Simulating spin models on GPU: A tour

Martin Weigel

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

20th Mardi Gras Conference
“Petascale Many Body Methods for Complex Correlated Systems”
LSU, Baton Rouge, February 13, 2015

GPU computing

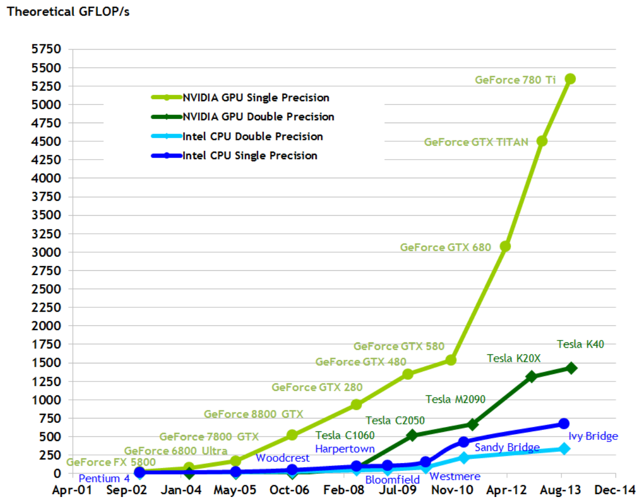
traditional interpretation of GPU computing

GPU computing

```
*DEAD*YourToiletPaper :    yeah, I know
[CoS]Nosferatu :    no ya dont
Chuppie :    ak is like a sniping rifle now
```

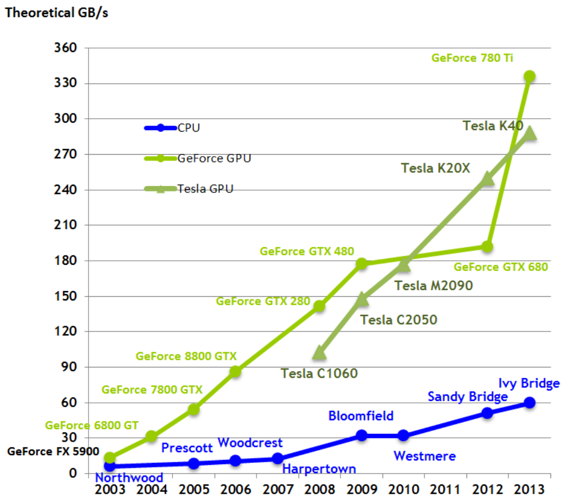
traditional interpretation of GPU computing

GPU computing



- Core i7 IvyBridge i7-3870: 122 GFLOP/s
- NVIDIA Tesla K10: 4580 GFLOP/s (single precision)

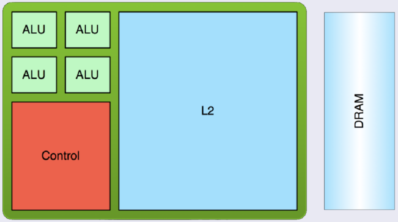
GPU computing



- Core i7 IvyBridge i7-3870: $\approx$ 21 GB/s
- NVIDIA Tesla K10: 320 GB/s

CPU vs. GPU hardware

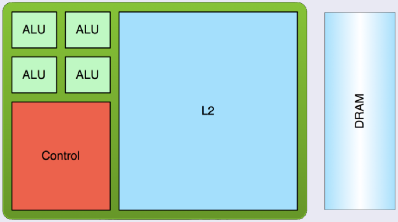
CPU



- optimized for low-latency access to cached data
- extensive logic for branch prediction and out-of-order execution
- do an unpredictable scalar job as fast as possible

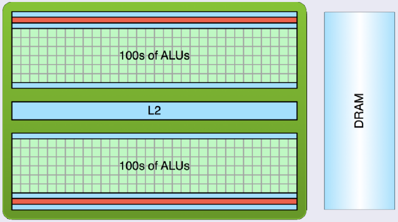
CPU vs. GPU hardware

CPU



- optimized for low-latency access to cached data
- extensive logic for branch prediction and out-of-order execution
- do an unpredictable scalar job as fast as possible

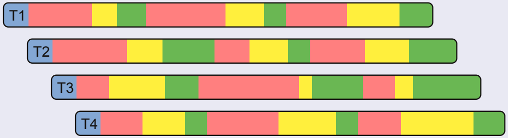
GPU



- optimized for data-parallel throughput computations
- latency hiding
- do as many simple, deterministic jobs in parallel as possible

Latency hiding

GPU threads

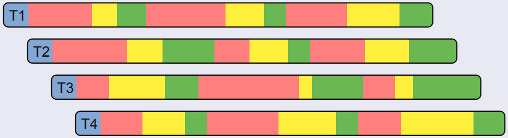


Legend

🟥 waiting for data
🟨 ready to run
🟩 processing

Latency hiding

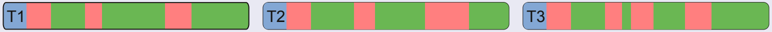
GPU threads



Legend

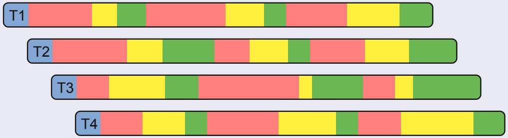
🟥 waiting for data
🟨 ready to run
🟩 processing

CPU threads



Latency hiding

GPU threads



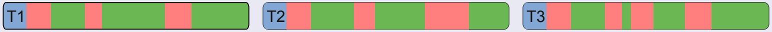
Legend

🟥 waiting for data

🟨 ready to run

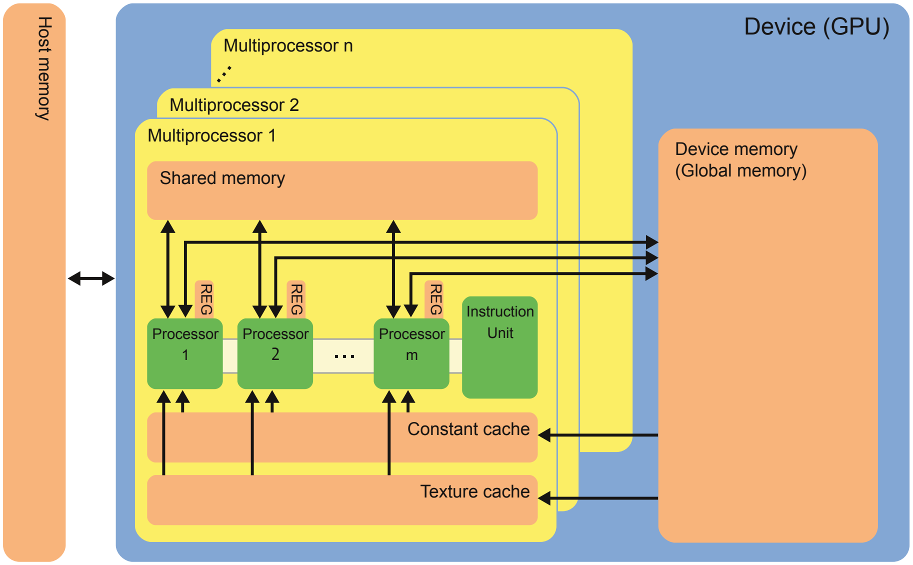
🟩 processing

CPU threads

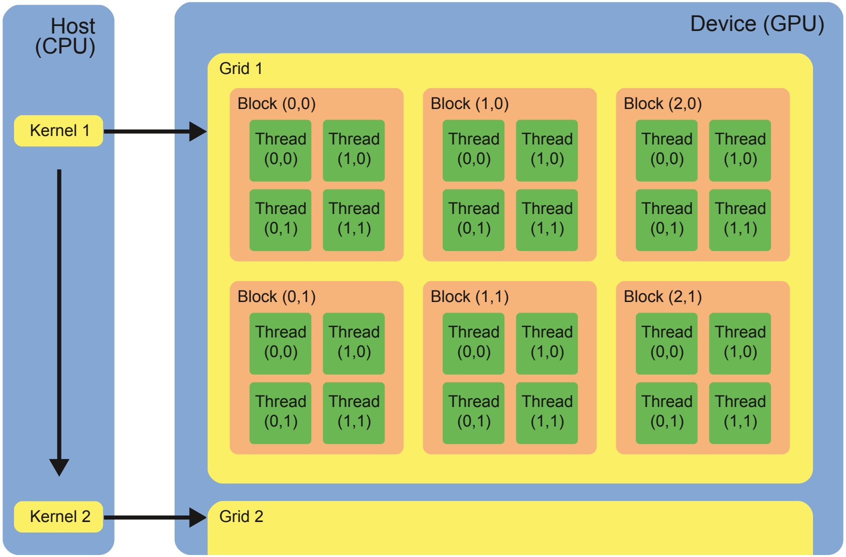


- CPU must minimize latency of individual thread for responsiveness
- GPU *hides* latency through interleaved execution

NVIDIA architecture



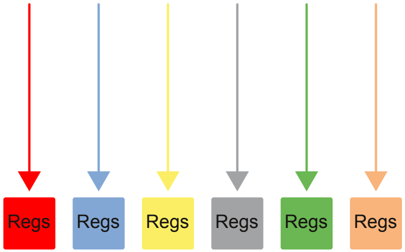
NVIDIA architecture



Memory hierarchy

Per thread

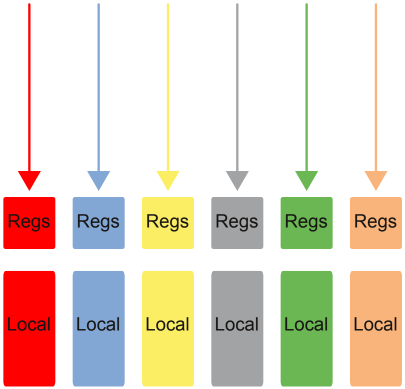
- Registers (extra fast, no copy for ops)



Memory hierarchy

Per thread

- Registers (extra fast, no copy for ops)
- Local memory



Memory hierarchy

Per thread

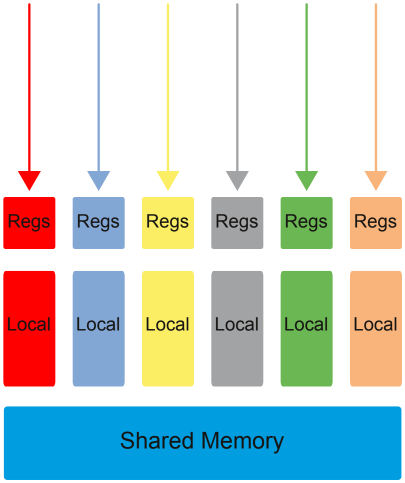
- Registers (extra fast, no copy for ops)
- Local memory

