```
Reduce thread divergence in stores, improve write coalescence:

```c
__global__ void metro_checkerboard_four ( spin_t *s, int * ranvec , int offset )
{
    int n = blockDim.x* blockIdx.x + threadIdx.x;
    int cur = blockDim.x* blockIdx.x + threadIdx.x + offset *(N /2) ;
    int north = cur + (1 -2* offset )*(N /2) ;
    int east = (( north +1) %L) ? north + 1 : north -L +1;
    int west = ( north %L) ? north - 1 : north +L -1;
    int south = (n - (1 -2* offset )*L + N /2) %(N /2) + (1 - offset )*(N /2) ;
    int ide = s[ cur ]*( s[ west ]+s[ north ]+s[ east ]+s[ south ]);
    int sign = 1;
    if( fabs ( RAN ( ranvec [n]) *4.656612 e -10 f) < tex1Dfetch ( boltzT , ide +2* DIM ))
        sign = -1;
    s[ cur ] = sign *s[ cur ];
}
```

Disable L1 to alleviate effect of scattered load of “south” spin:
Reduce thread divergence in stores, improve write coalescence:

---

**GPU code v4 - kernel**

```c
__global__ void metro_checkerboard_four(spin_t *s, int *ranvec, int offset)
{
    int n = blockDim.x*blockIdx.x + threadIdx.x;
    int cur = blockDim.x*blockIdx.x + threadIdx.x + offset*(N/2);
    int north = cur + (1-2*offset)*(N/2);
    int east = ((north+1)%L) ? north + 1 : north-L+1;
    int west = (north%L) ? north - 1 : north+L-1;
    int south = (n - (1-2*offset)*L + N/2)%(N/2) + (1-offset)*(N/2);

    int ide = s[cur]*(s[west]+s[north]+s[east]+s[south]);
    int sign = 1;
    if(fabs(RAN(ranvec[n])*4.656612e-10f) < tex1Dfetch(boltzT, ide+2*DIM)) sign = -1;
    s[cur] = sign*s[cur];
}
```
Reduce thread divergence in stores, improve write coalescence:

**GPU code v4 - kernel**

```c
__global__ void metro_checkerboard_four(spin_t *s, int *ranvec, int offset)
{
    int n = blockDim.x*blockIdx.x + threadIdx.x;
    int cur = blockDim.x*blockIdx.x + threadIdx.x + offset*(N/2);
    int north = cur + (1-2*offset)*(N/2);
    int east = ((north+1)%L) ? north + 1 : north-L+1;
    int west = (north%L) ? north - 1 : north+L-1;
    int south = (n - (1-2*offset)*L + N/2)%(N/2) + (1-offset)*(N/2);

    int ide = s[cur]*(s[west]+s[north]+s[east]+s[south]);
    int sign = 1;
    if(fabs(RAN(ranvec[n])*4.656612e-10f) < tex1Dfetch(boltzT, ide+2*DIM)) sign
        = -1;
    s[cur] = sign*s[cur];
}
```

Disable L1 to alleviate effect of scattered load of “south” spin:
GPU simulation: further improvements (cont’d)

- Reduce thread divergence in stores, improve write coalescence:

**GPU code v4 - kernel**

```c
__global__ void metro_checkerboard_four(spin_t *s, int *ranvec, int offset)
{
    int n = blockDim.x*blockIdx.x + threadIdx.x;
    int cur = blockDim.x*blockIdx.x + threadIdx.x + offset*(N/2);
    int north = cur + (1-2*offset)*(N/2);
    int east = ((north+1)%L) ? north + 1 : north-L+1;
    int west = (north%L) ? north - 1 : north+L-1;
    int south = (n - (1-2*offset)*L + N/2)% (N/2) + (1-offset)*(N/2);

    int ide = s[cur]*(s[west]+s[north]+s[east]+s[south]);
    int sign = 1;
    if(fabs(RAN(ranvec[n])*4.656612e-10f) < tex1Dfetch(boltzT, ide+2*DIM)) sign = -1;
    s[cur] = sign*s[cur];
}
```

- Disable L1 to alleviate effect of scattered load of “south” spin:

**CUDA compilation**

