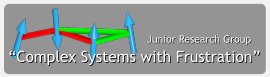
Computational Physics with GPUs

Lecture 3: Simulating spin models I

Martin Weigel

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

41st Heidelberg Physics Graduate Days
Heidelberg, October 8–12, 2018



Slides and exercises

Check out the lecture notes and example code at

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

Any questions? Contact me at

`Martin.Weigel@mail.com`

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

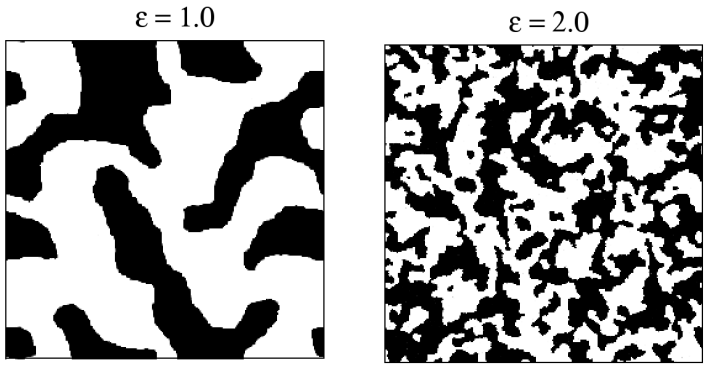
$$\mathcal{H} = -J \sum_{\langle ij \rangle} \vec{S}_i \cdot \vec{S}_j + \vec{H} \cdot \sum_i \vec{S}_i, \quad |\vec{S}_i| = 1$$

Edwards-Anderson spin glass

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

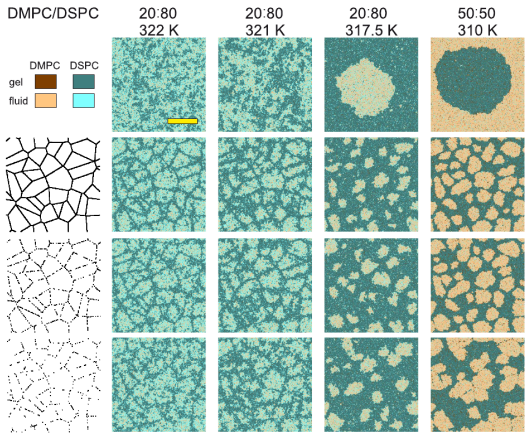
Spin models — Applications

Nucleation and phase ordering



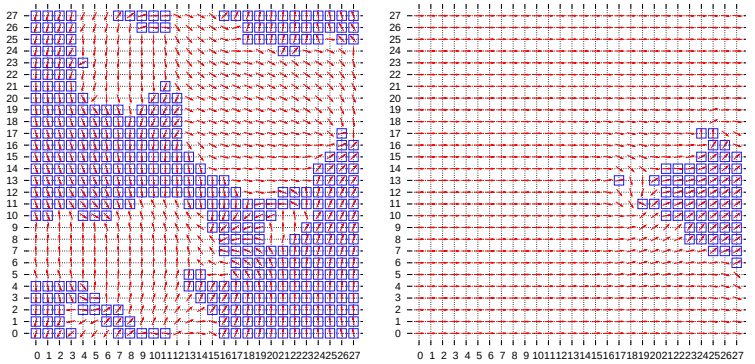
Spin models — Applications

Phase separation in lipid membranes



Spin models — Applications

Quenched disorder in magnetic systems

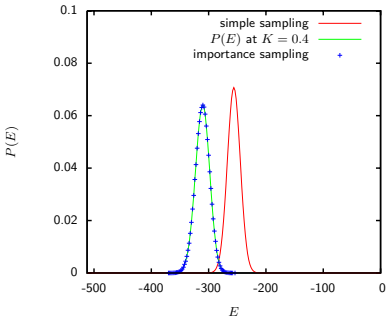


MCMC simulations

Markov-chain Monte Carlo

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

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



MCMC simulations

Markov chains

Simulate Markov chain $\{s_i\}_1 \to \{s_i\}_2 \to \ldots$, such that

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

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

(a) **Ergodicity:**
For each $\{s_i\}$ and $\{s_i'\}$, there exists $n > 0$, such that

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

(b) **Balance:**

$$\sum_{\{s_i'\}} T(\{s_i'\} \to \{s_i\}) p_\beta(\{s_i'\}) = \sum_{\{s_i'\}} T(\{s_i\} \to \{s_i'\}) p_\beta(\{s_i\}) = p_\beta(\{s_i\})$$

i.e., p_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,

$$T(\{s_i'\} \to \{s_i\})p_\beta(\{s_i'\}) = T(\{s_i\} \to \{s_i'\})p_\beta(\{s_i\})$$

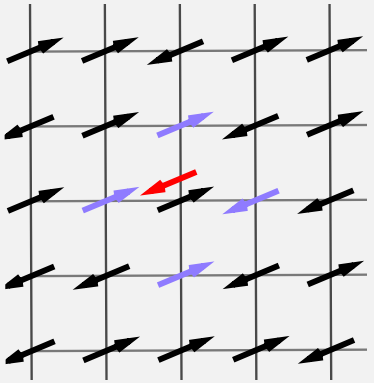
for each pair of states. A possible form of T fulfilling this last condition is

$$T(\{s_i\} \to \{s_i'\}) = \min\left(1, \frac{p_\beta(\{s_i'\})}{p_\beta(\{s_i\})}\right)$$

(Metropolis-Hastings algorithm).

