

Simulating spin models on GPU

Lecture 2: Local updates for Ising and Heisenberg models

Martin Weigel

Applied Mathematics Research Centre, Coventry University, Coventry, United Kingdom and
Institut für Physik, Johannes Gutenberg-Universität Mainz, Germany

IMPRS School 2012: GPU Computing,
Wroclaw, Poland, October 30, 2012



Outline

1 Monte Carlo simulations

2 Ising model: GPU implementation

3 Continuous spins

Spin models

Consider classical spin models with nn interactions, in particular

Ising model

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

Heisenberg model

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

Edwards-Anderson spin glass

$$\mathcal{H} = - \sum_{\langle ij \rangle} J_{ij} s_i s_j, \quad s_i = \pm 1$$

Spin models — Applications

Nucleation and phase ordering

$\varepsilon = 1.0$

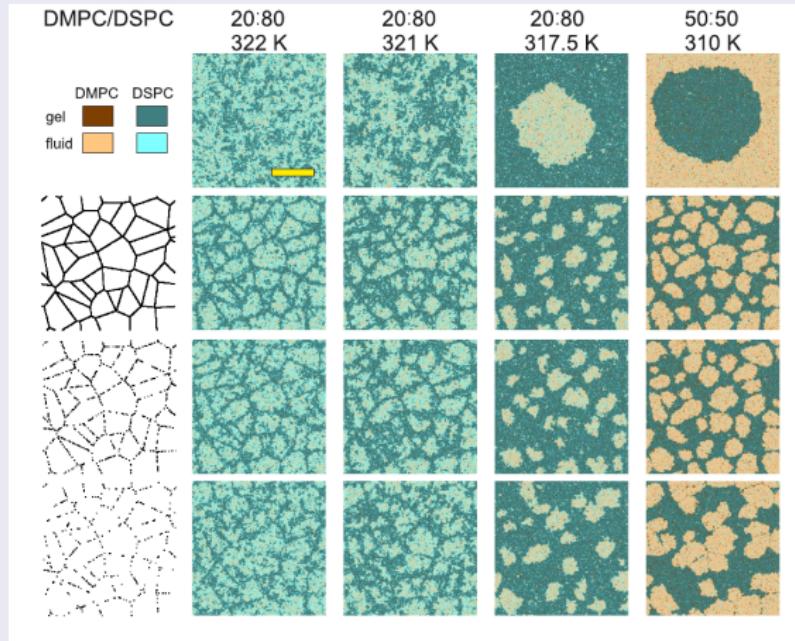


$\varepsilon = 2.0$



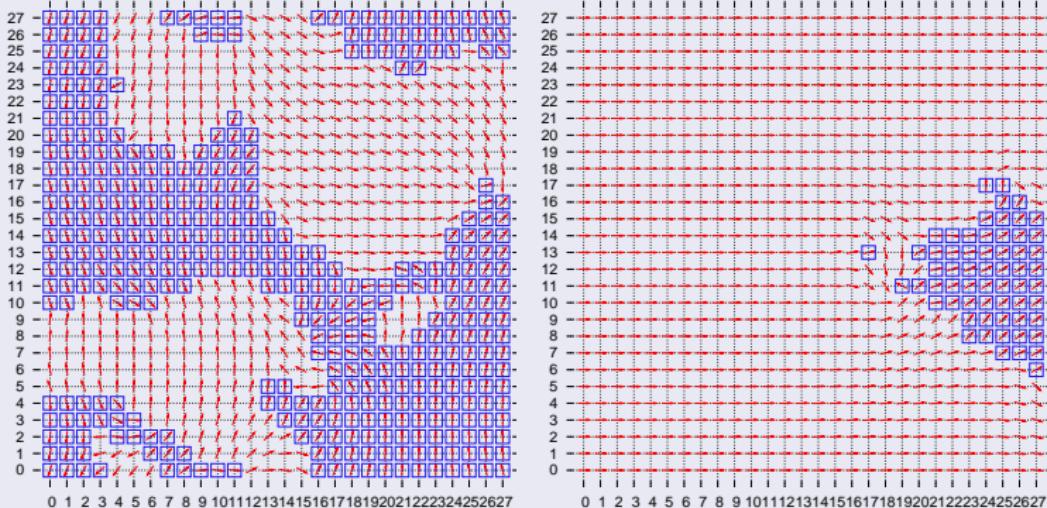
Spin models — Applications

Phase separation in lipid membranes



Spin models — Applications

Quenched disorder in magnetic systems



MCMC simulations

Markov-chain Monte Carlo

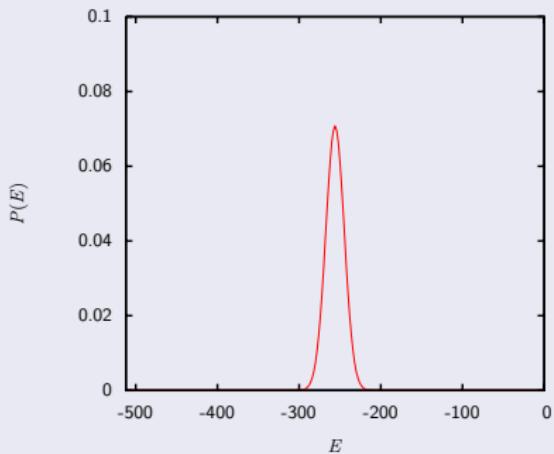
Basic Monte Carlo simulations as
all-in-one device for estimating thermal
expectation values:

MCMC simulations

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

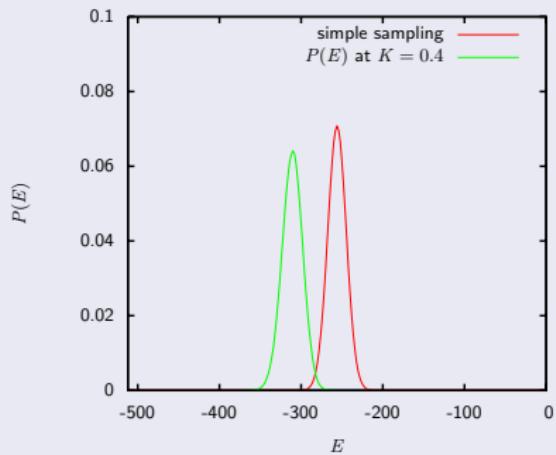


MCMC simulations

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

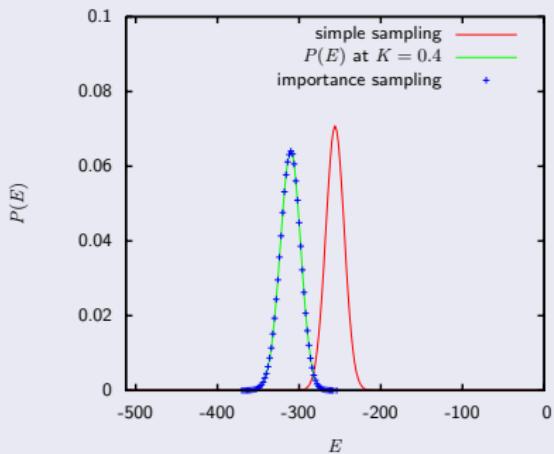


MCMC simulations

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



MCMC simulations

Markov chains

Simulate Markov chain $\{s_i\}_1 \rightarrow \{s_i\}_2 \rightarrow \dots$, such that

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

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

(a) **Ergodicity:**

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

$$T^n(\{s_i\} \rightarrow \{s_i\}) > 0.$$

(b) **Balance:**

$$\sum_{\{s'_i\}} T(\{s'_i\} \rightarrow \{s_i\}) p_\beta(\{s'_i\}) = \sum_{\{s'_i\}} T(\{s_i\} \rightarrow \{s'_i\}) p_\beta(\{s_i\}) = p_\beta(\{s_i\})$$

i.e., p_β is a stationary distribution of the chain.

MCMC simulations

Metropolis algorithm

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).