Thread blocks: shared memory

- allocated by thread block, same lifetime as block
- allocate as

```
__shared__ int shared_array[
    DIM];
```

- low latency (of the order of 10 cycles), bandwidth up to 1 TB/s
- use for data sharing and user-managed cache



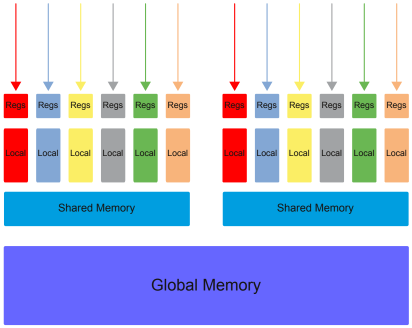
Memory hierarchy

Per device: global memory

- accessible to all threads on device
- lifetime is user-defined

```
cuda_malloc(void **pointer,
     size_t nbytes);
cuda_free(void* pointer);
```

- latency several hundred clock cycles
- bandwidth ≈ 160 GB/s on Fermi (access pattern needs to conform to coalescence rules for good performance)



Memory hierarchy

Per device: global memory

- accessible to all threads on device
- lifetime is user-defined

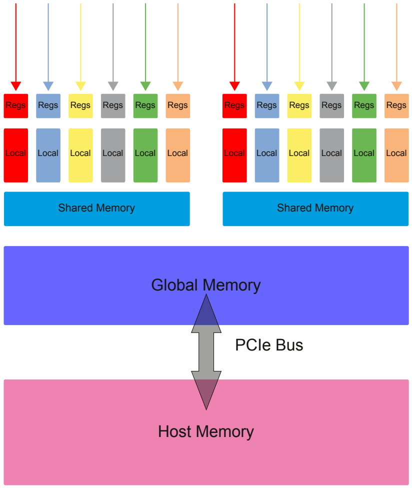
```
cuda_malloc(void **pointer,
     size_t nbytes);
cuda_free(void* pointer);
```

- latency several hundred clock cycles
- bandwidth ≈ 160 GB/s on Fermi (access pattern needs to conform to coalescence rules for good performance)

Per host: device memory

- no direct access from CUDA threads
- copy data to/from device with

```
cudaMemcpy(void* dest, void*
     src, size_t nbytes,
     cudaMemcpyHostToDevice);
```



Spin models

Consider **classical** spin models with nn interactions, in particular

Ising model

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

Heisenberg model

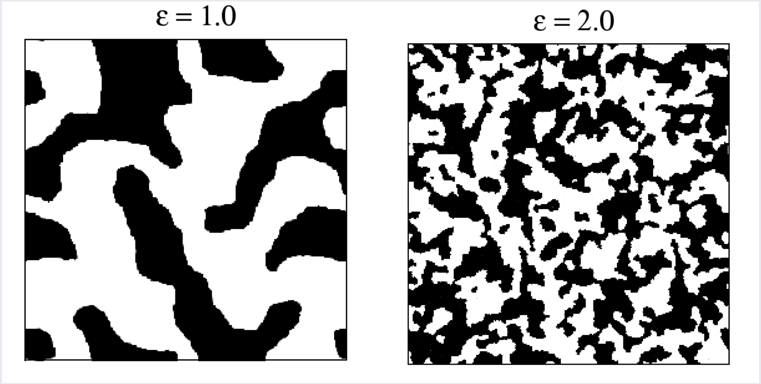
$$\boxed{\mathcal{H} = -J \sum_{\langle ij \rangle} \vec{S}_i \cdot \vec{S}_j + \vec{H} \cdot \sum_i \vec{S}_i, \quad |\vec{S}_i| = 1}$$

Edwards-Anderson spin glass

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

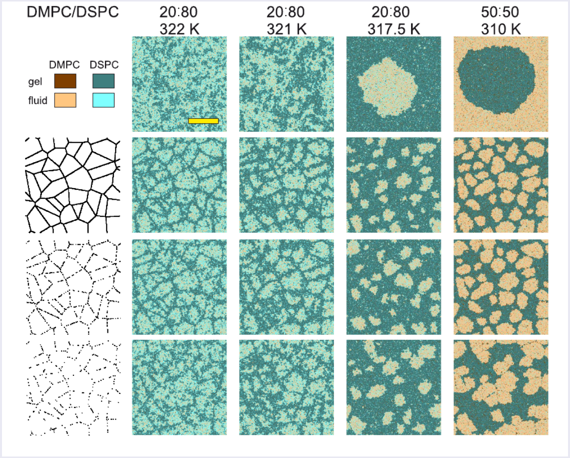
Spin models — Applications

Nucleation and phase ordering



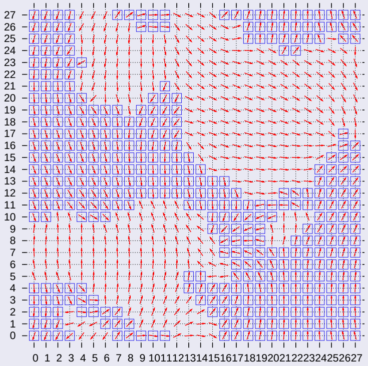
Spin models — Applications

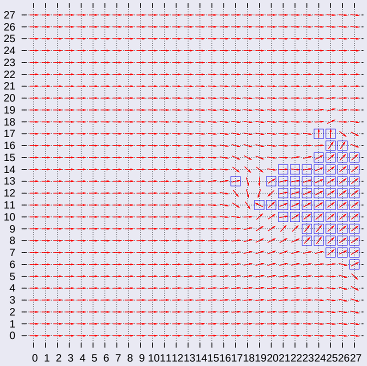
Phase separation in lipid membranes



Spin models — Applications

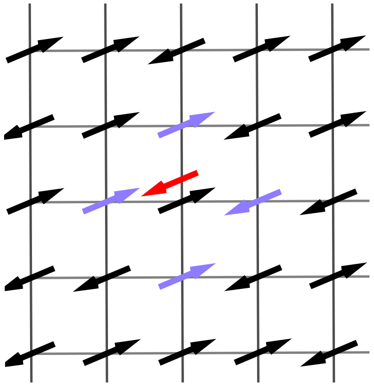
Quenched disorder in magnetic systems





Metropolis algorithm

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

$$r \leq \min[1, e^{-\beta \Delta E}]$$

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

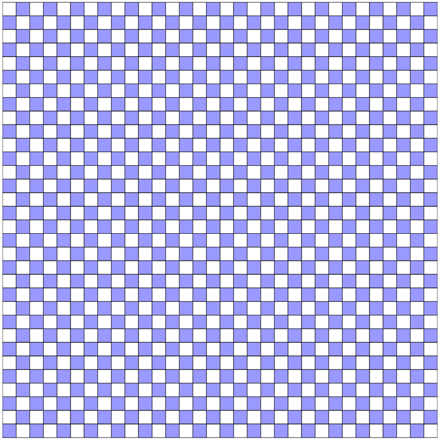
We need to perform updates on non- (or weakly) interacting sub-domains.

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.

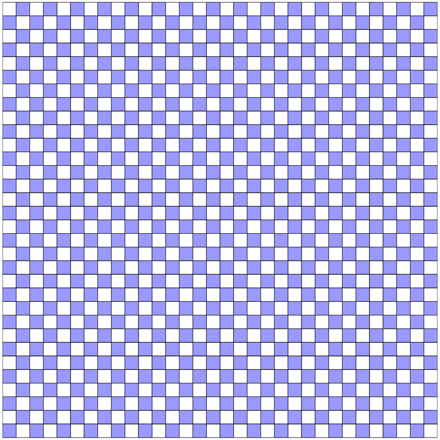
Checkerboard decomposition

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

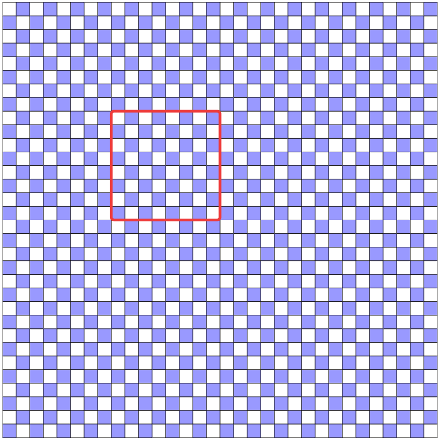
A straightforward minimal code translates the CPU version, using one thread to update each spin of a sub-lattice.

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
- $\sim$ 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

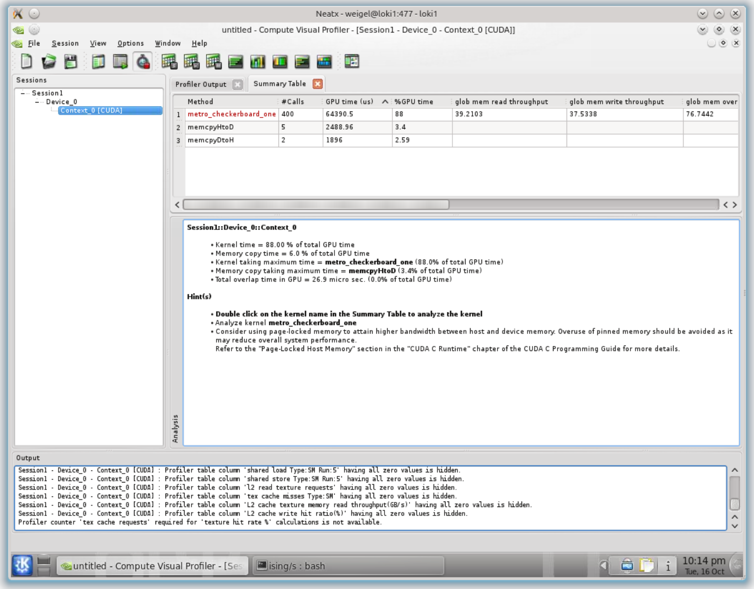
Compare the serial CPU code and the first GPU version:

- CPU code at 7.66 ns per spin flip on Intel Q9850
- GPU code at 0.84 ns per spin flip on Tesla C1060
- $\sim$ 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
- and/or read:
    - CUDA C Programming Guide
    - CUDA C Best Practices Guide