```
/usr/local/cuda/bin/nvcc -Xptxas -dlcm=cg -arch sm_21 ...
```
Thread divergence

- Scheduling happens on **warps**, groups of 32 threads that
  - share one program counter
  - execute in lock-step (cf. vector processor)
Thread divergence

- Scheduling happens on **warps**, groups of 32 threads that
  - share one program counter
  - execute in lock-step (cf. vector processor)
- However, possibility of thread divergence is built-in to keep flexibility, but leads to serialization and instruction replays.
Thread divergence

```c
if(threadIdx.x < 32) {
    do something
}
else {
    do something else
}
```

No serialization

Warp 0

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Warp 1

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Thread divergence

No serialization

if(threadIdx.x < 32) {
    do something
}

} else {
    do something else
}

...
Thread divergence

No serialization

if(threadIdx.x < 32) {
    do something
} else {
    do something else
}

...
Thread divergence

With serialization

if (threadIdx.x < 16) {
    do something
}
else {
    do something else
}

...
Thread divergence

With serialization

if(threadIdx.x < 16) {
    do something
} else {
    do something else
}

...
Thread divergence

With serialization

if(threadIdx.x < 16) {
    do something
} else {
    do something else
}

...
Thread divergence

With serialization

```
if (threadIdx.x < 16) {
    do something
}
else {
    do something else
}
```

Warp 0

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Warp 1

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

...
Global memory version: performance

- Performance v2 (crinkling), 0.595 ns/flip on Tesla C1060.
- Performance v2.5 (char variables), 0.391 ns/flip on Tesla C1060.
- Performance v3 (texture), 0.145 ns/flip on GTX 480.
- Performance v4 (coalesced writes, no L1), 0.119 ns/flip on GTX 480.

Now performing at a speed-up of $\sim 65$ vs. CPU at this system size.

What else? Use shared memory!
Global memory version: performance

- Performance v2 (crinkling), 0.595 ns/flip on Tesla C1060.
- Performance v2.5 (char variables), 0.391 ns/flip on Tesla C1060.
Global memory version: performance

- Performance v2 (crinkling), 0.595 ns/flip on Tesla C1060.
- Performance v2.5 (char variables), 0.391 ns/flip on Tesla C1060.
- Performance v3 (texture), 0.145 ns/flip on GTX 480.

Now performing at a speed-up of $\sim 65$ vs. CPU at this system size.

What else? Use shared memory!

M. Weigel (Coventry/Mainz)
Global memory version: performance

- Performance v2 (crinkling), 0.595 ns/flip on Tesla C1060.
- Performance v2.5 (char variables), 0.391 ns/flip on Tesla C1060.
- Performance v3 (texture), 0.145 ns/flip on GTX 480.
- Performance v4 (coalesced writes, no L1), 0.119 ns/flip on GTX 480.
Global memory version: performance

- Performance v2 (crinkling), 0.595 ns/flip on Tesla C1060.
- Performance v2.5 (char variables), 0.391 ns/flip on Tesla C1060.
- Performance v3 (texture), 0.145 ns/flip on GTX 480.
- Performance v4 (coalesced writes, no L1), 0.119 ns/flip on GTX 480.
- Now performing at a speed-up of ~ 65 vs. CPU at this system size.
Global memory version: performance

- Performance v2 (crinkling), 0.595 ns/flip on Tesla C1060.
- Performance v2.5 (char variables), 0.391 ns/flip on Tesla C1060.
- Performance v3 (texture), 0.145 ns/flip on GTX 480.
- Performance v4 (coalesced writes, no L1), 0.119 ns/flip on GTX 480.
- Now performing at a speed-up of ~65 vs. CPU at this system size.

What else? Use shared memory!
Double checkerboard decomposition

- (red) large tiles: thread blocks
- (red) small tiles: individual threads

Load one large tile (plus boundary) into shared memory, perform several spin updates per tile.
Double checkerboard decomposition

- (red) large tiles: thread blocks
- (red) small tiles: individual threads
- Load one large tile (plus boundary) into shared memory
- Perform several spin updates per tile
GPU simulation: shared-memory version

Execution configuration is slightly changed since only a quarter of the spins is updated at each time:

```
void simulate()
{
    ... declare variables ... setup RNG ... initialize spins ...

cudaMalloc((void**)&sD, N*sizeof(spin_t));
cudaMemcpy(sD, s, N*sizeof(spin_t), cudaMemcpyHostToDevice);

    // simulation loops

dim3 block5(BLOCKL, BLOCKL/2); // e.g., BLOCKL = 16
dim3 grid5(GRIDL, GRIDL/2);   // GRIDL = (L/BLOCKL)

   for(int i = 0; i < SWEEPS_GLOBAL; ++i) {
      for(int j = 0; j < SWEEPS_EMPTY; ++j) {
         metro_checkerboard_five<<<grid5, block5>>>(sD, ranvecD, 0);
         metro_checkerboard_five<<<grid5, block5>>>(sD, ranvecD, 1);
      }
   }

    ... clean up ...
}
```
**GPU code v5 - kernel 1/2**

```c
__global__ void metro_checkerboard_five(spin_t *s, int *ranvec, unsigned int offset)
{
    unsigned int n = threadIdx.y*BLOCKL+threadIdx.x;

    unsigned int xoffset = blockIdx.x*BLOCKL;
    unsigned int yoffset = (2*blockIdx.y+(blockIdx.x+offset)%2)*BLOCKL;

    __shared__ spin_t sS[(BLOCKL+2)*(BLOCKL+2)];

    sS[(2*threadIdx.y+1)*(BLOCKL+2)+threadIdx.x+1] = s[(yoffset+2*threadIdx.y)*L+xoffset+threadIdx.x];
    sS[(2*threadIdx.y+2)*(BLOCKL+2)+threadIdx.x+1] = s[(yoffset+2*threadIdx.y+1)*L+xoffset+threadIdx.x];
    if(threadIdx.y == 0)
        sS[threadIdx.x+1] = (yoffset == 0) ? s[(L-1)*L+xoffset+threadIdx.x] : s[(yoffset-1)*L+xoffset+threadIdx.x];
    if(threadIdx.y == BLOCKL/2-1)
        sS[(BLOCKL+1)*(BLOCKL+2)+threadIdx.x+1] = (yoffset == L-BLOCKL) ? s[xoffset+threadIdx.x] :
                                         s[(yoffset+BLOCKL)*L+xoffset+threadIdx.x];
    if(threadIdx.x == 0) {
        if(blockIdx.x == 0) {
            sS[(2*threadIdx.y+1)*(BLOCKL+2)] = s[(yoffset+2*threadIdx.y)*L+(L-1)];
            sS[(2*threadIdx.y+2)*(BLOCKL+2)] = s[(yoffset+2*threadIdx.y+1)*L+(L-1)];
        }
        else {
            sS[(2*threadIdx.y+1)*(BLOCKL+2)] = s[(yoffset+2*threadIdx.y)*L+xoffset-1];
            sS[(2*threadIdx.y+2)*(BLOCKL+2)] = s[(yoffset+2*threadIdx.y+1)*L+xoffset-1];
        }
    } if(threadIdx.x == BLOCKL-1) {
        if(blockIdx.x == GRIDL-1) {
            sS[(2*threadIdx.y+1)*(BLOCKL+2)+BLOCKL+1] = s[(yoffset+2*threadIdx.y)*L];
            sS[(2*threadIdx.y+2)*(BLOCKL+2)+BLOCKL+1] = s[(yoffset+2*threadIdx.y+1)*L];
        }
        else {
            sS[(2*threadIdx.y+1)*(BLOCKL+2)+BLOCKL+1] = s[(yoffset+2*threadIdx.y)*L+xoffset+BLOCKL];
            sS[(2*threadIdx.y+2)*(BLOCKL+2)+BLOCKL+1] = s[(yoffset+2*threadIdx.y+1)*L+xoffset+BLOCKL];
        }
    }
    ...
}```
Shared memory