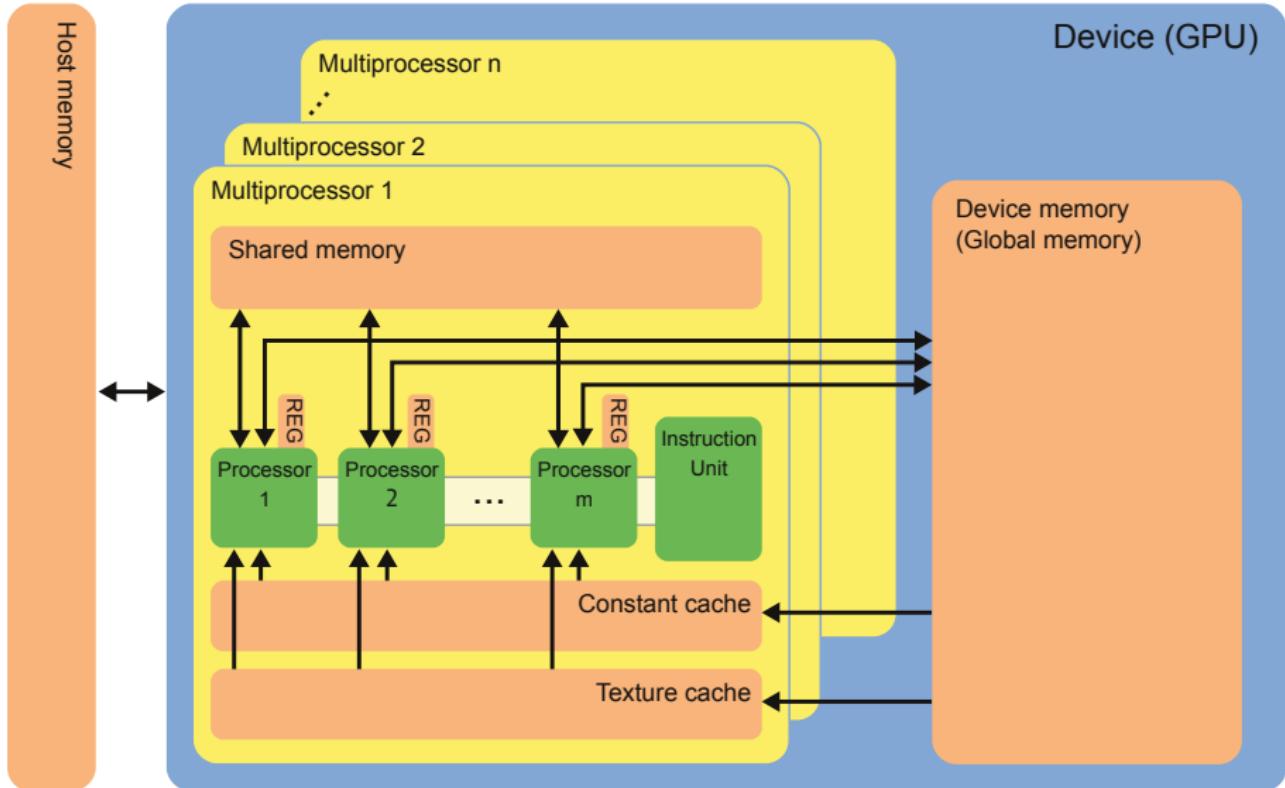
Outline

1 Monte Carlo simulations

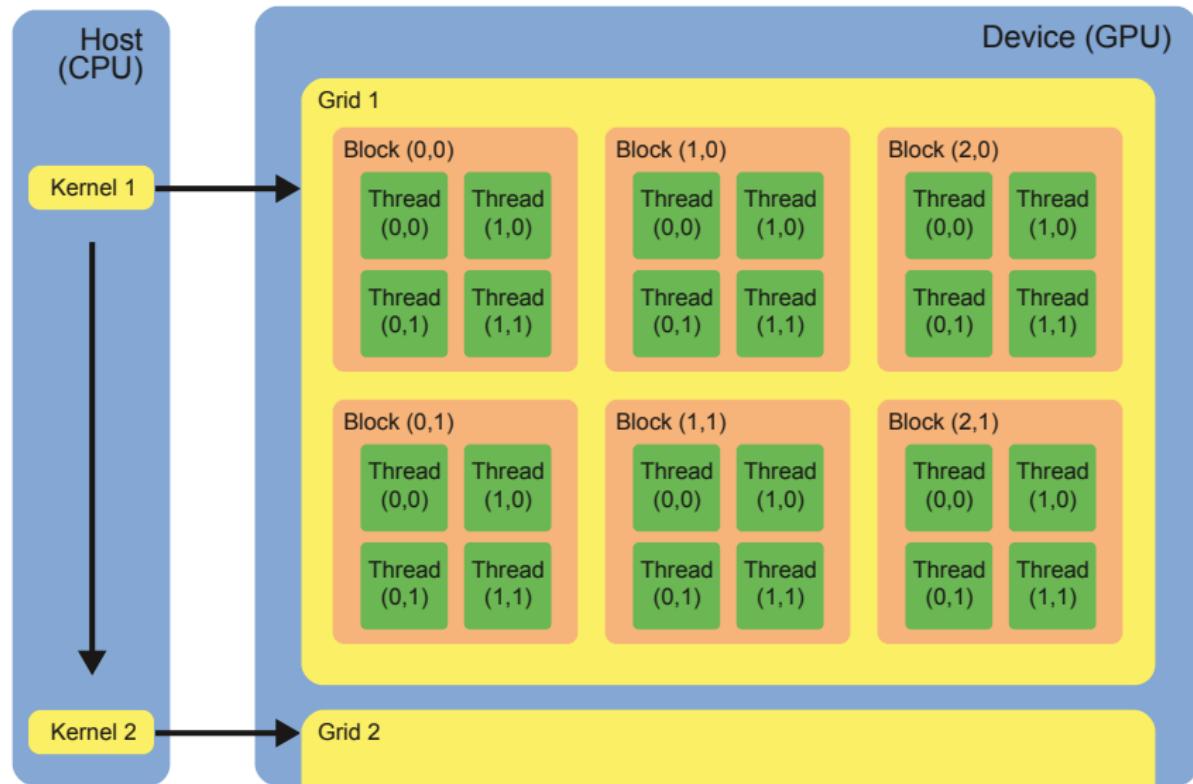
2 Ising model: GPU implementation

3 Continuous spins

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 a small amount (16 KB for Tesla, 48 KB for Fermi) of on chip, very small latency shared memory
- *Global memory*: large amount (currently up to 4 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

NVIDIA architecture

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

CPU implementation

For reference, consider the following CPU code for simulating a 2D nearest-neighbor Ising model with the Metropolis algorithm.

CPU code

```
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];
                }
            }
        }
    }
}
```

CPU implementation

For reference, consider the following CPU code for simulating a 2D nearest-neighbor Ising model with the Metropolis algorithm.

CPU code

```
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

CPU implementation

For reference, consider the following CPU code for simulating a 2D nearest-neighbor Ising model with the Metropolis algorithm.

CPU code

```
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

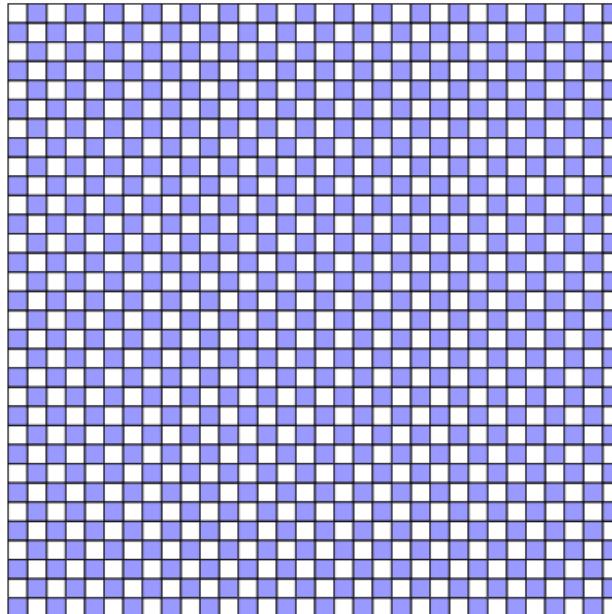
We need to perform updates on non- (or weakly) interacting sub-domains.

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](#).

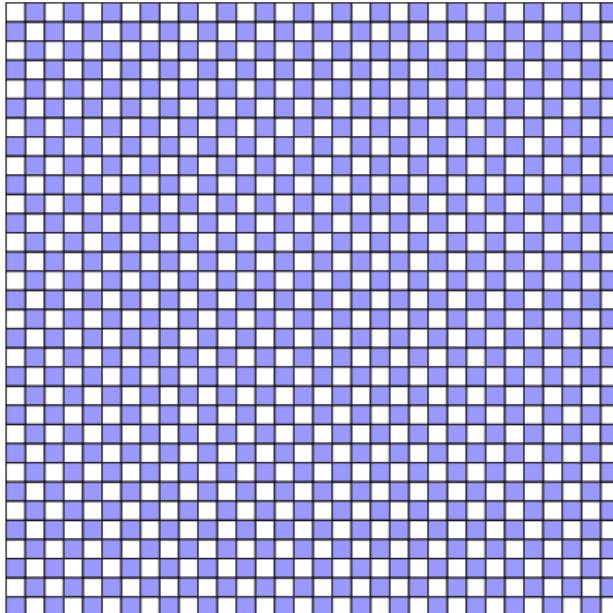
Checkerboard decomposition

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](#).



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.

GPU simulation: first version

A straightforward minimal code translates the CPU version, using one thread to update each spin of a sub-lattice.

GPU code v1 - driver

```
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(spin_t));
    cudaMemcpy(sD, s, N*sizeof(spin_t), 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(spin_t), cudaMemcpyDeviceToHost);
    cudaFree(sD);
}
```

GPU simulation: first version

A straightforward minimal code translates the CPU version, using one thread to update each spin of a sub-lattice.

GPU code v1 - kernel

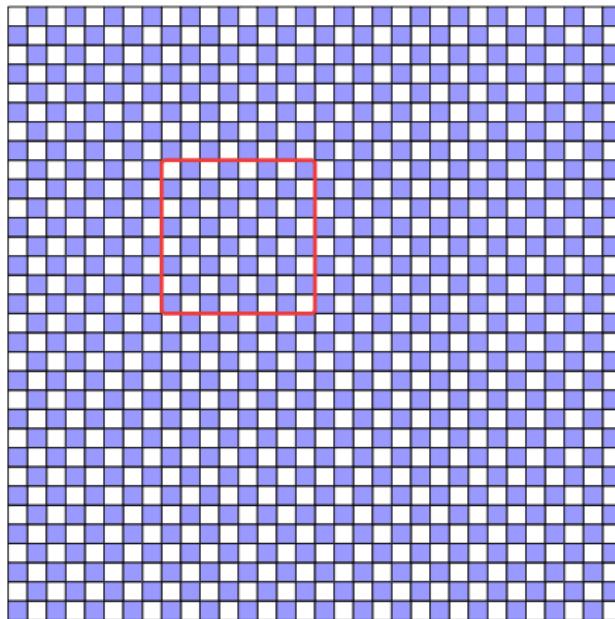
```
typedef int spin_t;

__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);
}
```

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.

GPU simulation: first version

A straightforward minimal code translates the CPU version, using one thread to update each spin of a sub-lattice.