Compute visual profiler


Neatx - weigel@loki1:477 - loki1
untitled - Compute Visual Profiler - [Session1 - Device_0 - Context_0 [CUDA]]
File
Session
View
Options
Window
Help
Sessions
Session1
Device_0
Context_0 [CUDA]
Profiler Output
Summary Table
Analysis
Output
untitled - Compute Visual Profiler - [Ses
ising/s : bash
10:14 pm
Tue, 16 Oct

	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

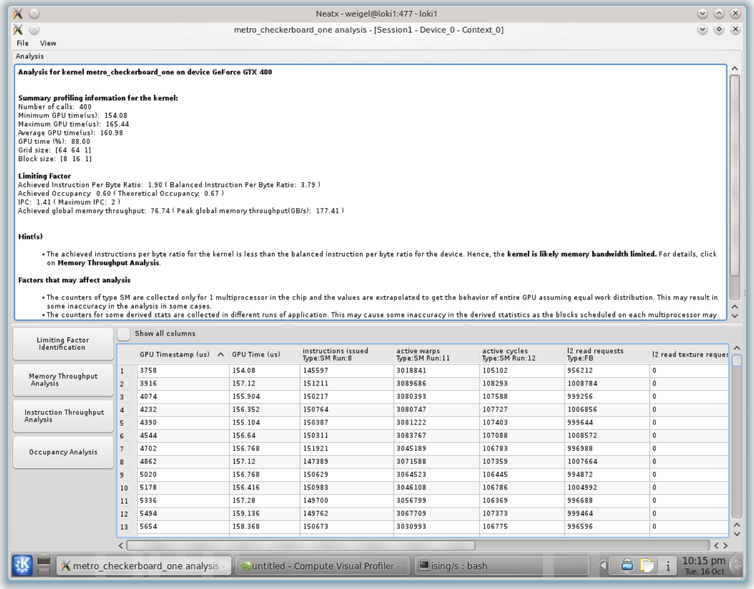
- Kernel time = 88.00 % of total GPU time
- Memory copy time = 6.0 % of total GPU time
- Kernel taking maximum time = **metro_checkerboard_one** (88.0% of total GPU time)
- Memory copy taking maximum time = **memcpyHtoD** (3.4% of total GPU time)
- Total overlap time in GPU = 26.9 micro sec. (0.0% of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

```
Session1 - Device_0 - Context_0 [CUDA] : Profiler table column 'shared load Type:SM Run:5' having all zero values is hidden.
Session1 - Device_0 - Context_0 [CUDA] : Profiler table column 'shared store Type:SM Run:5' having all zero values is hidden.
Session1 - Device_0 - Context_0 [CUDA] : Profiler table column 'l2 read texture requests' having all zero values is hidden.
Session1 - Device_0 - Context_0 [CUDA] : Profiler table column 'tex cache misses Type:SM' 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.
```

Compute visual profiler


Neatx - weigel@loki1:477 - loki1
metro_checkerboard_one analysis - [Session1 - Device_0 - Context_0]
File
View
Analysis

Analysis for kernel metro_checkerboard_one on device GeForce GTX 480

Summary profiling information for the kernel:
Number of calls: 400
Minimum GPU time(us): 154.08
Maximum GPU time(us): 165.44
Average GPU time(us): 160.98
GPU time (%): 88.00
Grid size: [64 64 1]
Block size: [8 16 1]

Limiting Factor
Achieved Instruction Per Byte Ratio: 1.90 (Balanced Instruction Per Byte Ratio: 3.79)
Achieved Occupancy: 0.60 (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

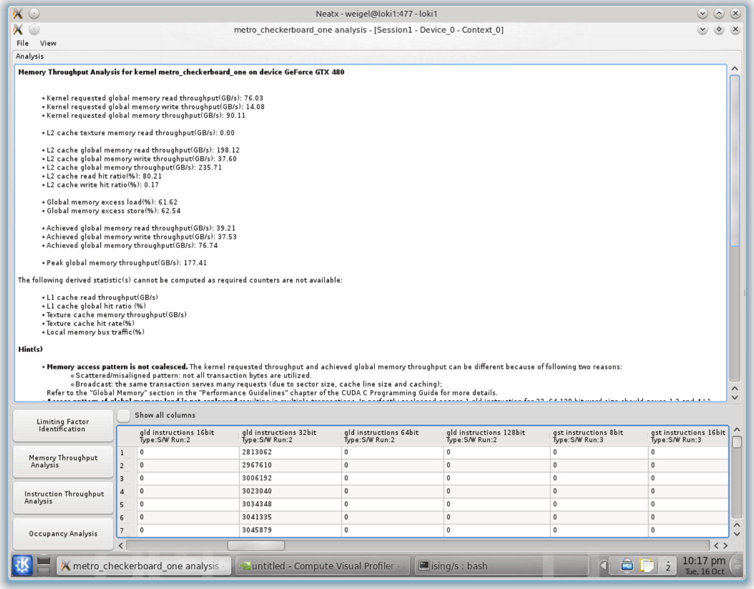
- The counters of type SM are collected only for 1 multiprocessor in the chip and the values are extrapolated to get the behavior of entire GPU assuming equal work distribution. This may result in some inaccuracy in the analysis in some cases.
- The counters for some derived stats are collected in different runs of application. This may cause some inaccuracy in the derived statistics as the blocks scheduled on each multiprocessor may

Limiting Factor Identification
Memory Throughput Analysis
Instruction Throughput Analysis
Occupancy Analysis
Show all columns

	GPU Timestamp (us)	GPU Time (us)	instructions issued Type:SM Run:8	active warps Type:SM Run:11	active cycles Type:SM Run:12	l2 read requests Type:FB	l2 read texture reques
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	999256	0
4	4232	156.352	150764	3080747	107727	1006856	0
5	4390	155.104	150387	3081222	107403	999644	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	999464	0
13	5654	158.368	150673	3030993	106775	996596	0

metro_checkerboard_one analysis -
untitled - Compute Visual Profiler -
ising/s : bash
10:15 pm
Tue, 16 Oct

Compute visual profiler


Neatx - weigel@loki1:477 - loki1
metro_checkerboard_one analysis - [Session1 - Device_0 - Context_0]
File
View
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 store(%): 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(%)

Hint(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
Memory Throughput Analysis
Instruction Throughput Analysis
Occupancy Analysis
Show all columns

	gld instructions 16bit Type:S/W Run:2	gld instructions 32bit Type:S/W Run:2	gld instructions 64bit Type:S/W Run:2	gld instructions 128bit Type:S/W Run:2	gst instructions 8bit Type:S/W Run:3	gst instructions 16bit Type:S/W Run:3
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

metro_checkerboard_one analysis
untitled - Compute Visual Profiler
ising/s : bash
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.* "

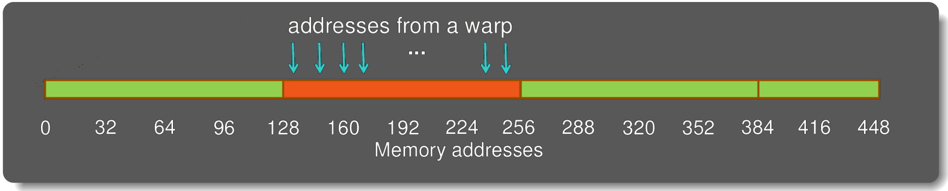
In Fermi and Kepler:

- 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

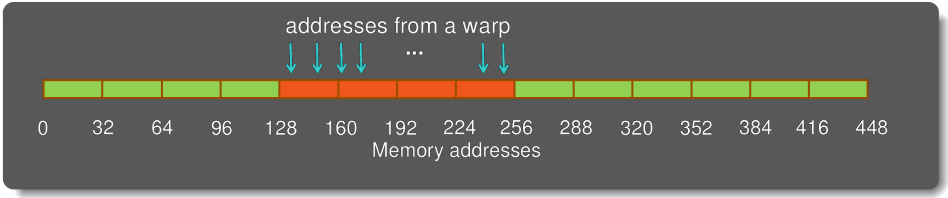
- cached load
- warp requests aligned to 32 bytes
- accessing consecutive 4-byte words
- addresses lie in one cache line
- efficiency:
  - warp needs 128 bytes
  - 128 bytes are transferred on a cache miss
  - bus utilization 100%



Memory coalescence

Access pattern

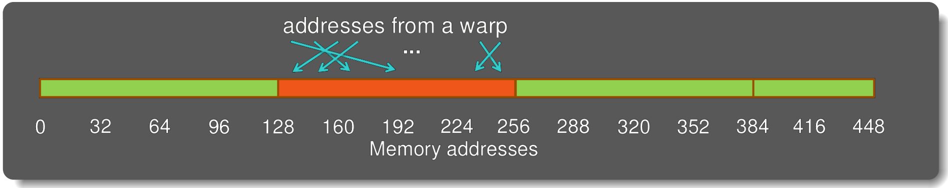
- L2 only load
- warp requests aligned to 32 bytes
- accessing consecutive 4-byte words
- addresses lie in 4 adjacent segments
- efficiency:
  - warp needs 128 bytes
  - 128 bytes are transferred on a cache miss
  - bus utilization 100%



Memory coalescence

Access pattern

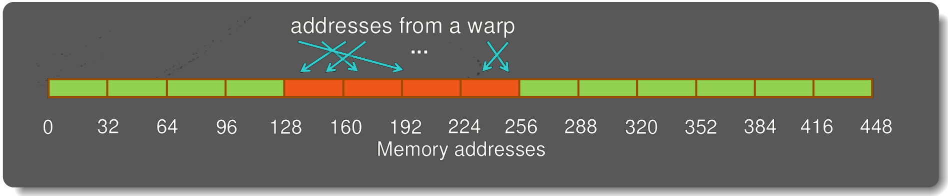
- cached load
- warp requests aligned to 32 bytes
- accessing permuted 4-byte words
- addresses lie in one cache line
- efficiency:
  - warp needs 128 bytes
  - 128 bytes are transferred on a cache miss
  - bus utilization 100%



Memory coalescence

Access pattern

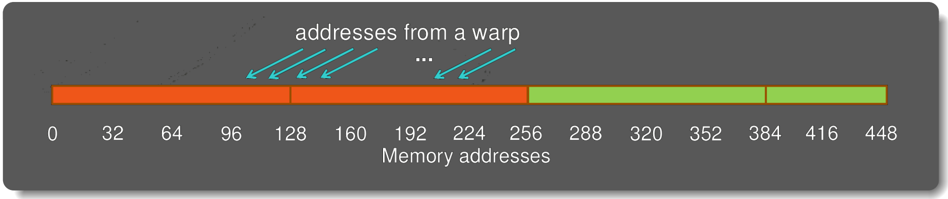
- L2 only load
- warp requests aligned to 32 bytes
- accessing permuted 4-byte words
- addresses lie in 4 adjacent segments
- efficiency:
  - warp needs 128 bytes
  - 128 bytes are transferred on a cache miss
  - bus utilization 100%