Metropolis update for the Ising model

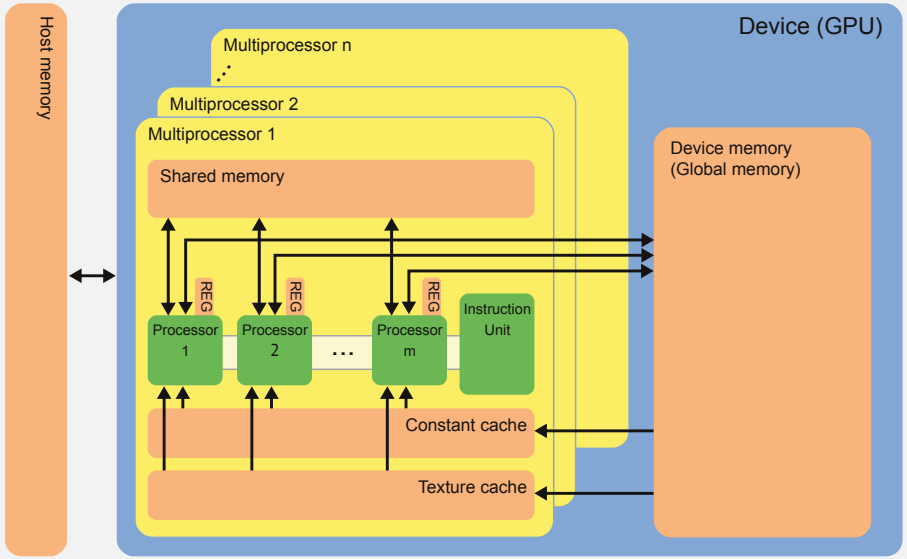
Markov chain Monte Carlo simulation using single spin flips:



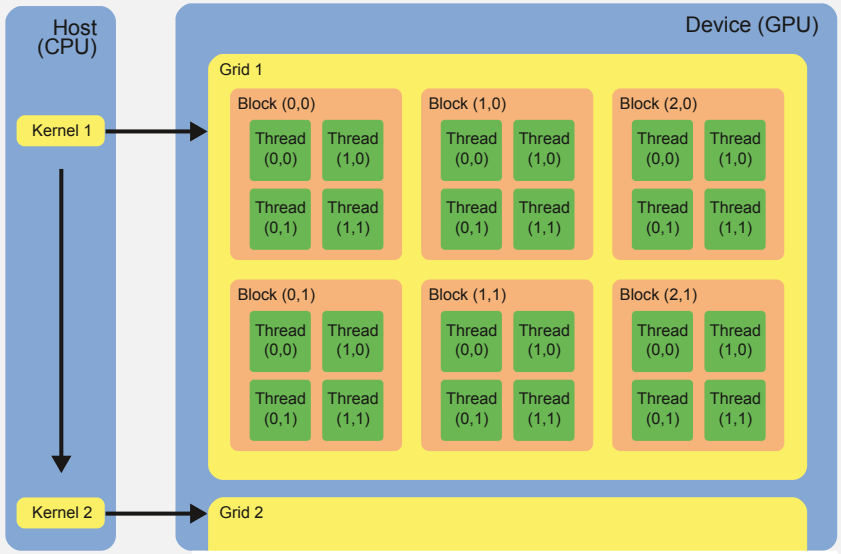
Algorithm

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

NVIDIA architecture



NVIDIA architecture



NVIDIA architecture

Compute model:

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

NVIDIA architecture

Memory layout:

- *Registers*: each multiprocessor is equipped with several thousand registers with local, zero-latency access
- *Shared memory*: processors of a multiprocessor have access a small amount (16–96 KB, depending on GPU generation) of on chip, very small latency shared memory
- *Global memory*: large amount (currently up to 16 GB) of memory on separate DRAM chips with access from every thread on each multiprocessor with a latency of several hundred clock cycles
- *Constant and texture memory*: read-only memories of the same speed as global memory, but cached
- *Host memory*: cannot be accessed from inside GPU functions, relatively slow transfers