- **organization:**
  - **pre-Fermi:** 16 banks, 32-bit wide
  - **Fermi:** 32 banks, 32-bit wide
  - **Kepler** and onwards: 32 banks, 32-bit or 64-bit wide chunks
  - successive 4-byte words belong to different banks
Shared memory

- **organization:**
  - **pre-Fermi:** 16 banks, 32-bit wide
  - **Fermi:** 32 banks, 32-bit wide
  - **Kepler** and onwards: 32 banks, 32-bit or 64-bit wide chunks
  - successive 4-byte words belong to different banks

- **performance:** 4 bytes per bank per two clock cycles per SM

- shared memory accesses are issued per warp (32 threads)
Shared memory

- organization:
  - pre-Fermi: 16 banks, 32-bit wide
  - Fermi: 32 banks, 32-bit wide
  - Kepler and onwards: 32 banks, 32-bit or 64-bit wide chunks
  - successive 4-byte words belong to different banks

- performance: 4 bytes per bank per two clock cycles per SM

- shared memory accesses are issued per warp (32 threads)

- serialization: if $n$ threads of 32 access different 4-byte words in the same bank, $n$ serial accesses are issued
Shared memory

- organization:
  - **pre-Fermi**: 16 banks, 32-bit wide
  - **Fermi**: 32 banks, 32-bit wide
  - **Kepler** and onwards: 32 banks, 32-bit or 64-bit wide chunks
  - successive 4-byte words belong to different banks

- **performance**: 4 bytes per bank per two clock cycles per SM

- shared memory accesses are issued per warp (32 threads)

- **serialization**: if \( n \) threads of 32 access different 4-byte words in the same bank, \( n \) serial accesses are issued

- **multicast**: \( n \) threads can access the same word/bits of the same word in one fetch (Fermi and up)
Shared memory

- **organization:**
  - **pre-Fermi:** 16 banks, 32-bit wide
  - **Fermi:** 32 banks, 32-bit wide
  - **Kepler** and onwards: 32 banks, 32-bit or 64-bit wide chunks
  - successive 4-byte words belong to different banks

- **performance:** 4 bytes per bank per two clock cycles per SM

- shared memory accesses are issued per warp (32 threads)

- **serialization:** if $n$ threads of 32 access different 4-byte words in the same bank, $n$ serial accesses are issued

- **multicast:** $n$ threads can access the same word/bits of the same word in one fetch (Fermi and up)

- standard trick for avoiding conflicts: appropriate **padding**
Shared memory (cont’d)

(Source: NVIDIA)
Shared memory (cont’d)

(Source: NVIDIA)
__global__ void metro_checkerboard_five(spin_t *s, int *ranvec, unsigned int offset)
{
    unsigned int n = threadIdx.y*BLOCKL+threadIdx.x;

    unsigned int xoffset = blockIdx.x*BLOCKL;
    unsigned int yoffset = (2*blockIdx.y+(blockIdx.x+offset)%2)*BLOCKL;