Memory coalescence

Access pattern

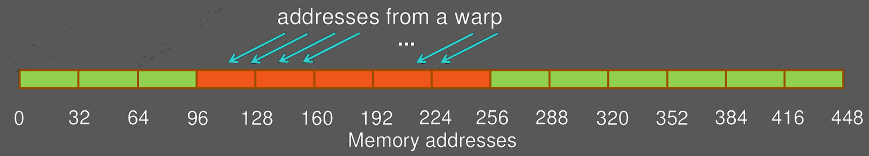
- cached load
- warp requests misaligned to 32 bytes
- accessing consecutive 4-byte words
- addresses fall in two adjacent cache lines
- efficiency:
  - warp needs 128 bytes
  - 256 bytes are transferred on a cache miss
  - bus utilization 50%



Memory coalescence

Access pattern

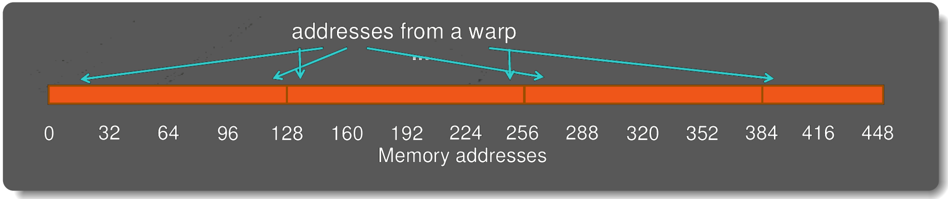
- L2 only load
- warp requests misaligned to 32 bytes
- accessing consecutive 4-byte words
- addresses fall in at most 5 segments
- efficiency:
  - warp needs 128 bytes
  - 160 bytes are transferred on cache misses
  - bus utilization at least 80% (100% for some patterns)



Memory coalescence

Access pattern

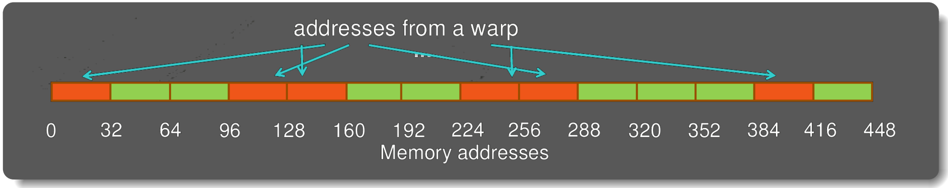
- cached load
- warp requests are 32 scattered 4-byte words
- addresses fall in N cache lines
- efficiency:
  - warp needs 128 bytes
  - $N \times 128$ bytes are transferred on cache misses
  - bus utilization $1/N$



Memory coalescence

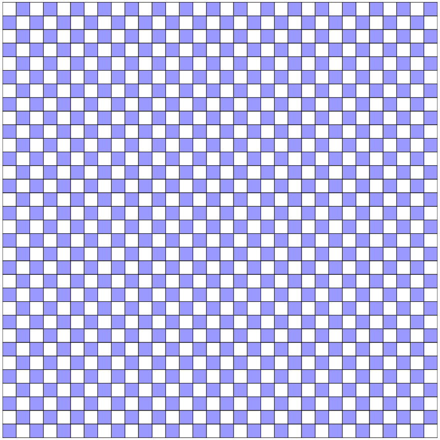
Access pattern

- 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

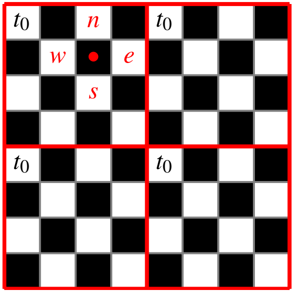
We need to perform updates on non- (or weakly) interacting sub-domains. For bi-partite lattices and nearest-neighbor interactions, this leads to a checkerboard decomposition.

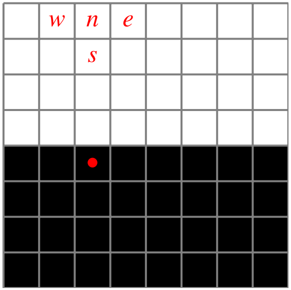


Generalizations for more general lattices and longer (but finite) range interactions are straightforward.

Coalescence for checkerboard accesses

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

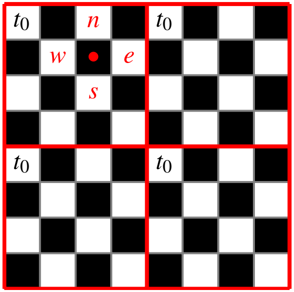


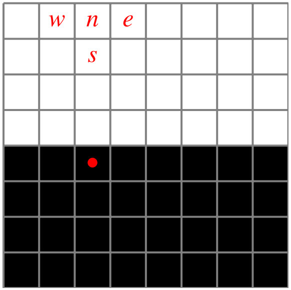


(E. E. Ferrero *et. al.*, Comput. Phys. Commun. 183, 1578 (2012))

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

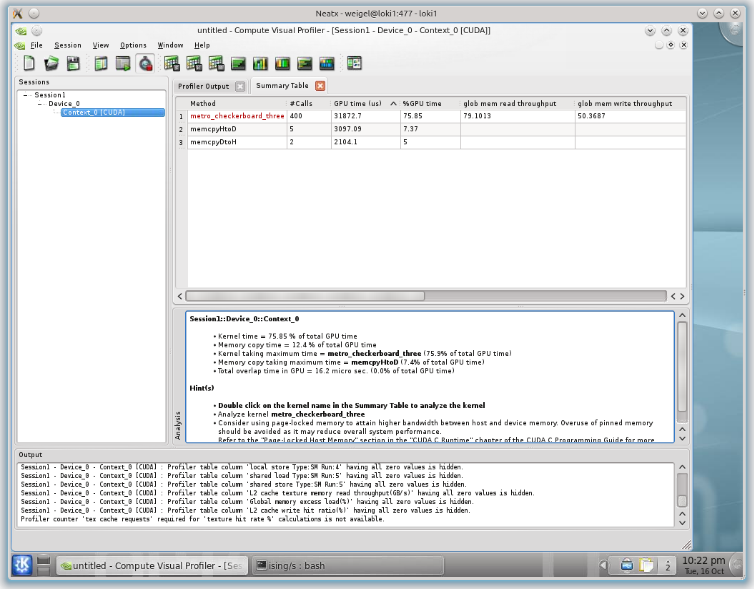
Arrange spins in the crinkled fashion in memory, leading to coalesced accesses to global memory:

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


Neatx - weigel@loki1:477 - loki1
untitled - Compute Visual Profiler - [Session1 - Device_0 - Context_0 [CUDA]]
File Session View Options Window Help
Sessions
Session1
Device_0
Context_0 [CUDA]
Profiler Output
Summary Table
Analysis
Output
untitled - Compute Visual Profiler - [Ses
ising/s : bash
10:22 pm
Tue, 16 Oct

	Method	#Calls	GPU time (us)	%GPU time	glob mem read throughput	glob mem write throughput
1	metro_checkerboard_three	400	31872.7	75.85	79.1013	50.3687
2	memcpyHtoD	5	3097.09	7.37		
3	memcpyDtoH	2	2104.1	5		

Session1::Device_0::Context_0

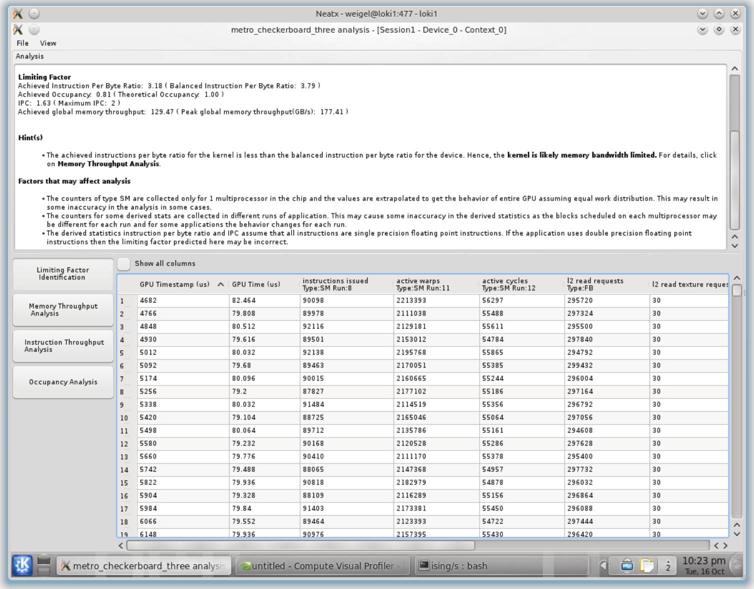
- Kernel time = 75.85 % of total GPU time
- Memory copy time = 12.4 % of total GPU time
- Kernel taking maximum time = **metro_checkerboard_three** (75.9% of total GPU time)
- Memory copy taking maximum time = **memcpyHtoD** (7.4% of total GPU time)
- Total overlap time in GPU = 16.2 micro sec. (0.0% of total GPU time)

Hint(s)

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

```
Session1 - Device_0 - Context_0 [CUDA] : Profiler table column 'local store Type:SM Run:4' having all zero values is hidden.
Session1 - Device_0 - Context_0 [CUDA] : Profiler table column 'shared load Type:SM Run:5' having all zero values is hidden.
Session1 - Device_0 - Context_0 [CUDA] : Profiler table column 'shared store Type:SM Run:5' 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 'Global memory excess load(%)' 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.
```

GPU simulation: second version


Neatx - weigel@loki1:477 - loki1
metro_checkerboard_three analysis - [Session1 - Device_0 - Context_0]
File
View
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)
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

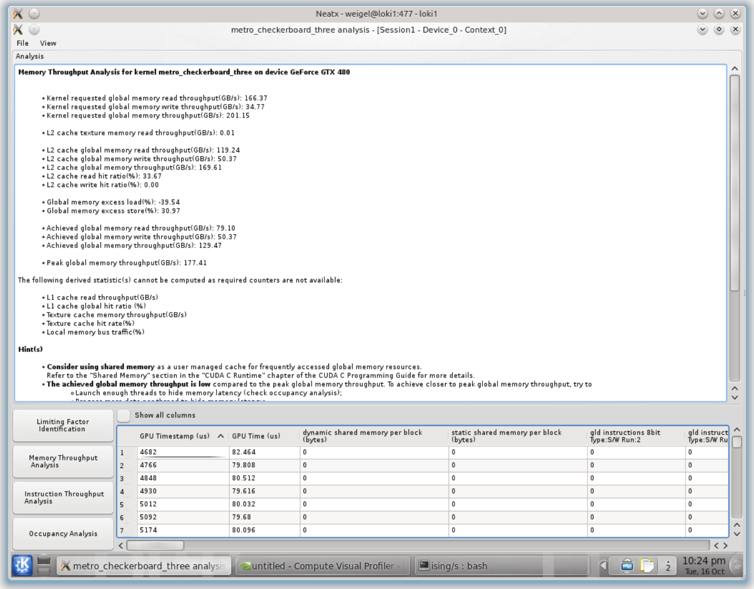
- The counters of type SM are collected only for 1 multiprocessor in the chip and the values are extrapolated to get the behavior of entire GPU assuming equal work distribution. This may result in some inaccuracy in the analysis in some cases.
- The counters for some derived stats are collected in different 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.