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];
        }
      }
    }
  }
}
```

- 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.

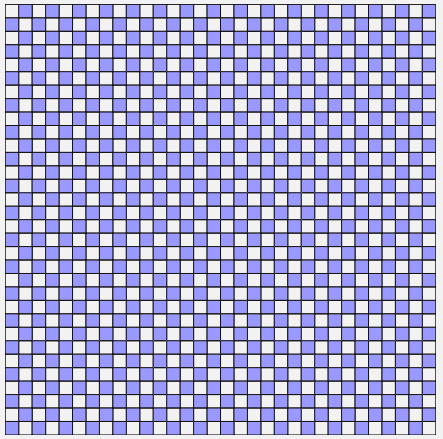
CPU code

```
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. 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

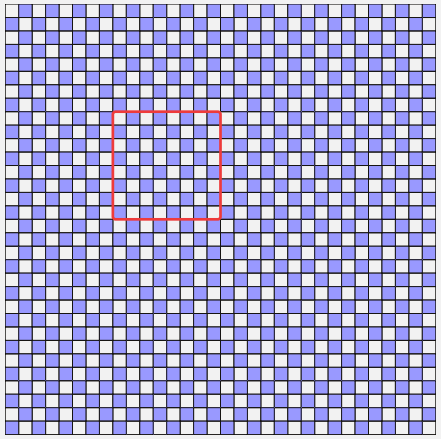
A straightforward minimal code translates the CPU version, using one thread to update each spin of a sub-lattice.

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

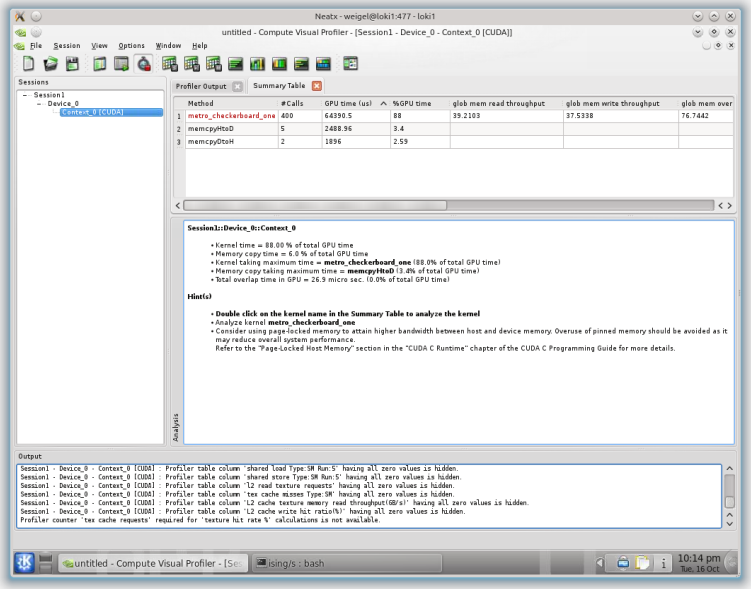
Compare the serial CPU code and the first GPU version:

- 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

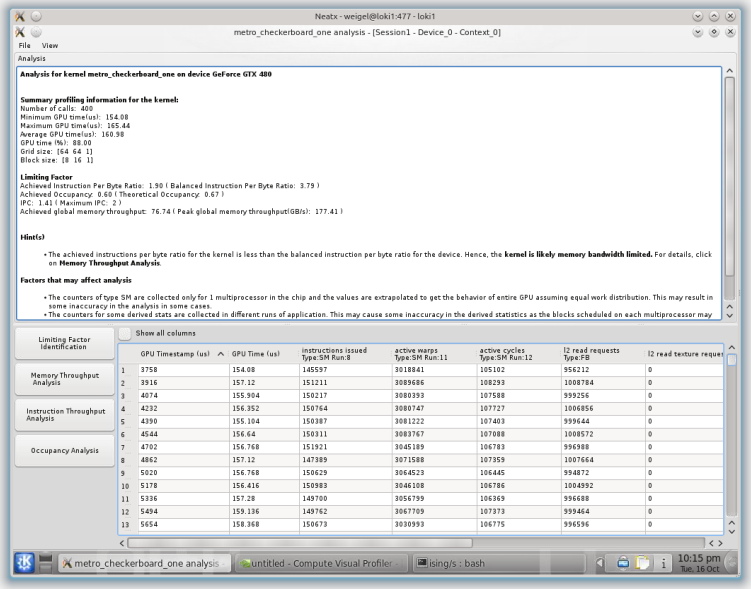
How to improve on this?