GPU code v1 - kernel

```

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

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
- ~ 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

Neabx - weigel@loki1:477 - loki1

untitled - Compute Visual Profiler - [Session1 - Device_0 - Context_0 [CUDA]]

Sessions

- Session1
 - Device_0
 - Context_0 [CUDA]

Profiler Output Summary Table

Method	#Calls	GPU time (us)	^ %GPU time	glob mem read throughput	glob mem write throughput	glob mem over
1 metro_checkerboard_one	400	64390.5	88	39.2103	37.5338	76.7442
2 memcpyHtoD	5	2488.96	3.4			
3 memcpyDtoH	2	1896	2.59			

Session1::Device_0::Context_0

- Kernel time = 88.00 % of total GPU time
- Memory copy time = 6.0 % of total GPU time
- Kernel taking maximum time = **metro_checkerboard_one** (88.0% of total GPU time)
- Memory copy taking maximum time = **memcpyHtoD** (3.4% of total GPU time)
- Total overlap time in GPU = 26.9 micro sec. (0.0% of total GPU time)

Hint(s)

- Double click on the kernel name in the Summary Table to analyze the kernel
- Analyze kernel **metro_checkerboard_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.

Analysis

Output

```
Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'shared load Type:SH Rm:S' having all zero values is hidden.
Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'shared load Type:SH Rm:B' having all zero values is hidden.
Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'L2 cache texture requests' having all zero values is hidden.
Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'tex cache misses Type:RM' having all zero values is hidden.
Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'L2 cache texture memory read throughput(GB/s)' having all zero values is hidden.
Session1 - Device_0 - Context_0 [CUDA] Profiler table column 'L2 cache write hit ratio(%)' having all zero values is hidden.
Profiler counter 'tex cache requests' required for 'texture hit rate %' calculations is not available.
```

10:14 pm Tue, 16 Oct

Compute visual profiler

Neatbx - weigel@loki1:477 - loki1
 metro_checkerboard_one analysis - [Session1 - Device_0 - Context_0]

Analysis for kernel metro_checkerboard_one on device GeForce GTX 480

Summary profiling information for the kernel:

- Number of CUDA API calls: 400
- Minimum GPU time(μs): 154.08
- Maximum GPU time(μs): 165.44
- Average GPU time(μs): 160.98
- GPU time (%): 88.00
- Grid size: [64 64 1]
- Block size: [8 16 1]

Limiting Factor

- Achieved Instruction Per Byte Ratio: 1.50 (Balanced Instruction Per Byte Ratio: 3.79)
- Achieved Occupancy: 0.50 (Theoretical Occupancy: 0.67)
- IPC: 1.41 (Maximum IPC: 2)
- Achieved global memory throughput: 76.74 (Peak global memory throughput(GB/s): 177.41)

Hint(s)

- 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 memory bandwidth limited**. For details, click on [Memory Throughput Analysis](#).

Factors that may affect analysis

- The counters of type SM are collected only for 1 multiprocessor in the chip and the values are 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

Show all columns								
Limiting Factor Identification	GPU Timestamp (μs)	GPU Time (μs)	instructions issued Type:SM Run:8	active warps Type:SM Run:11	active cycles Type:SM Run:12	I2 read requests Type:FB	I2 read texture requests	
Memory Throughput Analysis	1 3758	154.08	145597	3018841	105102	956212	0	
	2 3916	157.12	151211	3089686	108293	1008784	0	
	3 4074	155.904	150217	3080393	107588	99256	0	
	4 4232	156.352	150764	3080747	107727	1006856	0	
	5 4390	155.104	150387	3081222	107403	99644	0	
	6 4544	156.64	150311	3083767	107088	1008572	0	
	7 4702	156.768	151921	3045189	106783	996988	0	
	8 4862	157.12	147389	3071588	107359	1007664	0	
	9 5020	156.768	150629	3064523	106445	994872	0	
	10 5178	156.416	150983	3046108	106786	1004992	0	
	11 5336	157.28	149700	3056799	106369	996688	0	
	12 5494	159.136	149762	3067709	107373	99464	0	
	13 5654	158.368	150673	3030993	106775	996596	0	

← →

Neatbx - weigel@loki1:477 - loki1
 metro_checkerboard_one analysis - [Session1 - Device_0 - Context_0] untitled - Compute Visual Profiler - lsing/s : bash 10:15 pm Tue, 16 Oct

Compute visual profiler

Neatbx - weigel@loki1:477 - loki1
metro_checkerboard_one analysis - [Session1 - Device_0 - Context_0]

Analysis

Memory Throughput Analysis for kernel metro_checkerboard_one on device GeForce GTX 480

- Kernel requested global memory read throughput(GB/s): 76.03
- Kernel requested global memory write throughput(GB/s): 14.08
- Kernel requested global memory throughput(GB/s): 90.11
- L2 cache texture memory read throughput(GB/s): 0.00
- L2 cache global memory read throughput(GB/s): 198.12
- L2 cache global memory write throughput(GB/s): 37.60
- L2 cache global memory throughput(GB/s): 235.71
- L2 cache read hit ratio(%): 80.21
- L2 cache write hit ratio(%): 0.17
- Global memory excess load(%): 61.62
- Global memory excess stall(%): 62.54
- Achieved global memory read throughput(GB/s): 39.21
- Achieved global memory write throughput(GB/s): 37.53
- Achieved global memory throughput(GB/s): 76.74
- Peak global memory throughput(GB/s): 177.41

The following derived statistic(s) cannot be computed as required counters are not available:

- L1 cache read throughput(GB/s)
- L1 cache global hit ratio (%)
- Texture cache memory throughput(GB/s)
- Texture cache hit rate(%)
- Local memory bus traffic(%)

Hints(s)

- Memory access pattern is not coalesced.** The kernel requested throughput and achieved global memory throughput can be different because of following two reasons:
 - Scattered/misaligned pattern: not all transaction bytes are utilized.
 - Broadcast: the same transaction serves many requests (due to sector size, cache line size and caching).

Refer to the "Global Memory" section in the "Performance Guidelines" chapter of the CUDA C Programming Guide for more details.

Limiting Factor Identification		Show all columns					
		glid instructions 16bit Type:S/W Run:2	glid instructions 32bit Type:S/W Run:2	glid instructions 64bit Type:S/W Run:2	glid instructions 128bit Type:S/W Run:2	gst instructions 8bit Type:S/W Run:3	gst instructions 16bit Type:S/W Run:3
Memory Throughput Analysis	1	0	2813062	0	0	0	0
	2	0	2967610	0	0	0	0
	3	0	3006192	0	0	0	0
	4	0	3023040	0	0	0	0
	5	0	3034348	0	0	0	0
	6	0	3041335	0	0	0	0
	7	0	3045879	0	0	0	0

10:17 pm
Tue, 16 Oct

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.* ”

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%



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%



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%**



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

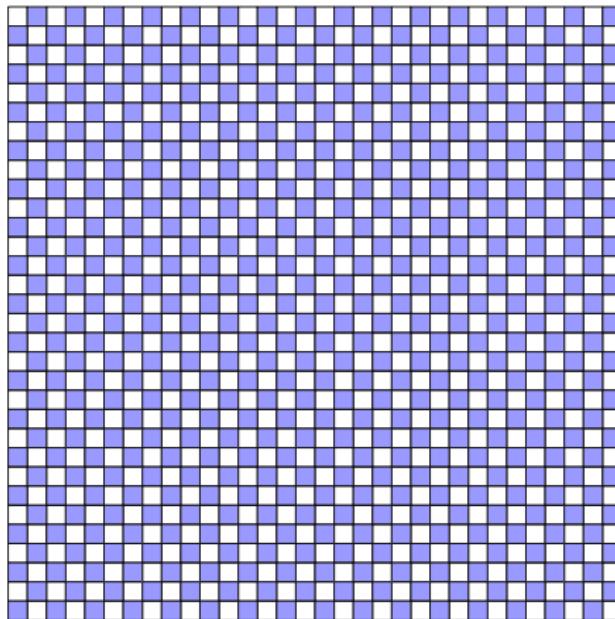
Access pattern

- 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 $4/N$



Checkerboard decomposition

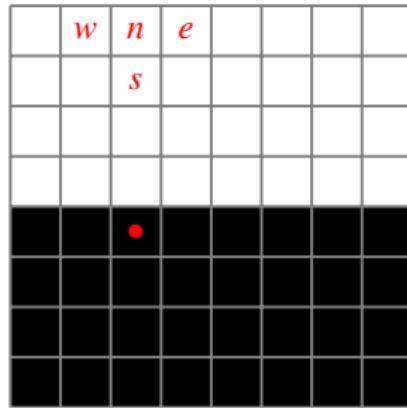
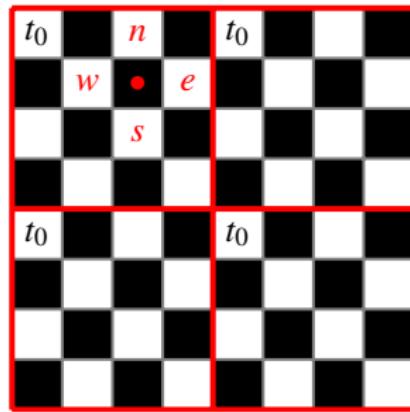
We need to perform updates on non- (or weakly) interacting sub-domains. For bi-partite lattices and nearest-neighbor interactions, this leads to a [checkerboard decomposition](#).



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

Coalescence for checkerboard accesses

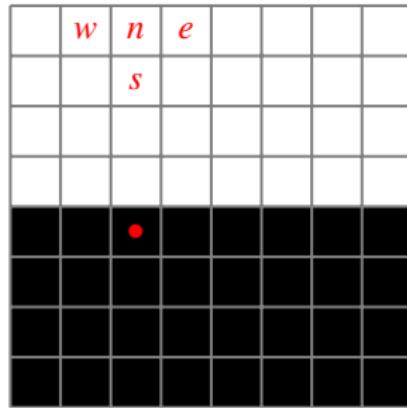
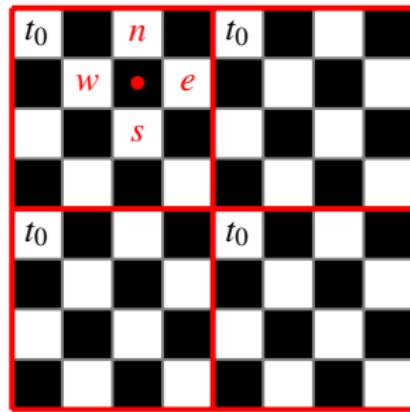
Re-arrange data for better coalescence: *crinkling* transformation



(E. E. Ferrero *et. al.*, Comput. Phys. Commun. 183, 1578 (2012))

Coalescence for checkerboard accesses

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



(E. E. Ferrero *et. al.*, Comput. Phys. Commun. 183, 1578 (2012))

This corresponds to the mapping

$$(x, y) \mapsto (x, \{[(x + y) \bmod 2] \times L + y\}/2)$$

GPU simulation: second version

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

GPU code v2 - kernel