Limiting Factor Identification
Memory Throughput Analysis
Instruction Throughput Analysis
Occupancy Analysis
Show all columns

	GPU Timestamp (us)	GPU Time (us)	instructions issued Type:SM Run:8	active warps Type:SM Run:11	active cycles Type:SM Run:12	l2 read requests Type:FB	l2 read texture reques
1	4682	82.464	90098	2213393	56297	295720	30
2	4766	79.808	89978	2111038	55488	297324	30
3	4848	80.512	92116	2129181	55611	295500	30
4	4930	79.616	89501	2153012	54784	297840	30
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	2114519	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	2111170	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

metro_checkerboard_three analysis
untitled - Compute Visual Profiler -
ising/s : bash
2
10:23 pm
Tue, 16 Oct

GPU simulation: second version


Neatx - weigel@loki1:477 - loki1
metro_checkerboard_three analysis - [Session1 - Device_0 - Context_0]
File
View
Analysis
Limiting Factor Identification
Memory Throughput Analysis
Instruction Throughput Analysis
Occupancy Analysis
Show all columns
metro_checkerboard_three analysis
untitled - Compute Visual Profiler -
ising/s : bash
10:24 pm
Tue, 16 Oct

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 store(%): 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(%)

Hint(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);

	GPU Timestamp (us)	GPU Time (us)	dynamic shared memory per block (bytes)	static shared memory per block (bytes)	gld instructions 8bit Type:S/W Run:2	gld instruct Type:S/W Ru
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

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)

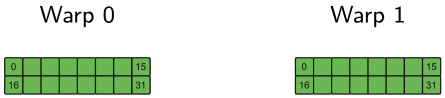
Thread divergence

- 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

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

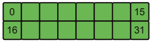


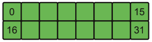
Thread divergence

No serialization

Warp 0 Warp 1

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





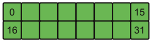
Thread divergence

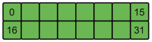
No serialization

Warp 0

Warp 1

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



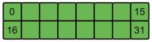


Thread divergence

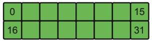
With serialization

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

    do something

} else {

    do something else

}

...
```

Warp 0



Warp 1



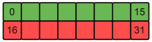
Thread divergence

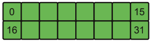
With serialization

Warp 0

Warp 1

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

    do something

} else {

    do something else

}

...
```





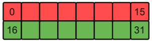
Thread divergence

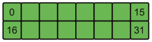
With serialization

Warp 0

Warp 1

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

    do something

} else {

    do something else

}

...
```





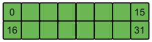
Thread divergence

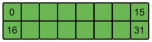
With serialization

Warp 0

Warp 1

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

    do something

} else {

    do something else

}

...
```





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.

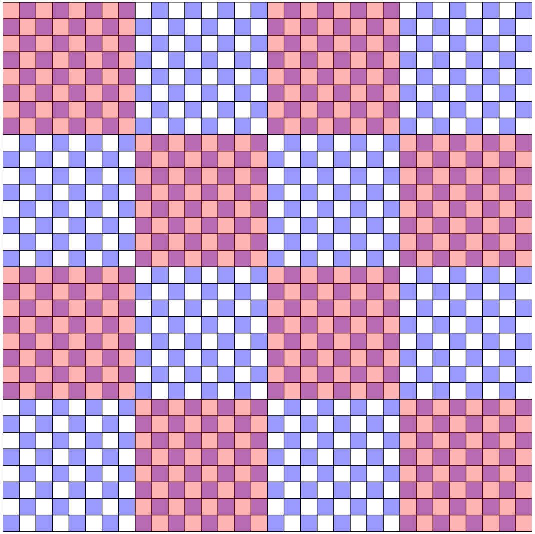
Global memory version: performance

- 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!