- 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

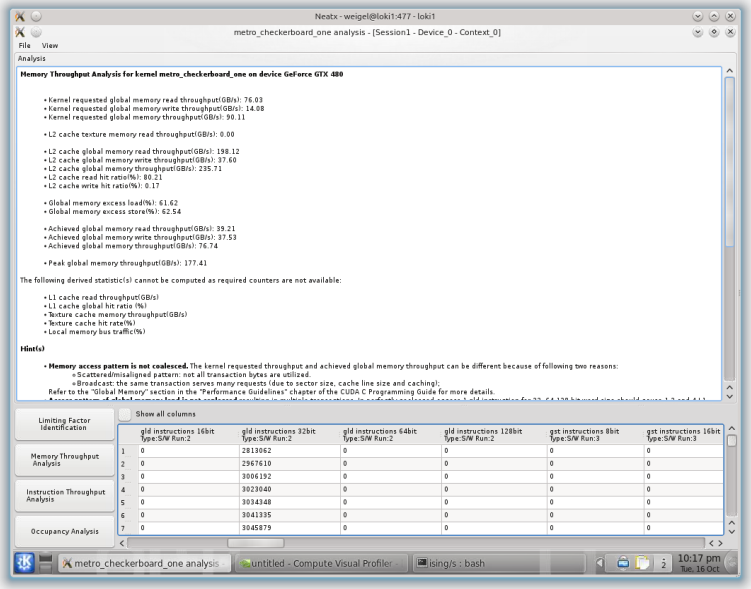
Compute visual profiler



Compute visual profiler



Compute visual profiler



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.* "

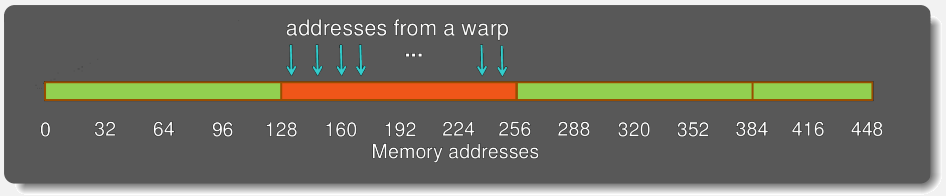
In Fermi and Kepler:

- 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

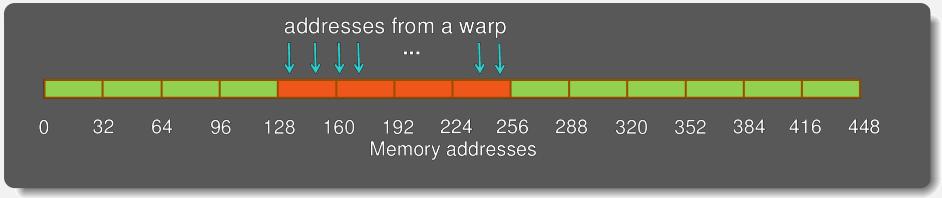
- **cached load**
- warp requests aligned to 32 bytes
- accessing consecutive 4-byte words
- addresses lie in one cache line
- efficiency:
  - warp needs 128 bytes
  - 128 bytes are transferred on a cache miss
  - bus utilization 100%



Memory coalescence

Access pattern

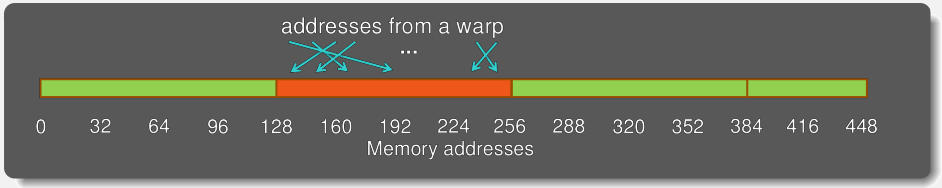
- **L2 only load**
- warp requests aligned to 32 bytes
- accessing consecutive 4-byte words
- addresses lie in 4 adjacent segments
- efficiency:
  - warp needs 128 bytes
  - 128 bytes are transferred on a cache miss
  - bus utilization 100%



Memory coalescence

Access pattern

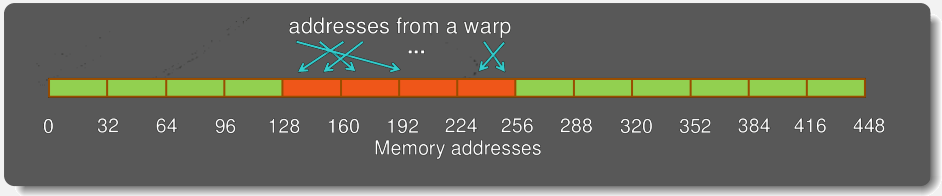
- **cached load**
- warp requests aligned to 32 bytes
- accessing permuted 4-byte words
- addresses lie in one cache line
- efficiency:
  - warp needs 128 bytes
  - 128 bytes are transferred on a cache miss
  - bus utilization 100%



Memory coalescence

Access pattern

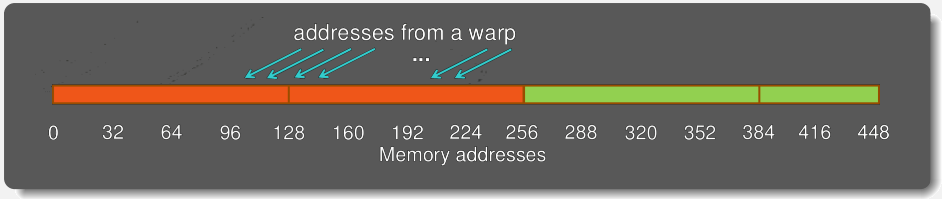
- **L2 only load**
- warp requests aligned to 32 bytes
- accessing permuted 4-byte words
- addresses lie in 4 adjacent segments
- efficiency:
  - warp needs 128 bytes
  - 128 bytes are transferred on a cache miss
  - bus utilization 100%