    __shared__ spin_t sS[(BLOCKL+2)*(BLOCKL+2)];

    sS[(2*threadIdx.y+1)*(BLOCKL+2)+threadIdx.x+1] = s[(yoffset+2*threadIdx.y)*L+xoffset+threadIdx.x];
    sS[(2*threadIdx.y+2)*(BLOCKL+2)+threadIdx.x+1] = s[(yoffset+2*threadIdx.y+1)*L+xoffset+threadIdx.x];

    if(threadIdx.y == 0)
        sS[threadIdx.x+1] = (yoffset == 0) ? s[(L-1)*L+xoffset+threadIdx.x] : s[(yoffset-1)*L+xoffset+threadIdx.x];
    if(threadIdx.y == BLOCKL/2 -1)
        sS[(BLOCKL+1)*(BLOCKL+2)+threadIdx.x+1] = (yoffset == L-BLOCKL) ? s[xoffset+threadIdx.x] : s[(yoffset+BLOCKL)*L+xoffset+threadIdx.x];
    if(threadIdx.x == 0) {
        if(blockIdx.x == 0) {
            sS[(2*threadIdx.y+1)*(BLOCKL+2)] = s[(yoffset+2*threadIdx.y)*L+(L-1)];
            sS[(2*threadIdx.y+2)*(BLOCKL+2)] = s[(yoffset+2*threadIdx.y+1)*L+(L-1)];
        }
        else {
            sS[(2*threadIdx.y+1)*(BLOCKL+2)] = s[(yoffset+2*threadIdx.y)*L+xoffset-1];
            sS[(2*threadIdx.y+2)*(BLOCKL+2)] = s[(yoffset+2*threadIdx.y+1)*L+xoffset-1];
        }
    }
    if(threadIdx.x == BLOCKL-1) {
        if(blockIdx.x == GRIDL-1) {
            sS[(2*threadIdx.y+1)*(BLOCKL+2)+BLOCKL+1] = s[(yoffset+2*threadIdx.y)*L];
            sS[(2*threadIdx.y+2)*(BLOCKL+2)+BLOCKL+1] = s[(yoffset+2*threadIdx.y+1)*L];
        }
        else {
            sS[(2*threadIdx.y+1)*(BLOCKL+2)+BLOCKL+1] = s[(yoffset+2*threadIdx.y)*L+xoffset+BLOCKL];
            sS[(2*threadIdx.y+2)*(BLOCKL+2)+BLOCKL+1] = s[(yoffset+2*threadIdx.y+1)*L+xoffset+BLOCKL];
        }
    }
    ...
}
Thread synchronization

- to ensure that all threads of a block have reached the same point in execution, use
  
  __syncthreads()

  in kernel code
Thread synchronization

- To ensure that all threads of a block have reached the same point in execution, use __syncthreads() in kernel code.

- Threads in a warp execute in lock-step, so in-warp synchronization is unnecessary, such that __syncthreads() in the following code is superfluous,

  ```
  if(tid < 32){ ... __syncthreads(); ... }
  ```
Thread synchronization

- to ensure that all threads of a block have reached the same point in execution, use
  
  ```c
  __syncthreads()
  ```
  
in kernel code

- threads in a warp execute in lock-step, so in-warp synchronization is unnecessary, such that `__syncthreads()` in the following code is superfluous,

  ```c
  if(tid < 32){ ... __syncthreads(); ... }
  ```

- if `__syncthreads()` is omitted, however, used shared or global memory must be declared `volatile` for writes to be visible to other threads

- there a number of more specific synchronization instructions,

  ```c
  __threadfence()
  __threadfence_block()
  __threadfence_system()
  ```

  for ensuring that previous memory transactions have completed
**GPU simulation: shared-memory version**