```
__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:

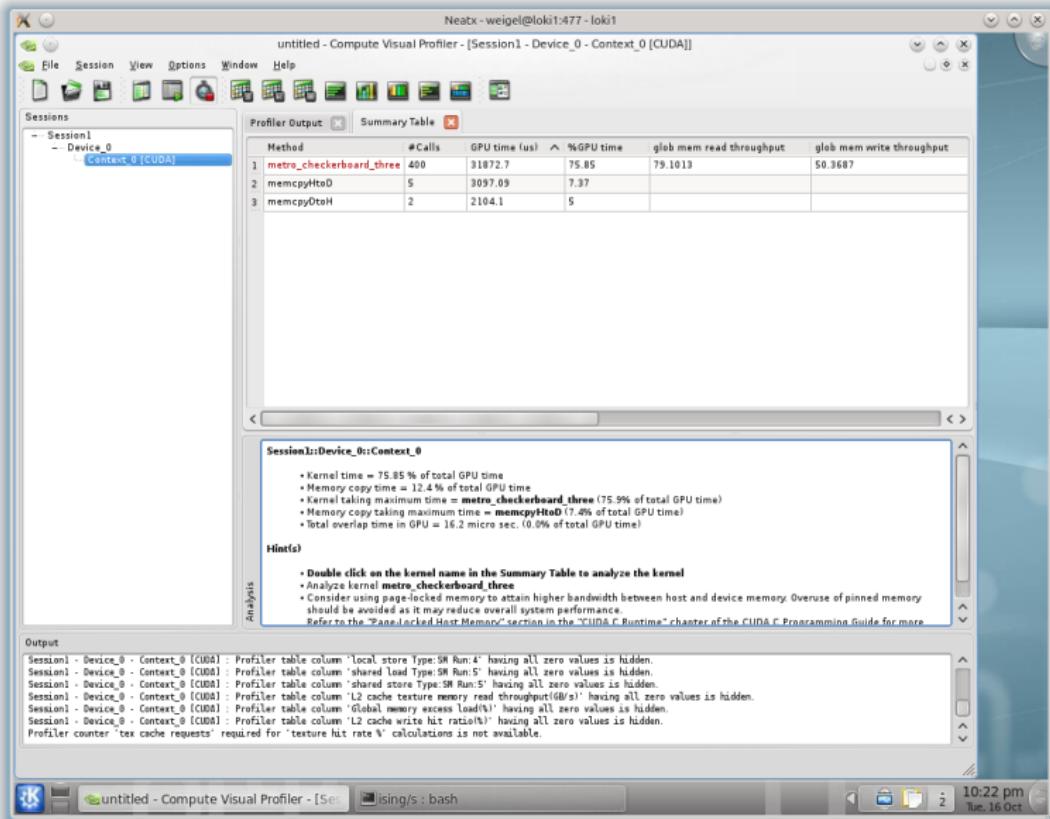
GPU code v2 - kernel

```
__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



GPU simulation: second version

Neatbx - weigel@loki1:477 - loki1
 metro_checkerboard_three analysis - [Session1 - Device_0 - Context_0]

Analysis

Limiting Factor
 Achieved Instruction Per Byte Ratio: 3.18 (Balanced Instruction Per Byte Ratio: 3.79)
 Achieved Occupancy: 0.81 (Theoretical Occupancy: 1.00)
 IPC: 1.63 (Maximum IPC: 2.1)
 Achieved global memory throughput: 129.47 (Peak global memory throughput(GB/s): 177.41)

Hint(s)
 • 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 memory bandwidth limited**. For details, click on [Memory Throughput Analysis](#).

Factors that may affect analysis
 • The counters of type SM are collected only for 1 multiprocessor in the chip and the values are 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 be different for each run and for some applications the behavior changes for each run.
 • The derived statistics instruction per byte ratio and IPC assume that all instructions are single precision floating point instructions. If the application uses double precision floating point instructions then the limiting factor predicted here may be incorrect.

Show all columns								
Limiting Factor Identification		GPU Timestamp (us)	GPU Time (us)	instructions issued Type:SM Run:8	active warps Type:SM Run:11	active cycles Type:SM Run:12	I2 read requests Type:FB	
Memory Throughput Analysis	1	4682	82.464	90098	2213393	56297	295720	30
	2	4766	79.808	89978	2111038	55488	297324	30
Instruction Throughput Analysis	3	4848	80.512	92116	2129181	55611	295500	30
	4	4930	79.616	89501	2153012	54784	297840	30
Occupancy Analysis	5	5012	80.032	92138	2195768	55865	294792	30
	6	5092	79.68	89463	2170051	55385	299432	30
	7	5174	80.096	90015	2160665	55244	296004	30
	8	5256	79.2	87827	2177102	55186	297164	30
	9	5338	80.032	91484	2114919	55356	296792	30
	10	5420	79.104	88725	2165046	55064	297056	30
	11	5498	80.064	89712	2135786	55161	294608	30
	12	5580	79.232	90168	2120528	55286	297628	30
	13	5660	79.776	90410	21111170	55378	295400	30
	14	5742	79.488	88065	2147368	54957	297732	30
	15	5822	79.936	90818	2182979	54878	296032	30
	16	5904	79.328	88109	2116289	55156	296864	30
	17	5984	79.84	91403	2173381	55450	296088	30
	18	6066	79.552	89464	2123393	54722	297444	30
	19	6148	79.936	90976	2157395	55430	296420	30

File View Analysis

10:23 pm Tue, 16 Oct

GPU simulation: second version

Neabx - weigel@loki1:477 - loki1
metro_checkerboard_three analysis - [Session1 - Device_0 - Context_0]

Analysis

Memory Throughput Analysis for kernel metro_checkerboard_three on device GeForce GTX 480

- Kernel requested global memory read throughput(GB/s): 166.37
- Kernel requested global memory write throughput(GB/s): 34.77
- Kernel requested global memory throughput(GB/s): 201.15
- L2 cache texture memory read throughput(GB/s): 0.01
- L2 cache global memory read throughput(GB/s): 119.24
- L2 cache global memory write throughput(GB/s): 50.37
- L2 cache global memory throughput(GB/s): 169.61
- L2 cache read hit ratio(%): 33.67
- L2 cache write hit ratio(%): 0.00
- Global memory excess load(%): -39.54
- Global memory excess stall(%): 30.97
- Achieved global memory read throughput(GB/s): 79.10
- Achieved global memory write throughput(GB/s): 50.37
- Achieved global memory throughput(GB/s): 129.47
- Peak global memory throughput(GB/s): 177.41

The following derived statistic(s) cannot be computed as required counters are not available:

- L1 cache read throughput(GB/s)
- L1 cache global hit ratio (%)
- Texture cache memory throughput(GB/s)
- Texture cache hit rate(%)
- Local memory bus traffic(%)

Hints(s)

- Consider using shared memory as a user managed cache for frequently accessed global memory resources.
Refer to the "Shared Memory" section in the "CUDA C Runtime" chapter of the CUDA C Programming Guide for more details.
- The achieved global memory throughput is low compared to the peak global memory throughput. To achieve closer to peak global memory throughput, try to
 - Launch enough threads to hide memory latency (check occupancy analysis);
 - ... (truncated)

Limiting Factor Identification		Show all columns					
		GPU Timestamp (us)	GPU Time (us)	dynamic shared memory per block (bytes)	static shared memory per block (bytes)	gid instructions @bit Type: S/W Run:2	gid instructions @bit Type: S/W Run:3
Memory Throughput Analysis	1	4682	82.464	0	0	0	0
	2	4766	79.808	0	0	0	0
	3	4848	80.512	0	0	0	0
	4	4930	79.616	0	0	0	0
	5	5012	80.032	0	0	0	0
	6	5092	79.68	0	0	0	0
	7	5174	80.096	0	0	0	0

10:24 pm
Tue, 16 Oct

metro_checkerboard_three analysis - untitled - Compute Visual Profiler - lsing/s : bash

GPU simulation: further improvements

- Use texture for look-up of Boltzmann factors:

GPU simulation: further improvements

- Use texture for look-up of Boltzmann factors:

GPU code v3 - kernel

```
__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];
}
```

GPU simulation: further improvements

- Use texture for look-up of Boltzmann factors:

GPU code v3 - kernel

```
__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:

GPU simulation: further improvements

- Use texture for look-up of Boltzmann factors:

GPU code v3 - kernel

```
--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:

GPU code v3 - kernel

```
typedef char spin_t;
```

GPU simulation: further improvements (cont'd)

- Reduce thread divergence in stores, improve write coalescence:

GPU simulation: further improvements (cont'd)

- Reduce thread divergence in stores, improve write coalescence:

GPU code v4 - kernel

```
__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];
}
```

GPU simulation: further improvements (cont'd)

- Reduce thread divergence in stores, improve write coalescence:

GPU code v4 - kernel