Memory coalescence

Access pattern

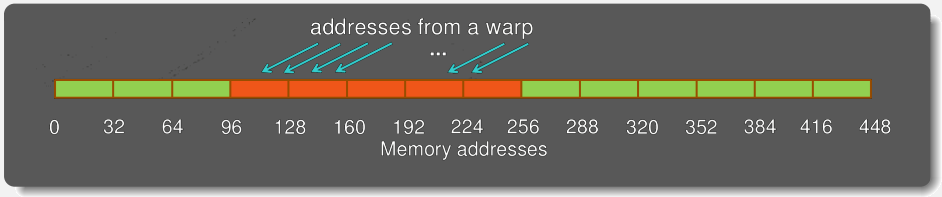
- **cached load**
- warp requests misaligned to 32 bytes
- accessing consecutive 4-byte words
- addresses fall in two adjacent cache lines
- efficiency:
  - warp needs 128 bytes
  - 256 bytes are transferred on a cache miss
  - bus utilization 50%



Memory coalescence

Access pattern

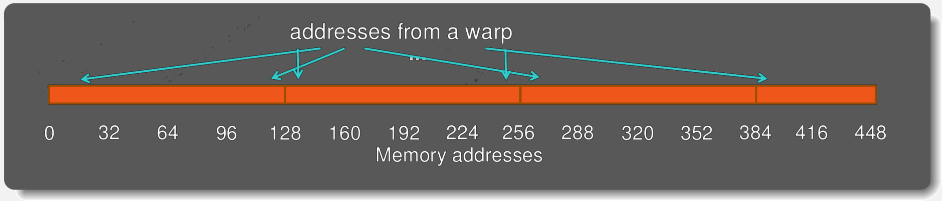
- **L2 only load**
- warp requests misaligned to 32 bytes
- accessing consecutive 4-byte words
- addresses fall in at most 5 segments
- efficiency:
  - warp needs 128 bytes
  - 160 bytes are transferred on cache misses
  - bus utilization at least 80% (100% for some patterns)



Memory coalescence

Access pattern

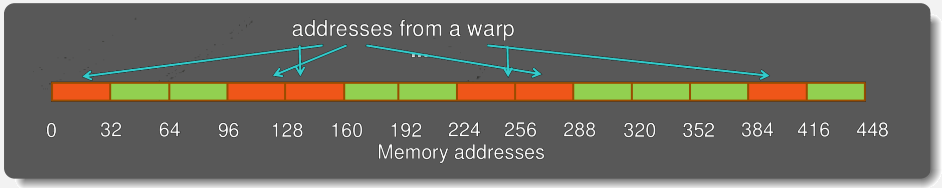
- **cached load**
- warp requests are 32 scattered 4-byte words
- addresses fall in N cache lines
- efficiency:
  - warp needs 128 bytes
  - $N \times 128$ bytes are transferred on cache misses
  - bus utilization $1/N$



Memory coalescence

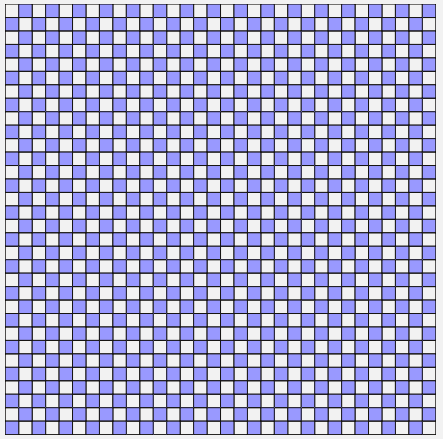
Access pattern

- **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

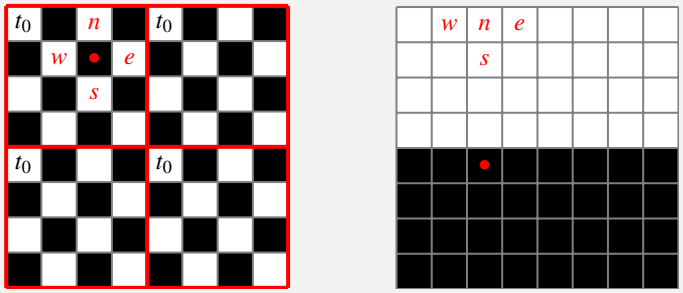
We need to perform updates on non- (or weakly) interacting sub-domains. For bi-partite lattices and nearest-neighbor interactions, this leads to a checkerboard decomposition.