**GPU code v5 - kernel 2/2**

```c
__syncthreads();

unsigned int ran = ranvec[(blockIdx.y*GRIDL+blockIdx.x)*THREADS+n];

unsigned int x = threadIdx.x;
unsigned int y1 = (threadIdx.x%2)+2*threadIdx.y;
unsigned int y2 = ((threadIdx.x+1)%2)+2*threadIdx.y;

for(int i = 0; i < SWEEPS_LOCAL; ++i) {
  int ide = sS(x,y1)*(sS(x-1,y1)+sS(x,y1-1)+sS(x+1,y1)+sS(x,y1+1));
  if(MULT*(*(unsigned int*)(&RAN(ran))) < tex1Dfetch(boltzT, ide+2*DIM)) {
    sS(x,y1) = -sS(x,y1);
  }
  __syncthreads();

  ide = sS(x,y2)*(sS(x-1,y2)+sS(x,y2-1)+sS(x+1,y2)+sS(x,y2+1));
  if(MULT*(*(unsigned int*)(&RAN(ran))) < tex1Dfetch(boltzT, ide+2*DIM)) {
    sS(x,y2) = -sS(x,y2);
  }
  __syncthreads();
}

s[(yoffset+2*threadIdx.y)*L+xoffset+threadIdx.x] = sS[(2*threadIdx.y+1)*(BLOCKL+2)+threadIdx.x+1];
s[(yoffset+2*threadIdx.y+1)*L+xoffset+threadIdx.x] = sS[(2*threadIdx.y+2)*(BLOCKL+2)+threadIdx.x+1];
ranvec[(blockIdx.y*GRIDL+blockIdx.x)*THREADS+n] = ran;
```
Performance

How to assess performance?

- what to compare to (one CPU core, whole CPU, SMP system, ...) here: Tesla C1060 vs. Intel QuadCore (Yorkfield) @ 3.0 GHz/6 MB
- for really fair comparison: optimize CPU code for cache alignment, use SSE instructions etc.
- ignore measurements, since spin flips per $\mu$s, (ns, ps) is well-established unit for spin systems
Performance

How to assess performance?

- what to compare to (one CPU core, whole CPU, SMP system, ...)
  here: Tesla C1060 vs. Intel QuadCore (Yorkfield) @ 3.0 GHz/6 MB
- for really fair comparison: optimize CPU code for cache alignment, use SSE instructions etc.
- ignore measurements, since spin flips per $\mu s$, (ns, ps) is well-established unit for spin systems

Example: Metropolis simulation of 2D Ising system

- use 32-bit linear congruential generator (see next lecture)
- use multi-hit updates to amortize share-memory load overhead
- need to play with tile sizes to achieve best throughput
2D Ising ferromagnet

![Graph showing the relationship between spin flip time and system size for different temperatures.](image)

- **Spin flip time**: $t_{\text{flip}}$ [ns]
- **System size**: $L$
- **Temperatures**: $T = 4, 8, 16, 32$

- **Legend**:
  - $T = 4$ (black triangles)
  - $T = 8$ (blue triangles)
  - $T = 16$ (red squares)
  - $T = 32$ (green crosses)
2D Ising ferromagnet

\[ t_{\text{flip}} [\text{ns}] \]

- \( L \)

CPU

- \( k = 1 \)
- \( k = 100 \)
- \( k = 100, \text{Fermi} \)
2D Ising ferromagnet

![Graph showing the relationship between $t_{\text{flip}}$ and $L$. The graph includes data points for different algorithms: Fibonacci, LCG64, LCG32 with measurements, and LCG32.](image-url)
Is it correct?

Detailed balance,

\[ T(\{s_i\} \rightarrow \{s_i\})p_\beta(\{s_i\}) = T(\{s_i\} \rightarrow \{s_i\})p_\beta(\{s_i\}), \]

only holds for \textit{random} updates.
**Is it correct?**

Detailed balance,

\[
T(\{s_i'\} \rightarrow \{s_i\})p_\beta(\{s_i'\}) = T(\{s_i\} \rightarrow \{s_i'\})p_\beta(\{s_i\}),
\]

only holds for *random* updates.