```
--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

```
--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

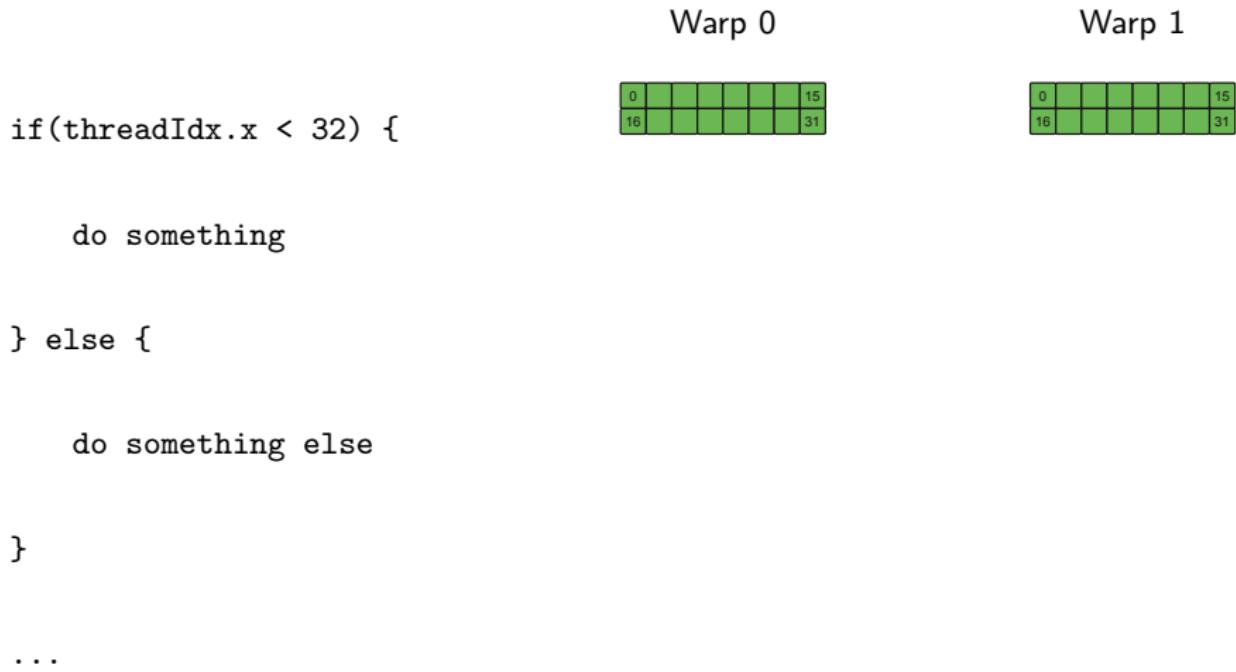
- Scheduling happens on **warps**, groups of 32 threads that
 - share one program counter
 - execute in lock-step (cf. vector processor)

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

No serialization



Thread divergence

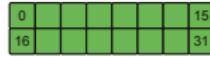
No serialization

Warp 0

Warp 1

```
if(threadIdx.x < 32) {
```

```
    do something
```

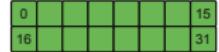


```
} else {
```

```
    do something else
```

```
}
```

...



Thread divergence

No serialization

Warp 0

Warp 1

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

0					15
16					31

0					15
16					31

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
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Warp 1

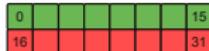
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Thread divergence

With serialization

```
if(threadIdx.x < 16) {
```

```
    do something
```



```
} else {
```

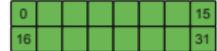
```
    do something else
```

```
}
```

```
...
```

Warp 0

Warp 1



Thread divergence

With serialization

```
if(threadIdx.x < 16) {
```

```
    do something
```

```
} else {
```

```
    do something else
```

```
}
```

```
...
```

Warp 0

Warp 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Thread divergence

With serialization

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

Warp 0

Warp 1

0					15
16					31

0					15
16					31

Global memory version: performance

- Performance v2 (crinkling), 0.595 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.

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.

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.

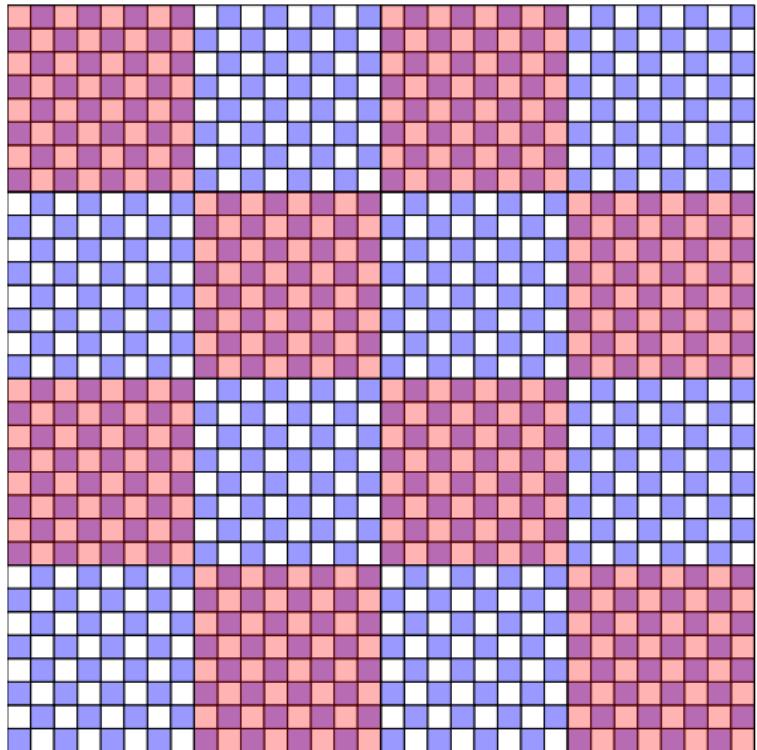
Global memory version: performance

- 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**!

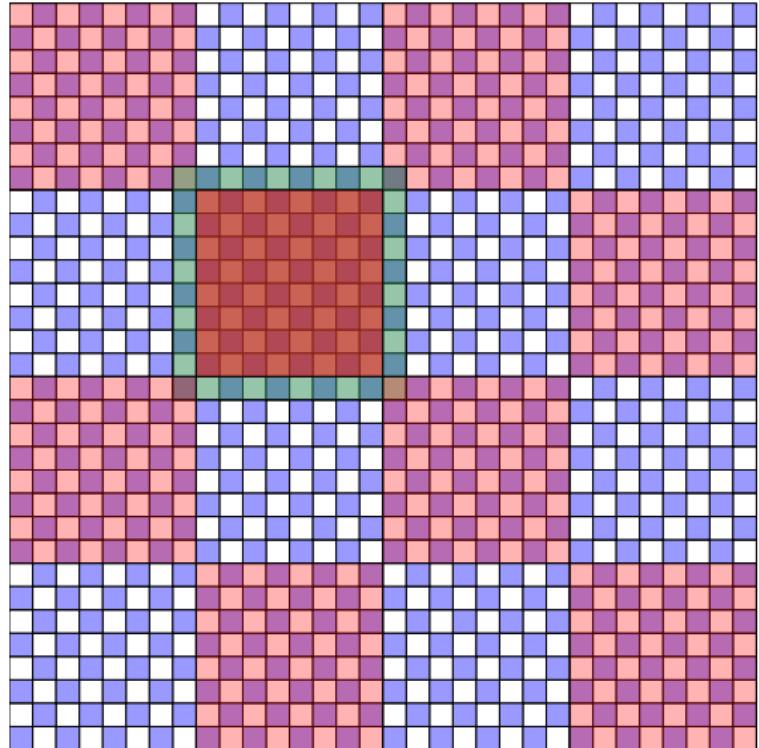
Double checkerboard decomposition

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



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:

GPU code v5 - driver

```
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 simulation: shared-memory version

GPU code v5 - kernel 1/2

```
__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: 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: 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: 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

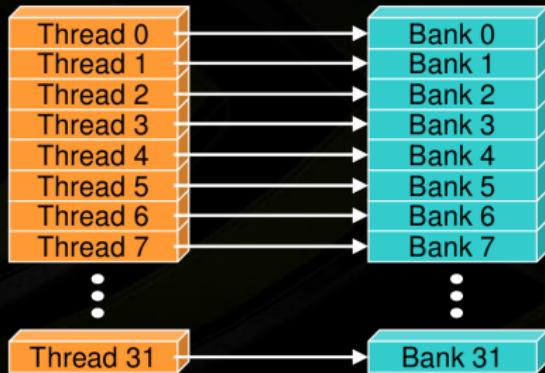
- organization:
 - pre-Fermi: 16 banks, 32-bit wide
 - Fermi: 32 banks, 32-bit wide
 - Kepler: 32 banks, 32-bit or 64-bit wide chunks
 - successive 4-byte words belong to different banks
- performance: 4 bytes per bank per two clock cycles per SM
- shared memory accesses are issued per warp (32 threads)
- 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

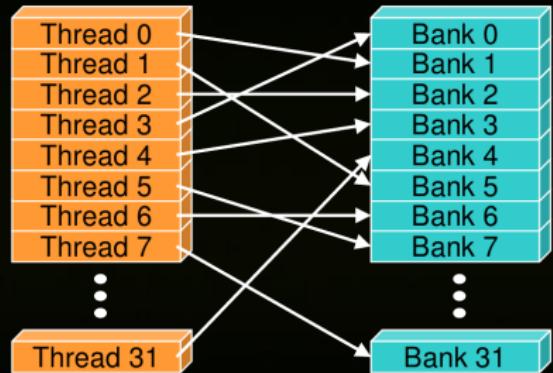
- organization:
 - pre-Fermi: 16 banks, 32-bit wide
 - Fermi: 32 banks, 32-bit wide
 - Kepler: 32 banks, 32-bit or 64-bit wide chunks
 - successive 4-byte words belong to different banks
- performance: 4 bytes per bank per two clock cycles per SM
- shared memory accesses are issued per warp (32 threads)
- 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)

- No Bank Conflicts



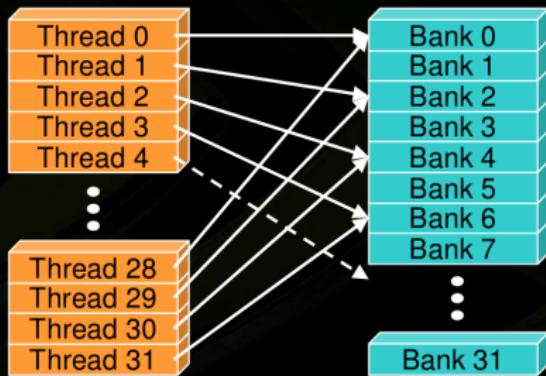
- No Bank Conflicts



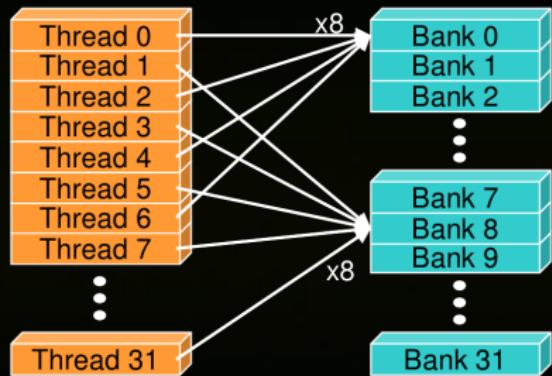
(Source: NVIDIA)

Shared memory (cont'd)

- **2-way Bank Conflicts**



- **8-way Bank Conflicts**



(Source: NVIDIA)

GPU simulation: shared-memory version

GPU code v5 - kernel 1/2