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))

This corresponds to the mapping

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

GPU simulation: second version

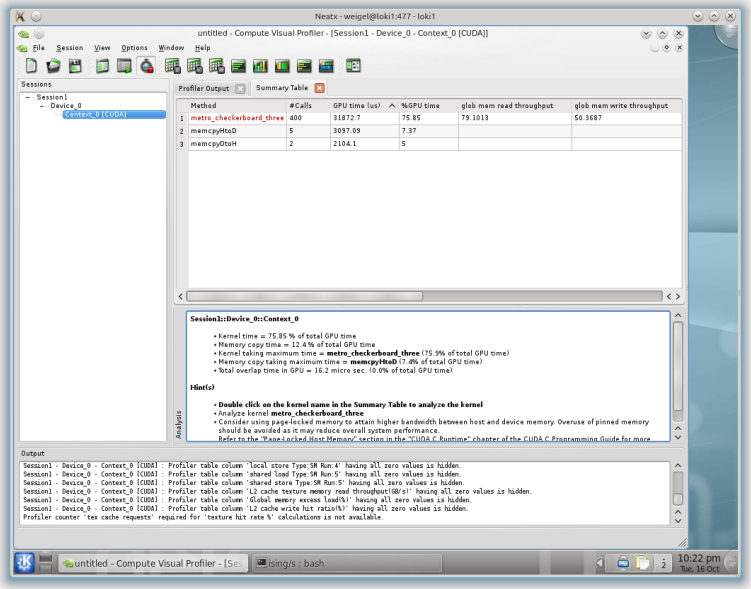
Arrange spins in the crinkled fashion in memory, leading to coalesced accesses to global memory:

GPU code v2 - kernel

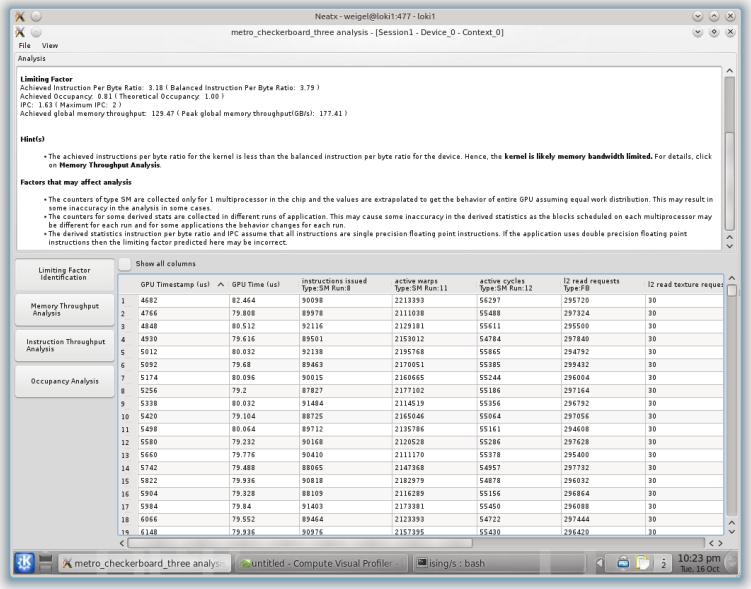
```
__global__ void metro_checkerboard_two(spin_t *s, int *ranvec, int offset)
{
  int n = blockDim.x*blockIdx.x + threadIdx.x;
  int cur = blockDim.x*blockIdx.x + threadIdx.x + offset*(N/2);
  int north = cur + (1-2*offset)*(N/2);
  int east = ((north+1)%L) ? north + 1 : north-L+1;
  int west = (north%L)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

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

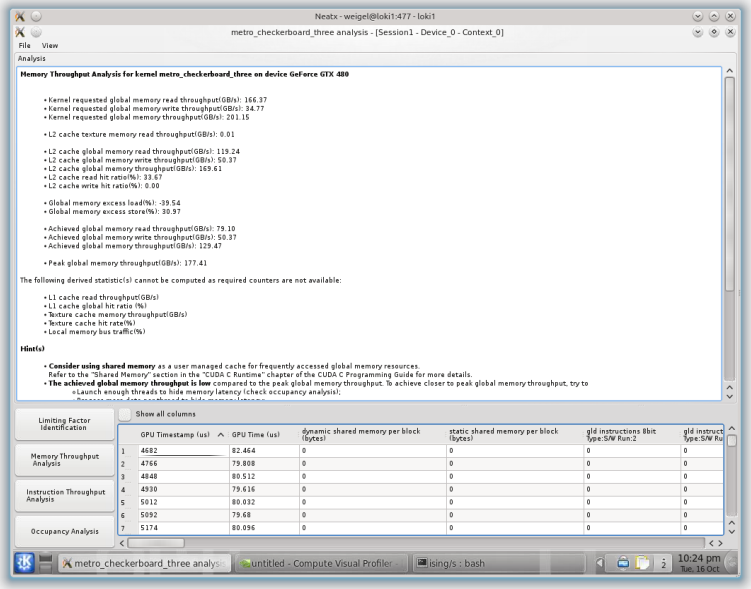
GPU simulation: second version



GPU simulation: second version



GPU simulation: second version



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 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 ...
```