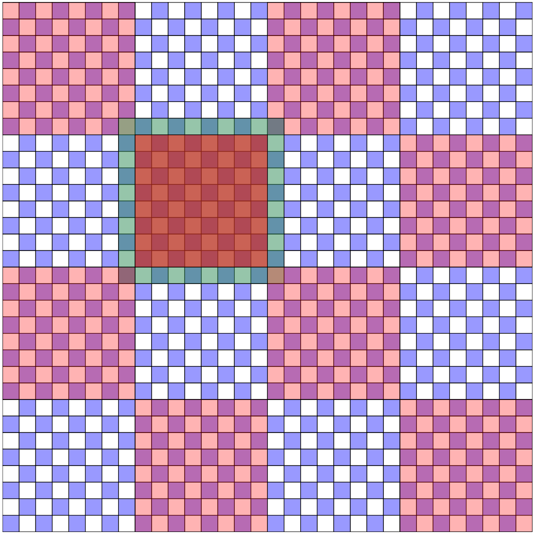
Double checkerboard decomposition

- (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];
    }
  }
```

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

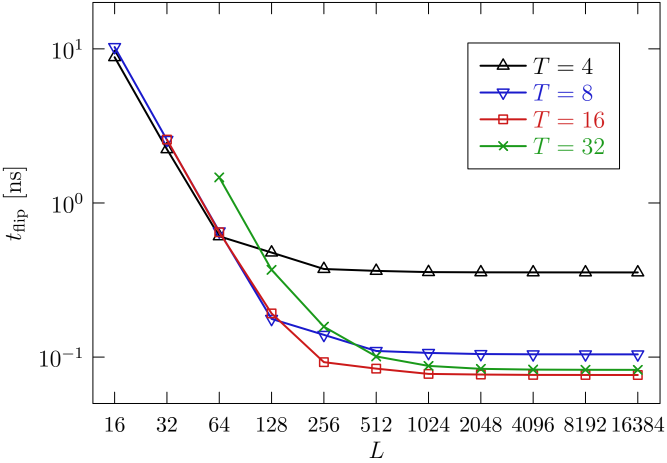
How to assess performance?

- 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

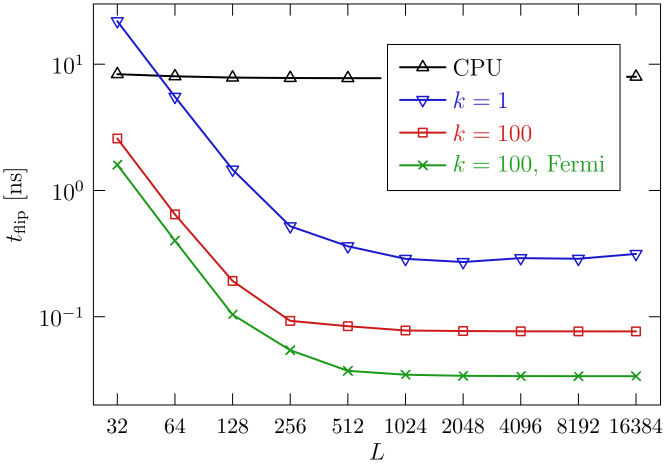
Example: Metropolis simulation of 2D Ising system

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

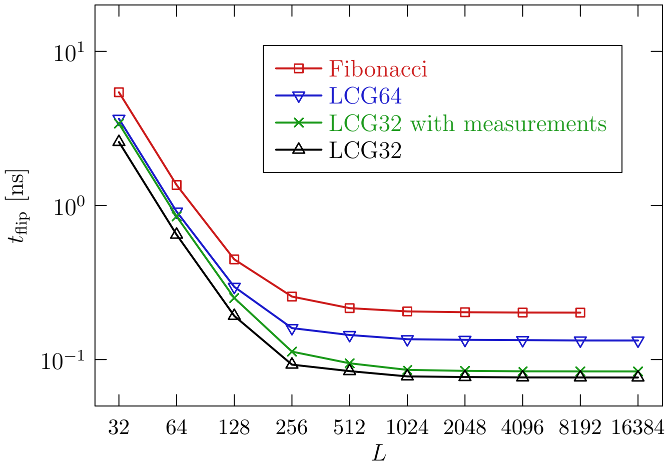
2D Ising ferromagnet



2D Ising ferromagnet

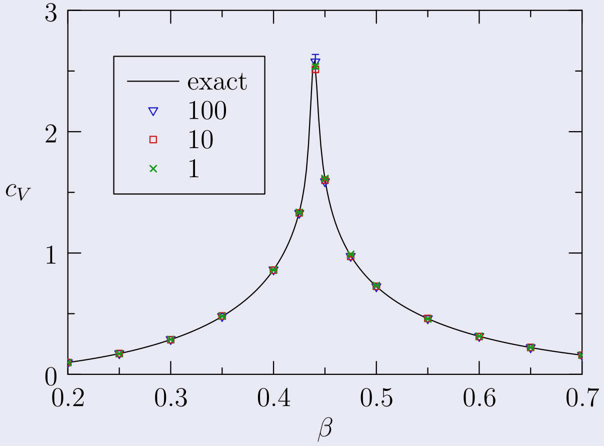


2D Ising ferromagnet



A closer look

Comparison to exact results:



A closer look

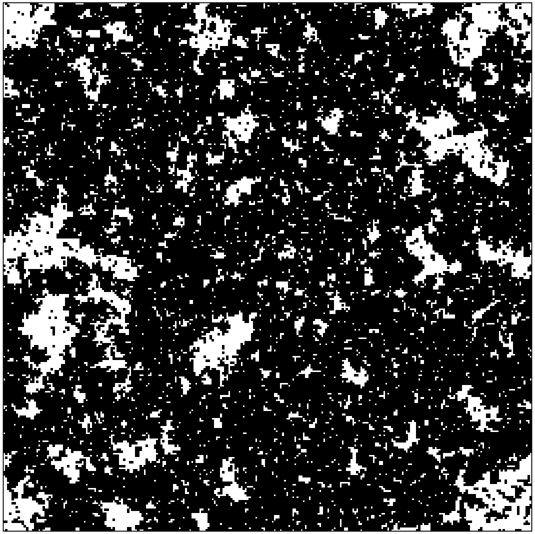
Random number generators: significant deviations from exact result for test case of 1024×1024 system at $\beta = 0.4$, 10^7 sweeps

method	e	Δ_{rel}	C_V	Δ_{rel}
exact	1.106079207	0	0.8616983594	0
		sequential update (CPU)		
LCG32	$1.1060788(15)$	-0.26	$0.83286(45)$	-63.45
LCG64	$1.1060801(17)$	0.49	$0.86102(60)$	-1.14
Fibonacci, $r = 512$	$1.1060789(17)$	-0.18	$0.86132(59)$	-0.64
		checkerboard update (GPU)		
LCG32	$1.0944121(14)$	-8259.05	$0.80316(48)$	-121.05
LCG32, random	$1.1060775(18)$	-0.97	$0.86175(56)$	0.09
LCG64	$1.1061058(19)$	13.72	$0.86179(67)$	0.14
LCG64, random	$1.1060803(18)$	0.62	$0.86215(63)$	0.71
Fibonacci, $r = 512$	$1.1060890(15)$	6.43	$0.86099(66)$	-1.09
Fibonacci, $r = 1279$	$1.1060800(19)$	0.40	$0.86084(53)$	-1.64

Details on random number generators for GPUs:

`M. Manssen, MW, and A. K. Hartmann, Eur. Phys. J. Spec. Topics` **210**`, 53 (2012)`

Critical configuration

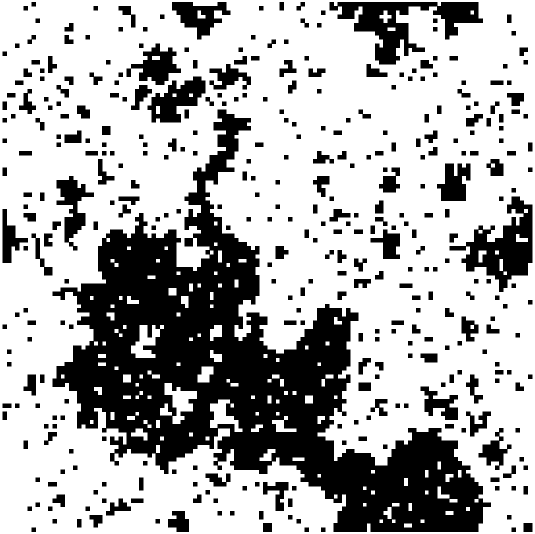


Cluster algorithms

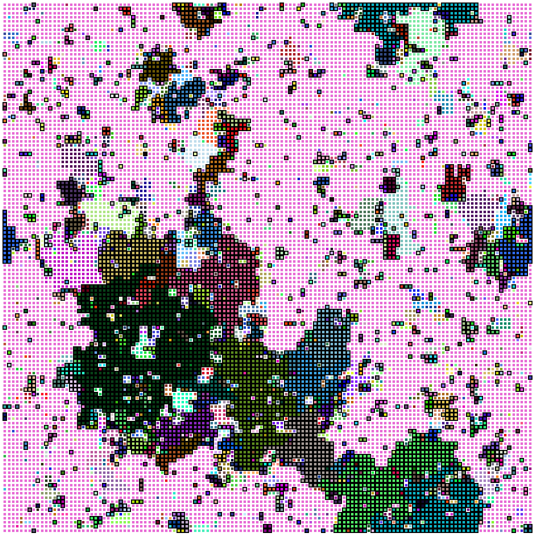
Would need to use cluster algorithms for efficient equilibrium simulation of spin models at criticality:

1. Activate bonds between like spins with probability $p = 1 - e^{-2\beta J}$.
2. Construct (Swendsen-Wang) spin clusters from domains connected by active bonds.
3. Flip independent clusters with probability $1/2$.
4. Goto 1.

Swendsen-Wang update



Swendsen-Wang update



Cluster algorithms

Would need to use cluster algorithms for efficient equilibrium simulation of spin models at criticality:

1. Activate bonds between like spins with probability $p = 1 - e^{-2\beta J}$.
2. Construct (Swendsen-Wang) spin clusters from domains connected by active bonds.
3. Flip independent clusters with probability $1/2$.
4. Goto 1.

Steps 1 and 3 are local $\Rightarrow$ Can be efficiently ported to GPU.
What about step 2? $\Rightarrow$ Domain decomposition into tiles.

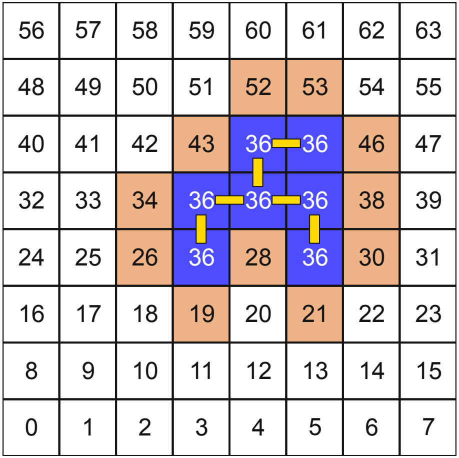
labeling *inside* of domains

- Hoshen-Kopelman
- breadth-first search
- self-labeling
- union-find algorithms

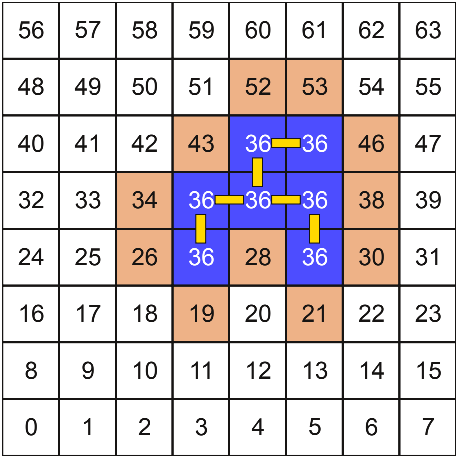
relabeling *across* domains

- self-labeling
- hierarchical approach
- iterative relaxation

BFS or Ants in the Labyrinth

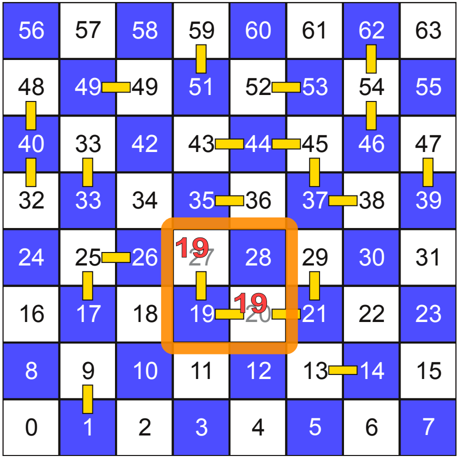


BFS or Ants in the Labyrinth

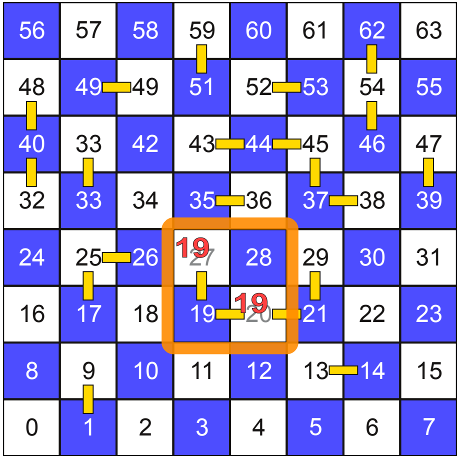


only wave-front vectorization would be possible $\Rightarrow$ many idle threads

Self-labeling

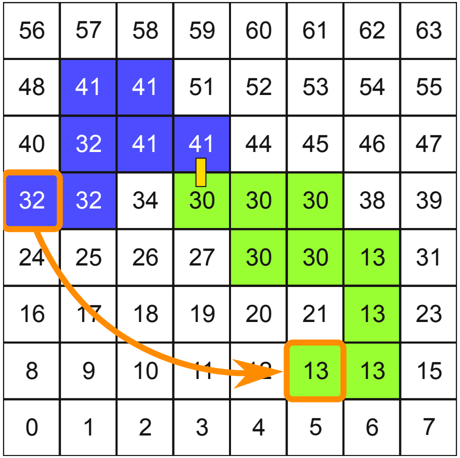


Self-labeling

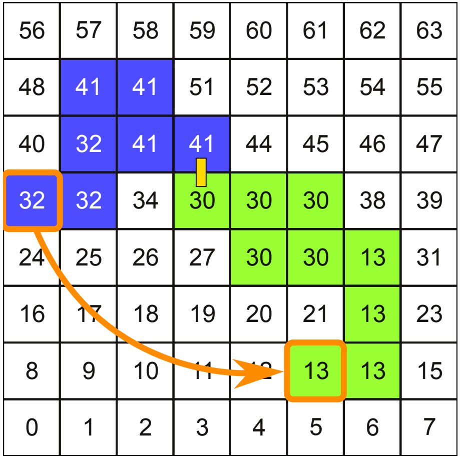


effort is O(L^3) at the critical point, but can be vectorized with O(L^2) threads

Union-find


56 57 58 59 60 61 62 63
48 41 41 51 52 53 54 55
40 32 41 41 44 45 46 47
32 32 34 30 30 30 38 39
24 25 26 27 30 30 13 31
16 17 18 19 20 21 13 23
8 9 10 11 12 13 13 15
0 1 2 3 4 5 6 7

Union-find

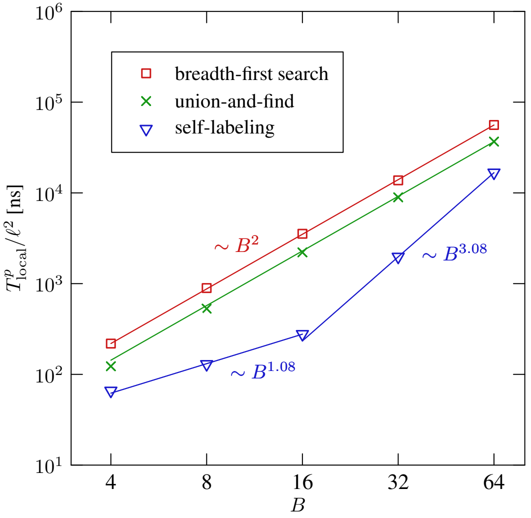


tree structure with two optimizations:

- balanced trees
- path compression

$\Rightarrow$ root finding and cluster union essentially $O(1)$ operations

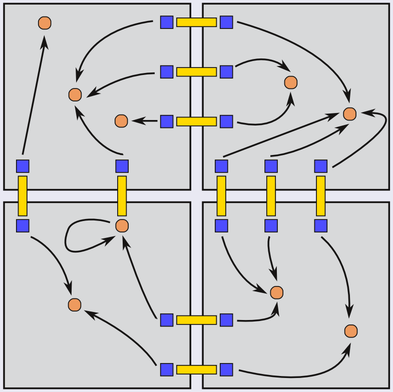
Comparison



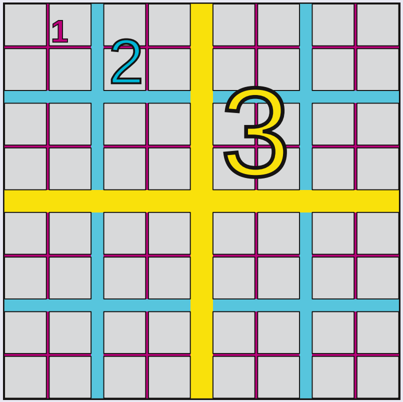
10^6
10^5
10^4
10^3
10^2
10^1
4
8
16
32
64
B
$T^p_{\mathrm{local}}/\ell^2$ [ns]
□ breadth-first search
× union-and-find
▽ self-labeling
$\sim B^2$
$\sim B^{3.08}$
$\sim B^{1.08}$

Label consolidation

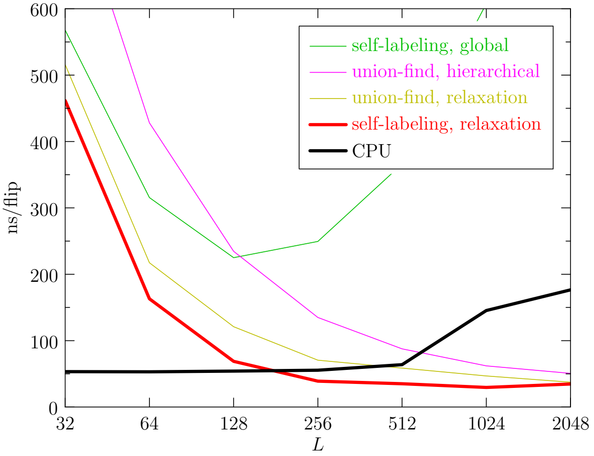
Label relaxation



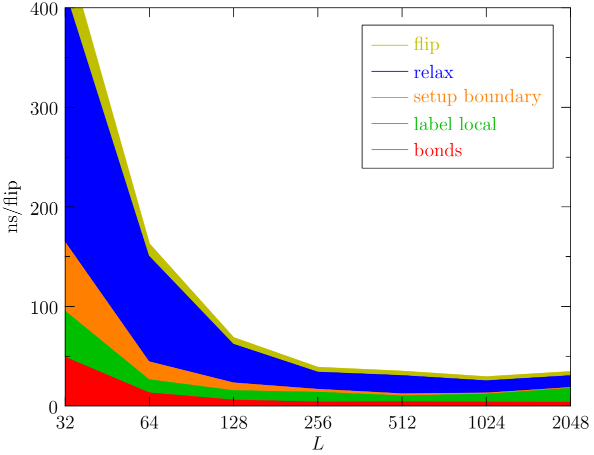
Hierarchical sewing



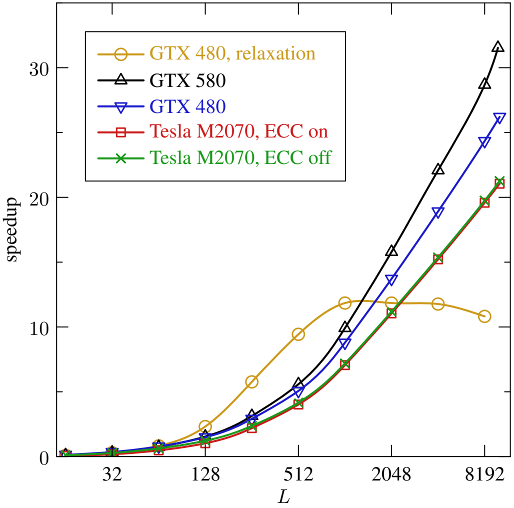