```
__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

```
__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(); ... }
```

- if `__syncthreads()` is omitted, however, used shared or global memory must be declared **volatile** for writes to be visible to other threads
- there are a number of more specific synchronization instructions,

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

for ensuring that previous memory transactions have completed

GPU simulation: shared-memory version

GPU code v5 - kernel 2/2

```

__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 μs , (ns, ps) is well-established unit for spin systems

Performance

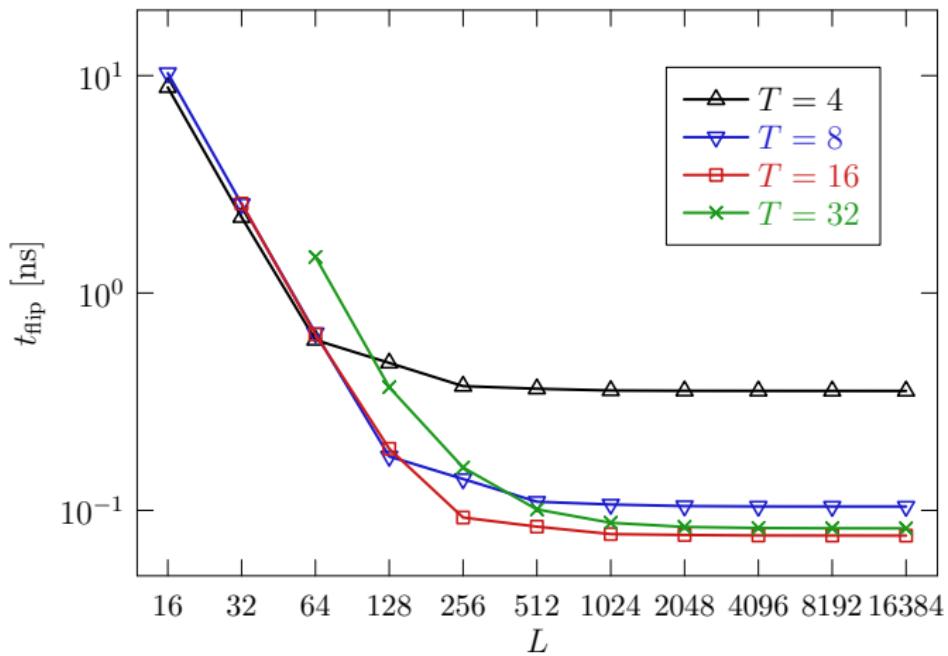
How to assess performance?

- 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 μs , (ns, ps) is well-established unit for spin systems

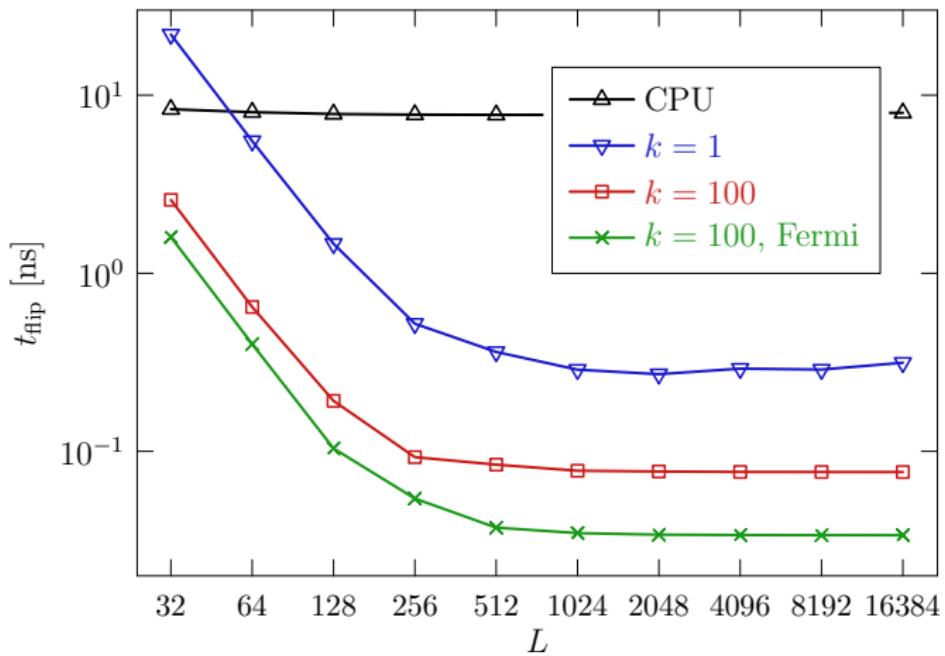
Example: Metropolis simulation of 2D Ising system

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

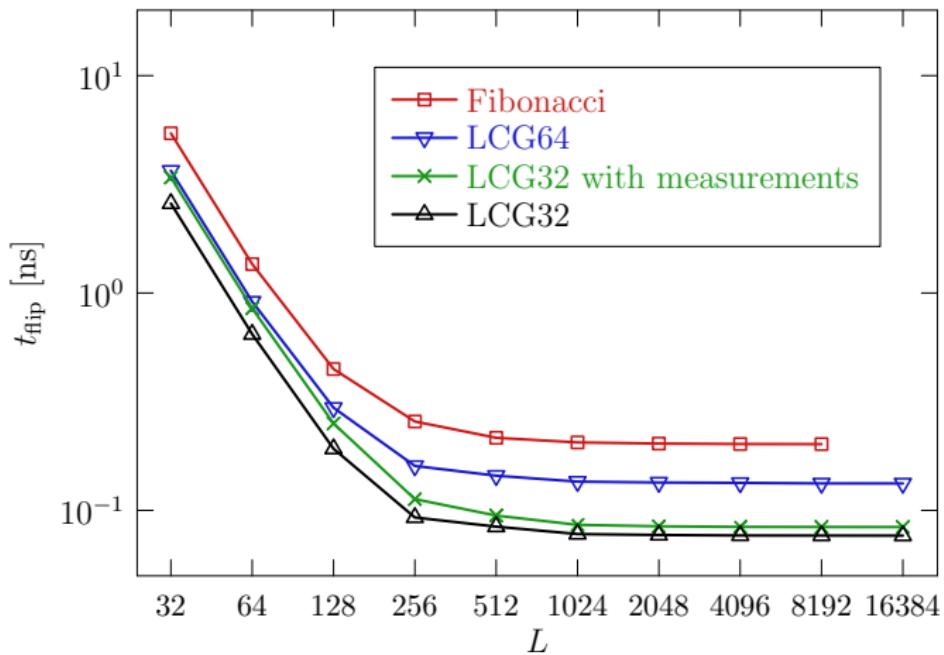
2D Ising ferromagnet



2D Ising ferromagnet



2D Ising ferromagnet

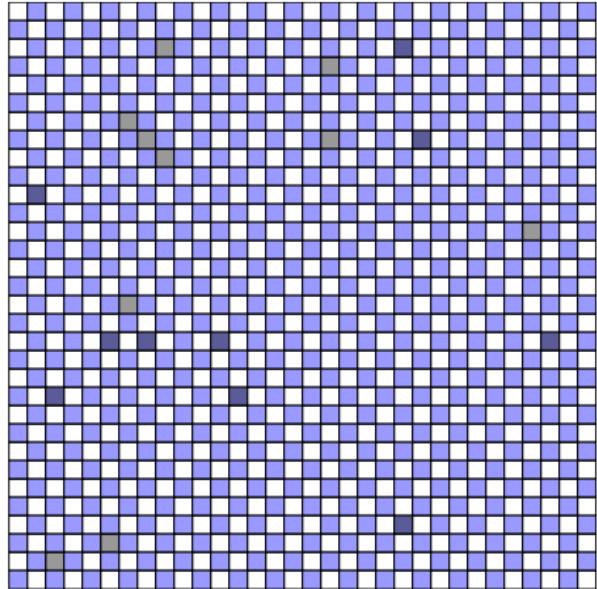


Is it correct?

Detailed balance,

$$\begin{aligned} T(\{s'_i\} \rightarrow \{s_i\}) p_\beta(\{s'_i\}) \\ = T(\{s_i\} \rightarrow \{s'_i\}) p_\beta(\{s_i\}), \end{aligned}$$

only holds for *random* updates.



Is it correct?

Detailed balance,

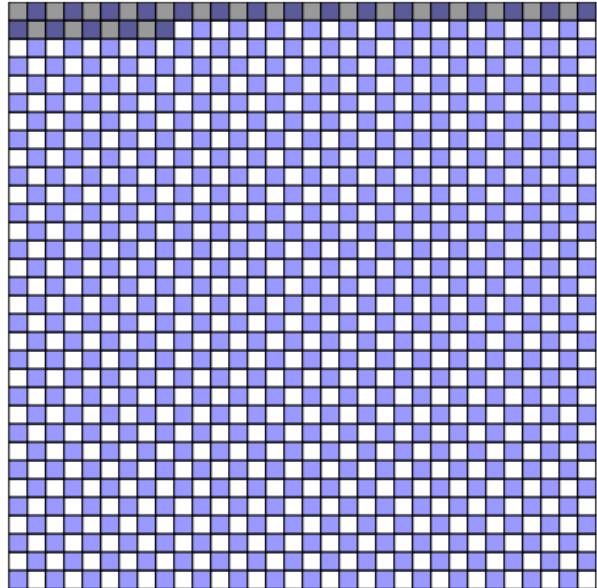