Usually applied sequential update merely satisfies (global) *balance*,

\[
\sum_{\{s'_i\}} T(\{s'_i\} \rightarrow \{s_i\})p_\beta(\{s'_i\}) = \sum_{\{s'_i\}} T(\{s_i\} \rightarrow \{s'_i\})p_\beta(\{s_i\}) = p_\beta(\{s_i\})
\]
Is it correct?

Detailed balance,

\[ T(\{s_i\} \rightarrow \{s_i\})p_\beta(\{s_i\}) = T(\{s_i\} \rightarrow \{s_i\})p_\beta(\{s_i\}), \]

only holds for random updates.

Usually applied sequential update merely satisfies (global) balance,

\[ \sum_{\{s_i\}} T(\{s_i\} \rightarrow \{s_i\})p_\beta(\{s_i\}) = \sum_{\{s_i\}} T(\{s_i\} \rightarrow \{s_i\})p_\beta(\{s_i\}) = p_\beta(\{s_i\}) \]

Similarly for checkerboard update. Could restore detailed balance on the level of several sweeps, though:

\[ AAAA(M)AAAAABBBBB(M)BBBBAAAAA(M)AAAAABBBBB(M)BBBB \cdots \]
A closer look

Comparison to exact results:
A closer look

Temperature dependent spin flip times:

\[ \beta \]

\[ 0.2 \quad 0.3 \quad 0.4 \quad 0.5 \quad 0.6 \quad 0.7 \]

\[ 0.4 \quad 0.45 \quad 0.5 \quad 0.6 \quad 0.7 \]

\[ \text{ns} \]

M. Weigel (Coventry/Mainz)

spin models I
A closer look

Temperature dependent spin flip times:

![Graph showing temperature dependent spin flip times]

- Temperature range: $0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7$
- Spin flip times: $0.2, 0.25, 0.3, 0.4, 0.5, 0.6, 0.7$

- Data points marked with ▼ for temperature $\beta = 100$

M. Weigel (Coventry/Mainz)
Temperature dependent spin flip times:
A closer look

Autocorrelation times:

![Graph showing autocorrelation times vs. beta for different values of interaction time and CPU.](image-url)
A closer look

Real time to create independent spin configuration:

![Graph depicting the real time to create independent spin configuration for different values of $\beta$. The graph shows the relationship between the inverse temperature $\tilde{\tau}$ (in ns) and the inverse temperature $\beta$. There are lines for different CPU speeds: 1, 5, 10, 50, and 100, each represented by a different color. The graph indicates a peak in the real time at a certain value of $\beta$.](image-url)
Outline

1. Monte Carlo simulations
2. Ising model: GPU implementation
3. Continuous spins
Heisenberg model

Maximum performance around 100 ps per spin flip for Ising model (vs. around 10 ns on CPU). What about continuous spins, i.e., float instead of int variables?
Continuous spins

Heisenberg model

Maximum performance around 100 ps per spin flip for Ising model (vs. around 10 ns on CPU). What about continuous spins, i.e., float instead of int variables?

⇒ use same decomposition, but now floating-point computations are dominant:

- CUDA before Fermi was not 100% IEEE compliant
- single-precision computations are supposed to be fast, double precision (supported since Fermi) much slower
- for single precision, normal ("high precision") and extra-fast, device-specific versions of sin, cos, exp etc. are provided
There are two types of special function implementations:

- C library implementations: \( \sin(x) \), \( \cos(x) \), \( \exp(x) \), etc.
- hardware intrinsics: \(__sinf(x)\), \(__cosf(x)\), \(__expf(x)\), etc.