Performance



Performance



Performance

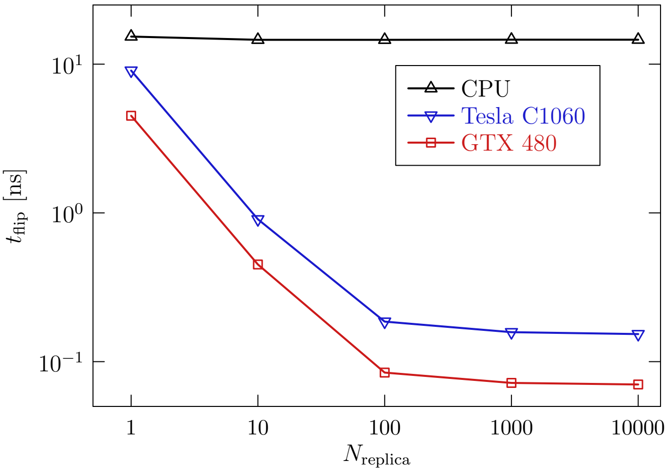


Spin glasses

Simulate Edwards-Anderson model on GPU:

- same domain decomposition (checkerboard)
- slightly bigger effort due to non-constant couplings
- higher performance due to larger independence?
- very simple to combine with parallel tempering

Ising spin glass: performance



Ising spin glass: continued

Seems to work well with

- 15 ns per spin flip on CPU
- 70 ps per spin flip on GPU

but not better than ferromagnetic Ising model.

Ising spin glass: continued

Seems to work well with

- 15 ns per spin flip on CPU
- 70 ps per spin flip on GPU

but not better than ferromagnetic Ising model.

Further improvement: use multi-spin coding

- Synchronous multi-spin coding: different spins in a single configurations in one word
- Asynchronous multi-spin coding: spins from different realizations in one word

Ising spin glass: continued

Seems to work well with

- 15 ns per spin flip on CPU
- 70 ps per spin flip on GPU

but not better than ferromagnetic Ising model.

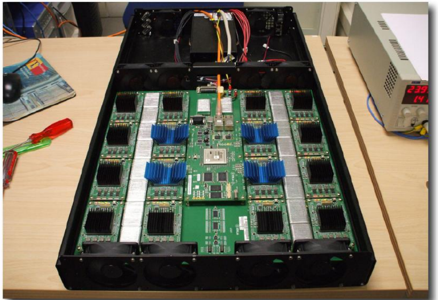
Further improvement: use multi-spin coding

- Synchronous multi-spin coding: different spins in a single configurations in one word
- Asynchronous multi-spin coding: spins from different realizations in one word

$\Rightarrow$ brings us down to about 3 ps per spin flip (in 2D)

Janus

JANUS, a modular massively parallel and reconfigurable FPGA-based computing system.



Janus

JANUS, a modular massively parallel and reconfigurable FPGA-based computing system.

		JANUS		PC		
MODEL	Algorithm	Max size	perfs	AMSC	SMSC	NO MSC
3D Ising EA	Metropolis	96^3	16 ps	$45\times$	$190\times$	
3D Ising EA	Heat Bath	96^3	16 ps	$60\times$		
$Q=4$ 3D Glassy Potts	Metropolis	16^3	64 ps	$1250\times$	$1900\times$	
$Q=4$ 3D disordered Potts	Metropolis	88^3	32 ps	$125\times$		$1800\times$
$Q=4$, $C_m=4$ random graph	Metropolis	24000	2.5 ns	$2.4\times$		$10\times$

Janus

JANUS, a modular massively parallel and reconfigurable FPGA-based computing system.

		JANUS		PC		
MODEL	Algorithm	Max size	perfs	AMSC	SMSC	NO MSC
3D Ising EA	Metropolis	96^3	16 ps	$45\times$	$190\times$	
3D Ising EA	Heat Bath	96^3	16 ps	$60\times$		
$Q = 4$ 3D Glassy Potts	Metropolis	16^3	64 ps	$1250\times$	$1900\times$	
$Q = 4$ 3D disordered Potts	Metropolis	88^3	32 ps	$125\times$		$1800\times$
$Q = 4$, $C_m = 4$ random graph	Metropolis	24000	2.5 ns	$2.4\times$		$10\times$

Costs:

- Janus: 256 units, total cost about $700,000$ Euros
- Same performance with GPU: 64 PCs (2000 Euros) with 2 GTX 295 cards (500 Euros) $\Rightarrow$ $200,000$ Euros
- Same performance with CPU only (assuming a speedup of ~ 50): 800 blade servers with two dual Quadcore sub-units (3500 Euros) $\Rightarrow$ $2,800,000$ Euros

Heisenberg spin glass

Consider EA Heisenberg model:

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

in a random field. Optimal update consists of three parts:

1. Over-relaxation steps (deterministic!):

   $$\vec{s}_i \mapsto 2(\vec{s}_i\cdot\vec{e}^{\,i}_\lambda) - \vec{s}_i,$$

   where $\vec{e}^{\,i}_\lambda = \frac{\vec{\lambda}_i}{|\vec{\lambda}_i|}$ and $\vec{\lambda}_i$ is the local molecular field at site i.

2. Heat-bath update:

   $$\cos\theta = \frac{1}{\beta|\vec{\lambda}_i|}\ln\left[e^{\beta|\vec{\lambda}_i|}(1-R) + e^{-\beta|\vec{\lambda}_i|}R\right]$$

   plus back-rotation into lab frame

3. parallel tempering: exchange at neighboring temperatures with probability

   $$p_{\mathrm{acc}}(\{s_i\},\beta \mapsto \{s_i'\},\beta') = \min\left[1,\ e^{\Delta\beta\Delta E}\right],$$

Heisenberg spin glass

Consider EA Heisenberg model:

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

in a random field.

$\Rightarrow$ maximum performance around 0.45 ns per spin update

Performance

Benchmark results for various models considered:

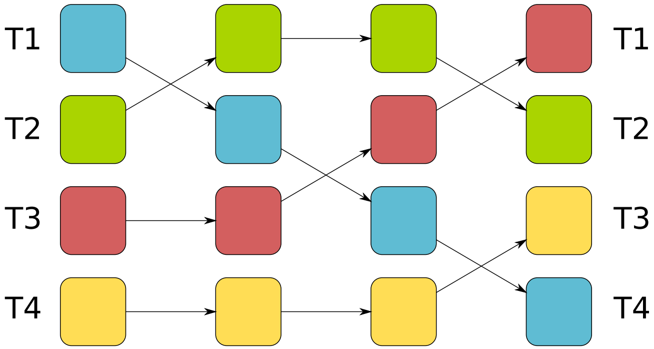
System	Algorithm	L	CPU ns/flip	C1060 ns/flip	GTX 480 ns/flip	speed-up
2D Ising	Metropolis	32	8.3	2.58	1.60	3/5
2D Ising	Metropolis	16 384	8.0	0.077	0.034	103/235
2D Ising	Metropolis, $k = 1$	16 384	8.0	0.292	0.133	28/60
3D Ising	Metropolis	512	14.0	0.13	0.067	107/209
2D Heisenberg	Metro. double	4096	183.7	4.66	1.94	39/95