$$\begin{aligned} T(\{s'_i\} \rightarrow \{s_i\}) p_\beta(\{s'_i\}) \\ = T(\{s_i\} \rightarrow \{s'_i\}) p_\beta(\{s_i\}), \end{aligned}$$

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,

$$\begin{aligned} T(\{s'_i\} \rightarrow \{s_i\}) p_\beta(\{s'_i\}) \\ = T(\{s_i\} \rightarrow \{s'_i\}) p_\beta(\{s_i\}), \end{aligned}$$

only holds for *random* updates.

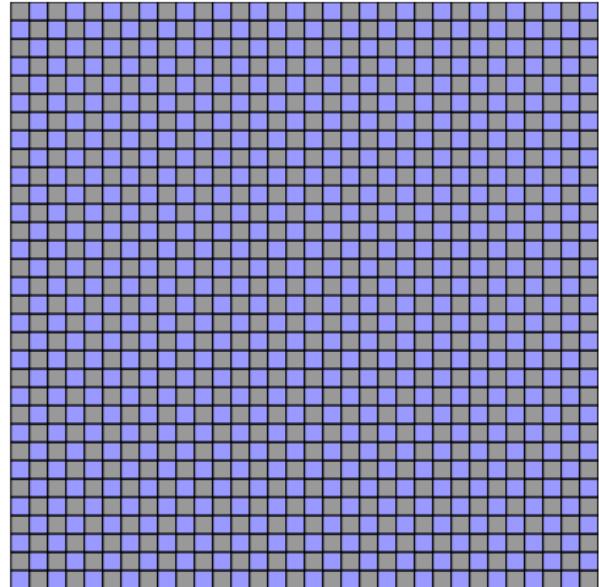
Usually applied sequential update merely satisfies (global) *balance*,

$$\sum_{\{s'_i\}} T(\{s'_i\} \rightarrow \{s_i\}) p_\beta(\{s'_i\}) =$$

$$\sum_{\{s'_i\}} T(\{s_i\} \rightarrow \{s'_i\}) p_\beta(\{s_i\}) = p_\beta(\{s_i\})$$

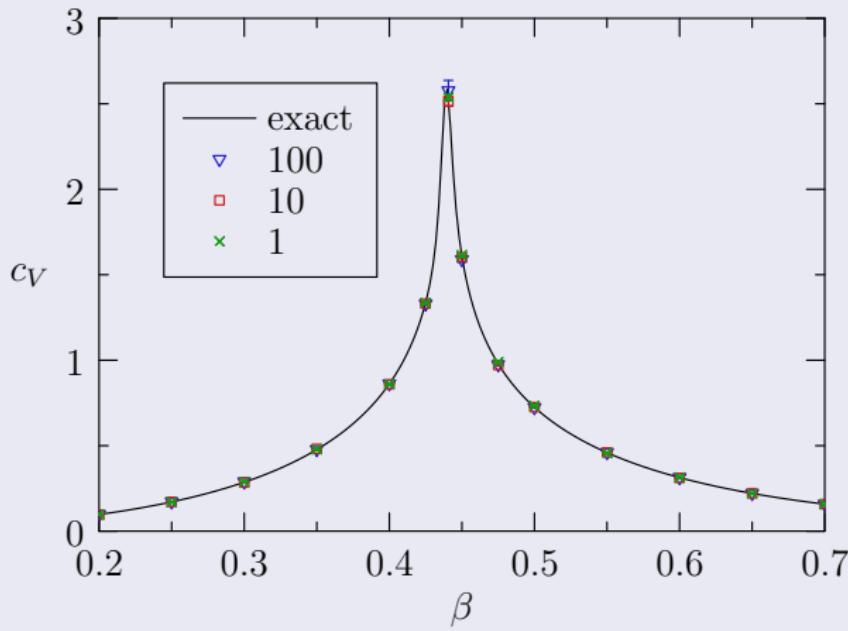
Similarly for checkerboard update. Could restore detailed balance on the level of several sweeps, though:

AAAA(M)AAAABB(M)BBBAA(AA(M)AAAABB(M)BBB ...



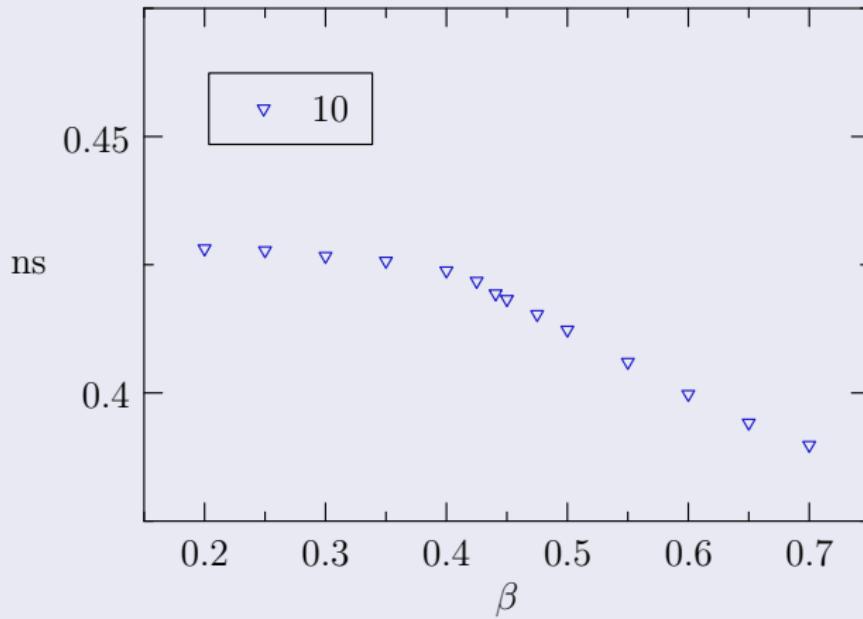
A closer look

Comparison to exact results:



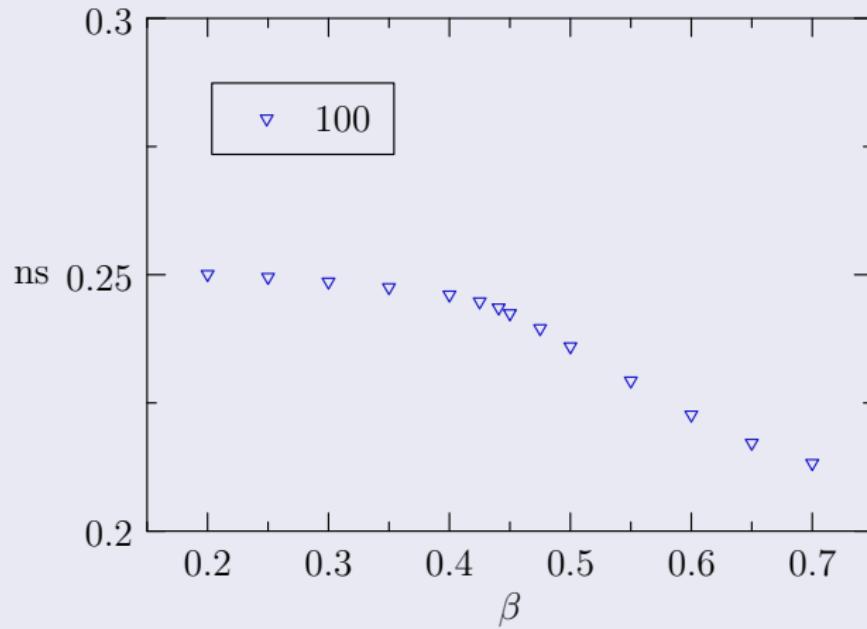
A closer look

Temperature dependent spin flip times:



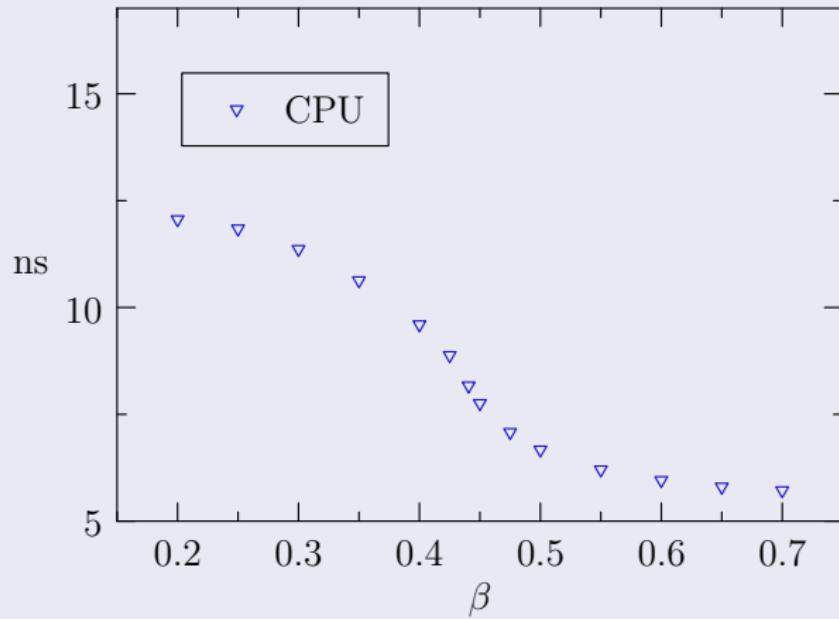
A closer look

Temperature dependent spin flip times:



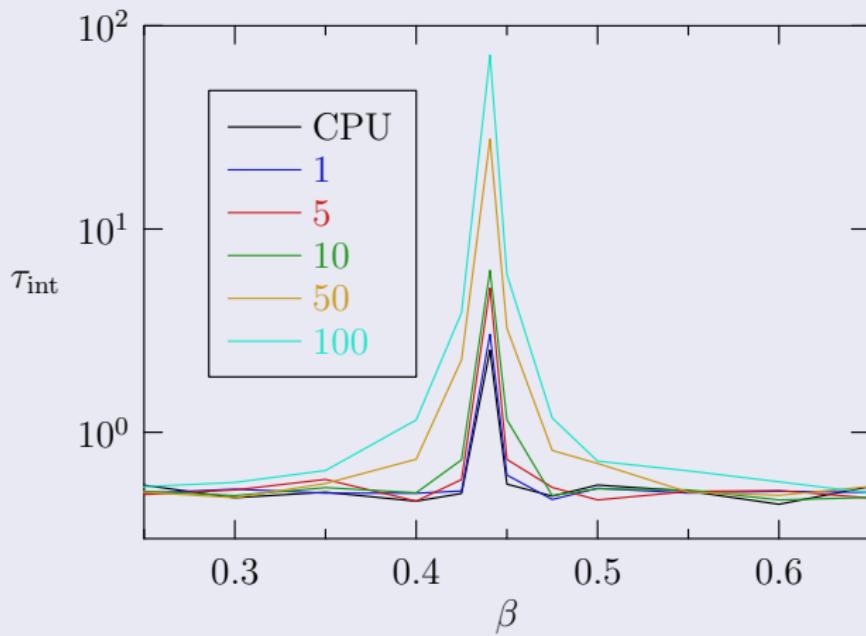
A closer look

Temperature dependent spin flip times:



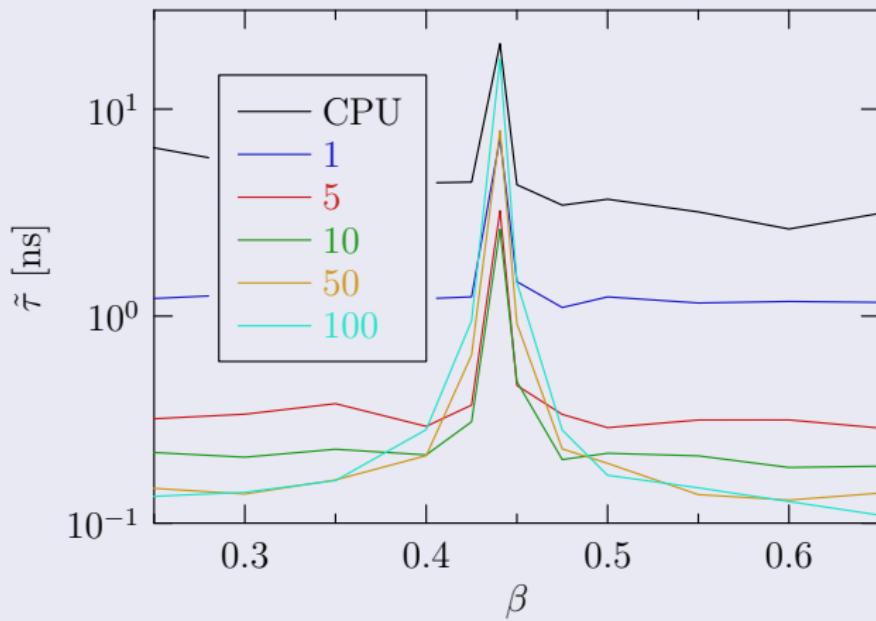
A closer look

Autocorrelation times:



A closer look

Real time to create independent spin configuration:



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?

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 recently) much slower