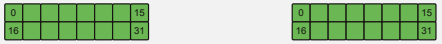

Thread divergence

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

Thread divergence

No serialization

Warp 0 Warp 1

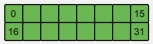



```
if(threadIdx.x < 32) {
    do something
} else {
    do something else
}
...
```


Thread divergence

No serialization

Warp 0 Warp 1

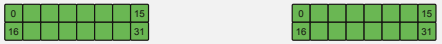



```
if(threadIdx.x < 32) {
    do something
} else {
    do something else
}
...
```


Thread divergence

No serialization

Warp 0 Warp 1




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


Thread divergence

With serialization

Warp 0 Warp 1

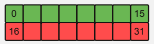

```
if(threadIdx.x < 16) {
    do something
} else {
    do something else
}
...
```

Thread divergence

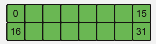
With serialization

Warp 0 | Warp 1

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




```
} else {
    do something else
```




```
}
...
```

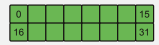

Thread divergence

With serialization

Warp 0 | Warp 1

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


```
}
```



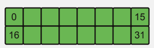

```
...
```


Thread divergence

With serialization

Warp 0 | Warp 1

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



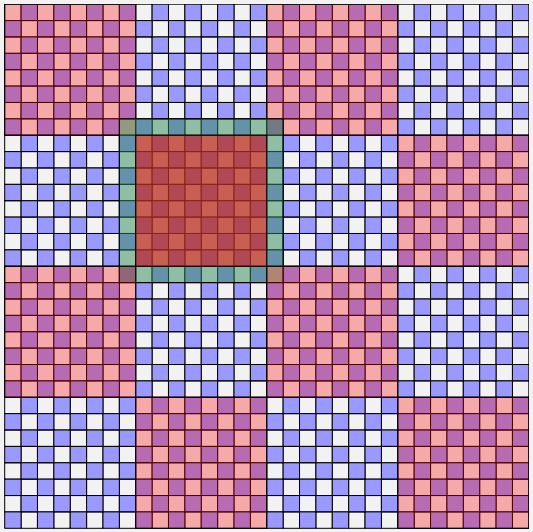
Global memory version: performance

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

What else? Use **shared memory**!

Double checkerboard decomposition

- (red) large tiles: thread blocks
- (red) small tiles: individual threads
- load one large tile (plus boundary) into shared memory
- perform several spin updates per tile