2D Heisenberg	Metro. single	4096	183.2	0.74	0.50	248/366
2D Heisenberg	Metro. fast math	4096	183.2	0.30	0.18	611/1018
2D spin glass	Metropolis	32	14.6	0.15	0.070	97/209
2D spin glass	Metro. multi-spin	32	0.18	0.0075	0.0023	24/78
2D Ising	Swendsen-Wang	10240	77.4	—	2.97	–/26
2D Ising	multicanonical	64	42.1	—	0.33	–/128
2D Ising	Wang-Landau	64	43.6	—	0.94	–/46

Parallel tempering

One standard technique for systems with complex free energy landscapes is *parallel tempering* or *replica exchange* Monte Carlo.

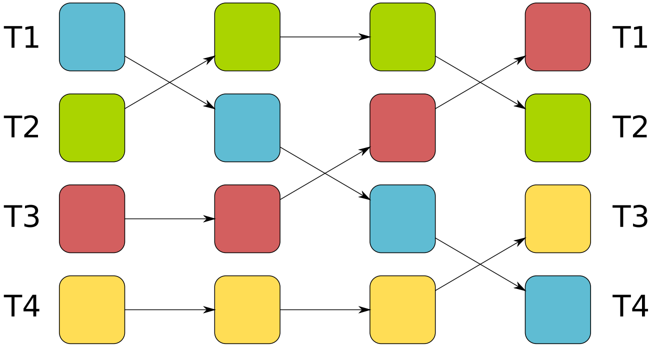
Parallel tempering

One standard technique for systems with complex free energy landscapes is *parallel tempering* or *replica exchange* Monte Carlo.



Parallel tempering

One standard technique for systems with complex free energy landscapes is *parallel tempering* or *replica exchange* Monte Carlo.

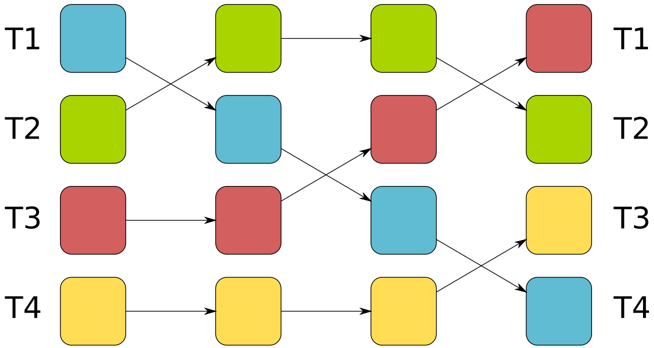


Replica exchanges are subject to Metropolis-Hastings acceptance,

$$p_{\mathrm{acc}} = \min\{1, \exp[(\beta - \beta')(E - E')]\}$$

Parallel tempering

One standard technique for systems with complex free energy landscapes is *parallel tempering* or *replica exchange* Monte Carlo.



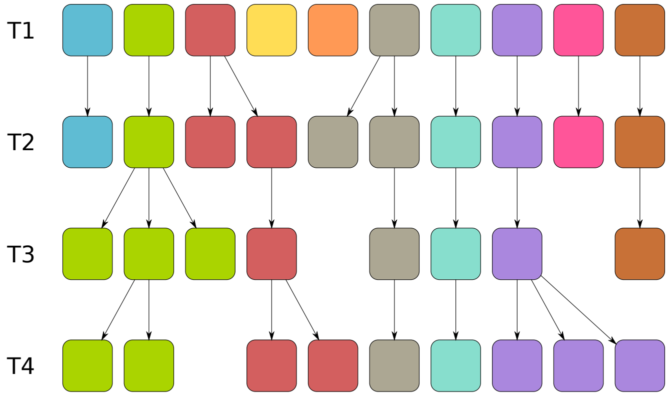
This works very well in parallel, however, typically there is only scope for a moderate number of replicas, say 10-50.

Population annealing

A related technique is *population annealing* (Hukushima, Iba; Machta) which is not a Markov chain method, but sequential Monte Carlo.

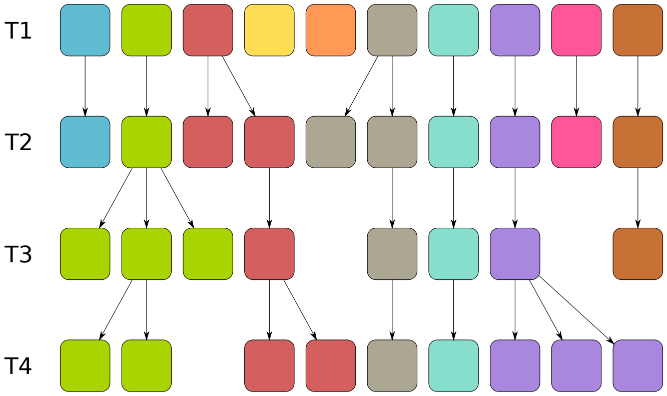
Population annealing

A related technique is *population annealing* (Hukushima, Iba; Machta) which is not a Markov chain method, but sequential Monte Carlo.



Population annealing

A related technique is *population annealing* (Hukushima, Iba; Machta) which is not a Markov chain method, but sequential Monte Carlo.

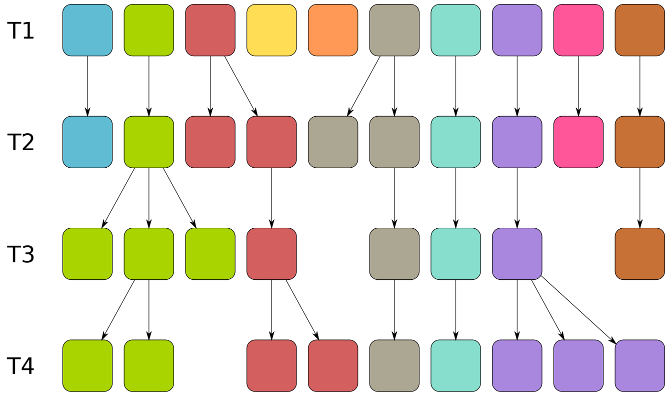


Re-sampling occurs according to the relative Boltzmann weights,

$$\rho_j(\beta, \beta') \propto \exp[-(\beta - \beta')E_j]$$

Population annealing

A related technique is *population annealing* (Hukushima, Iba; Machta) which is not a Markov chain method, but sequential Monte Carlo.



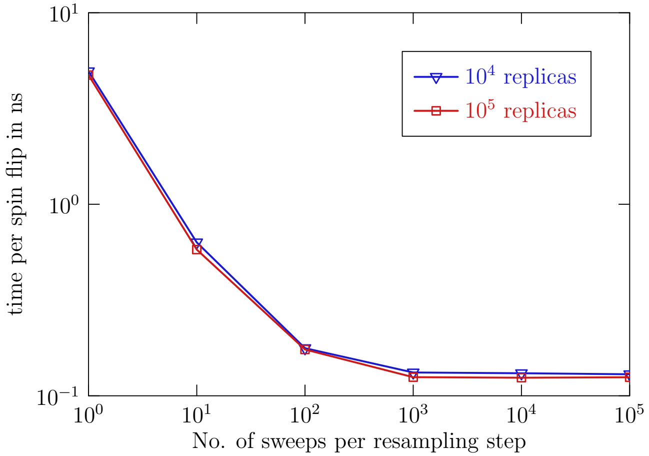
For good results, population sizes of 10^4 to 10^6 replicas are required. Hence this appears to be an ideal match for massively parallel architectures.

Population annealing

In practice, significant decorrelation is achieved through interspersing the resampling with regular spin updates.

Population annealing

In practice, significant decorrelation is achieved through interspersing the resampling with regular spin updates.



Outlook

Conclusions:

- GPGPU promises significant speedups at moderate coding effort
- Requirements for good performance:
  - large degree of locality $\Rightarrow$ domain decomposition
  - suitability for parallelization (blocks) *and* vectorization (threads)
  - total number of threads much larger than processing units (memory latency)
  - opportunity for using shared memory $\Rightarrow$ performance is memory limited
  - ideally continuous variables
- effort significantly smaller than for special-purpose machines
- GPGPU might be a fashion, but CPU computing goes the same way

References:

- MW, Comput. Phys. Commun. **182**, 1833 (2011); J. Comp. Phys. **231**, 3064 (2012).
- MW, Phys. Rev. E 84, 036709 (2011).
- MW and T. Yavors'kii, Physics Procedia **15**, 92 (2011).
- T. Yavors'kii and MW, Eur. Phys. J. Spec. Topics **210**, 159 (2012).
- M. Manssen, MW, and A. K. Hartmann, Eur. Phys. J. Spec. Topics **210**, 53 (2012).
- Code at `http://www.martin-weigel.org/research/gpu-computing/`.

Outlook

Outlook:

- spin-glass simulations
- generalized ensembles and population annealing

References:

- MW, Comput. Phys. Commun. **182**, 1833 (2011); J. Comp. Phys. **231**, 3064 (2012).
- MW, Phys. Rev. E 84, 036709 (2011).
- MW and T. Yavors'kii, Physics Procedia **15**, 92 (2011).
- T. Yavors'kii and MW, Eur. Phys. J. Spec. Topics **210**, 159 (2012).
- M. Manssen, MW, and A. K. Hartmann, Eur. Phys. J. Spec. Topics **210**, 53 (2012).
- Code at `http://www.martin-weigel.org/research/gpu-computing/`.