- for single precision, normal ("high precision") and extra-fast, device-specific versions of sin, cos, exp etc. are provided

Special function units

Fast math

- 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`

Special function units

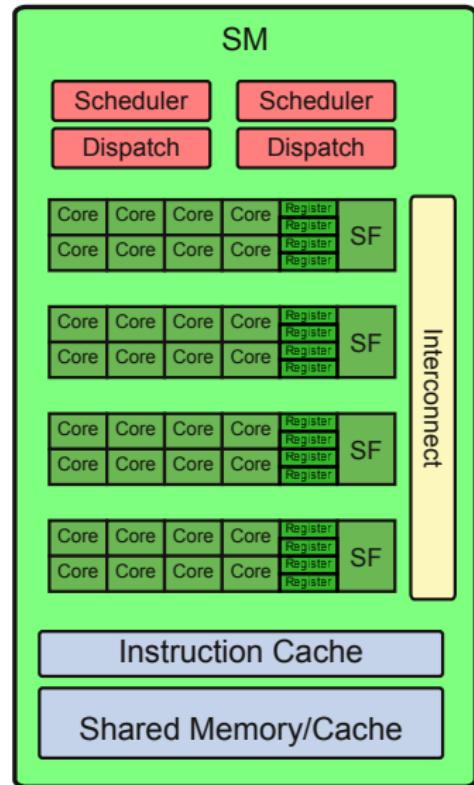
Fast math

- 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)`

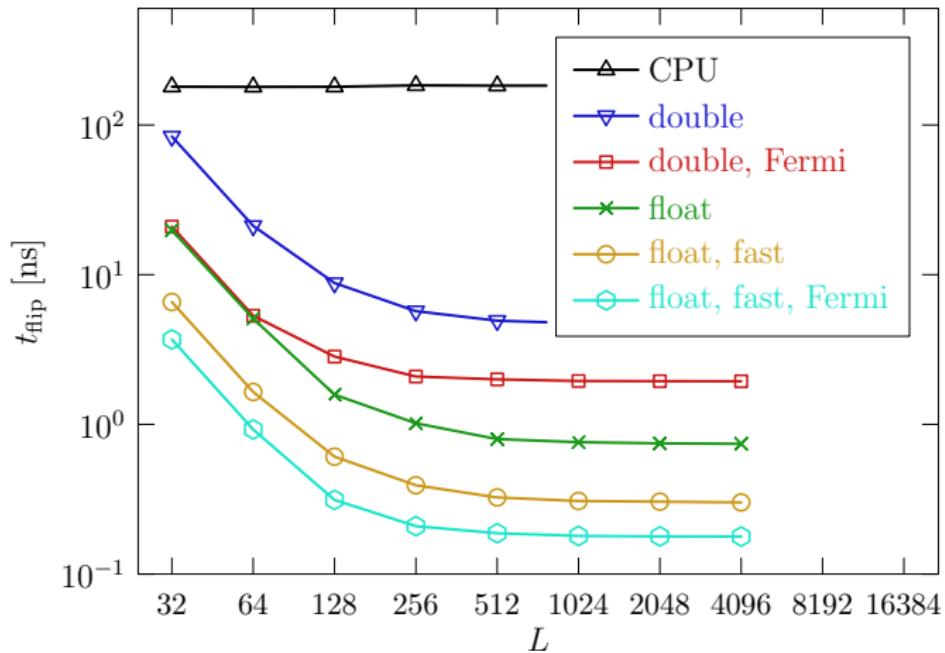
Special function units

Fast math

- 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., **8x** slower for Tesla and gaming cards, **2x** slower for Fermi and up
 - use mixed precision whenever possible



Heisenberg model: performance



Heisenberg model: stability

Performance results:

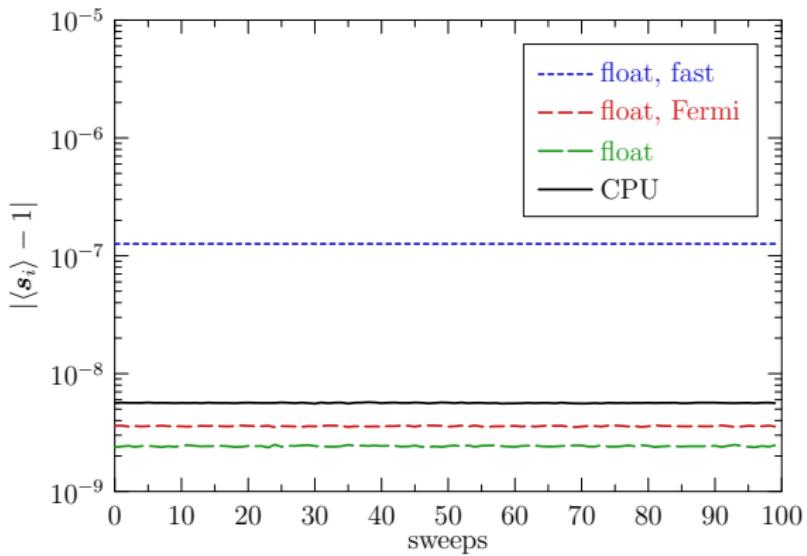
- CPU: 183 ns (single or double) per spin flip
- GPU: 0.74 ns (single), 0.30 ns (fast single) resp. 4.7 ns (double) per spin flip

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.

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.

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.

Next lecture

In lecture 3, we will have a look at the issue of random number generators suitable for massively parallel environments.

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.

Next lecture

In lecture 3, we will have a look at the issue of random number generators suitable for massively parallel environments.

Reading

- M. Weigel, Comput. Phys. Commun. **182**, 1833 (2011) [arXiv:1006.3865].
- M. Weigel, J. Comp. Phys. **231**, 3064 [arXiv:1101.1427]