Intrinsics are fast, but have lower accuracy
available for float only, not double
There are two types of special function implementations:

- C library implementations: \( \sin(x) \), \( \cos(x) \), \( \exp(x) \), etc.
- hardware intrinsics: \( \_\_\text{sinf}(x) \), \( \_\_\text{cosf}(x) \), \( \_\_\text{expf}(x) \), etc.

intrinsics are fast, but have lower accuracy
available for float only, not double

Also, there are a number of additional intrinsics, e.g., \( \_\_\text{sincosf}(x) \), \( \_\_\text{frcp}_\text{rz}(x) \)
There are two types of special function implementations:

- C library implementations: \( \sin(x) \), \( \cos(x) \), \( \exp(x) \), etc.
- hardware intrinsics: \_\_sinf(x), \_\_cosf(x), \_\_expf(x), etc.

intrinsics are fast, but have lower accuracy

available for float only, not double

Also, there are a number of additional intrinsics, e.g., \_\_sincosf(x), \_\_frcp_rz(x)

Double precision is significantly slower than single precision:

- e.g., \( 8 \times \) slower for Tesla and gaming cards, \( 2 \times \) slower for Fermi and up
- use mixed precision whenever possible
Heisenberg model: performance

- $t_{\text{flip}}$ [ns]
- $L$
- CPU
double, Fermi
float, fast, Fermi

$10^{-1}$ $10^0$ $10^1$ $10^2$

$32$ $64$ $128$ $256$ $512$ $1024$ $2048$ $4096$ $8192$ $16384$
Heisenberg model: stability

Performance results:

- **CPU**: 183 ns (single or double) per spin flip
- **GPU**: 0.74 ns (single), 0.30 ns (fast single) resp. 4.7 ns (double) per spin flip
Continuous spins

Heisenberg model: stability

Performance results:

- CPU: 183 ns (single or double) per spin flip
- GPU: 0.74 ns (single), 0.30 ns (fast single) resp. 4.7 ns (double) per spin flip

How about stability?
Heisenberg model: stability

Performance results:

- **CPU:** 183 ns (single or double) per spin flip
- **GPU:** 0.74 ns (single), 0.30 ns (fast single) resp. 4.7 ns (double) per spin flip

How about stability?

- no drift of spin normalization
- no deviations in averages from reference implementation (at least at low precision)
- more subtle effects: non-uniform trial vectors etc.
This lecture

We have now covered in detail various implementations of Ising and Heisenberg model simulations with local updates, learning about a number of relevant performance issues on the way. You should be able to write a CUDA program with decent performance for a problem in your field.

Next lecture

In lecture 4, we will have a look at the issue of random number generators suitable for massively parallel environments, discuss more advanced simulation algorithms and a few tricks used for implementing them, for example atomic operations.

Reading

Some of this is based on my publications on the subject, for example


Summary and outlook

This lecture

We have now covered in detail various implementations of Ising and Heisenberg model simulations with local updates, learning about a number of relevant performance issues on the way. You should be able to write a CUDA program with decent performance for a problem in your field.

Next lecture

In lecture 4, we will have a look at the issue of random number generators suitable for massively parallel environments, discuss more advanced simulation algorithms and a few tricks used for implementing them, for example atomic operations.

Reading

Summary and outlook

This lecture

We have now covered in detail various implementations of Ising and Heisenberg model simulations with local updates, learning about a number of relevant performance issues on the way. You should be able to write a CUDA program with decent performance for a problem in your field.

Next lecture

In lecture 4, we will have a look at the issue of random number generators suitable for massively parallel environments, discuss more advanced simulation algorithms and a few tricks used for implementing them, for example atomic operations.

Reading

Some of this is based on my publications on the subject, for example

Ising code

The code in ising_v1.cu simulates the 2D Ising model on CPU and on GPU using the simplest implementation.

Tasks:

- Compile and run it. Inspect the timings, possibly on different GPUs. (How to select a given GPU in your code?)
- Visualize the result: uncomment code that saves the final configuration and create a plot with \texttt{asy -f pdf plot_conf.asy}
- Add some of the improvements discussed in this lecture. In most cases, this also requires changes to the driver code (execution configuration, memory layout etc.), not just insertion of the kernel code.
- Compare timings for different kernel versions and different block sizes etc.
- Change the code for simulations of the Heisenberg model. Check the stability.