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

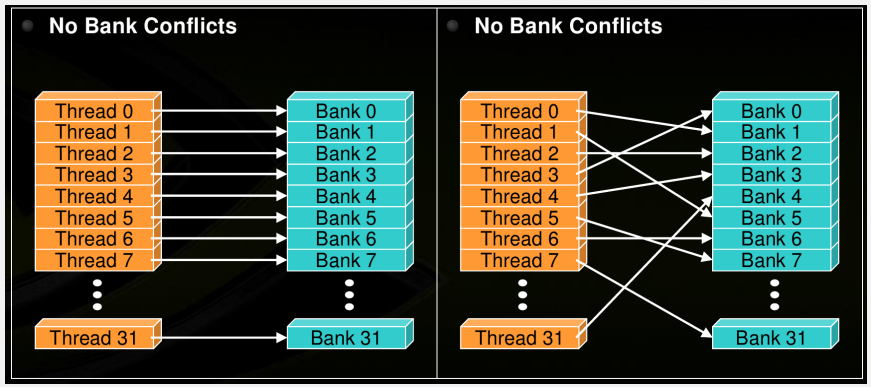
GPU code v5 - kernel 1/2

```
__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

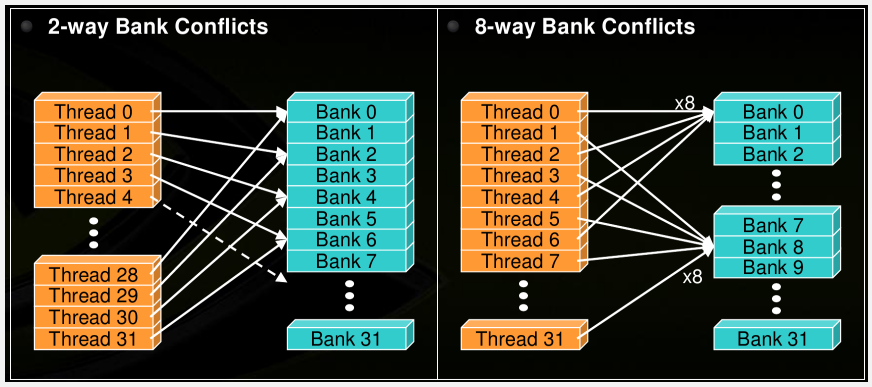
- organization:
  - pre-Fermi: 16 banks, 32-bit wide
  - Fermi: 32 banks, 32-bit wide
  - Kepler and onwards: 32 banks, 32-bit or 64-bit wide chunks
  - successive 4-byte words belong to different banks
- performance: 4 bytes per bank per two clock cycles per SM
- shared memory accesses are issued per warp (32 threads)
- serialization: if n threads of 32 access different 4-byte words in the same bank, n serial accesses are issued
- multicast: n threads can access the same word/bits of the same word in one fetch (Fermi and up)
- standard trick for avoiding conflicts: appropriate **padding**

Shared memory (cont'd)



(Source: NVIDIA)

Shared memory (cont'd)



(Source: NVIDIA)

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
- 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 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

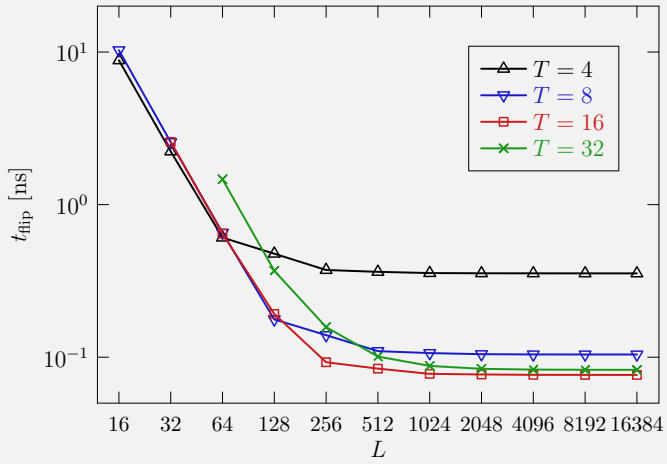
How to assess performance?

- 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

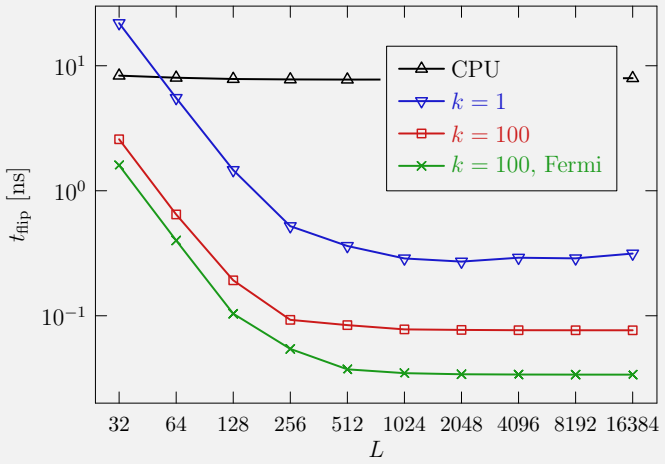
Example: Metropolis simulation of 2D Ising system

- 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

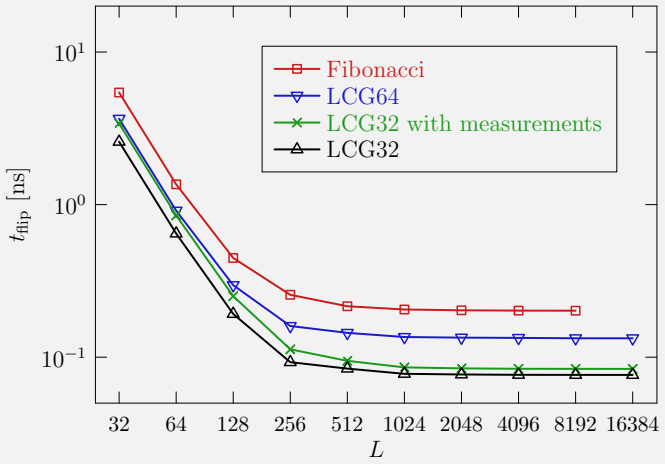
2D Ising ferromagnet



2D Ising ferromagnet



2D Ising ferromagnet

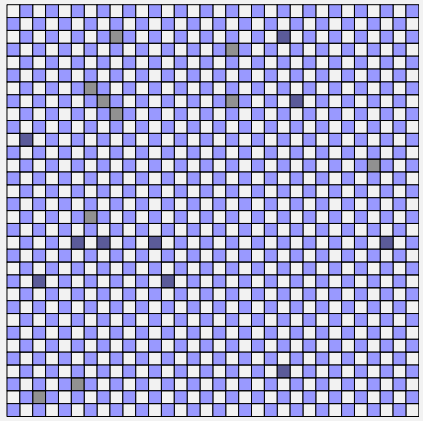


Is it correct?

Detailed balance,

$$T(\{s_i'\} \to \{s_i\})p_\beta(\{s_i'\}) = T(\{s_i\} \to \{s_i'\})p_\beta(\{s_i\}),$$

only holds for *random* updates.



Is it correct?

Detailed balance,

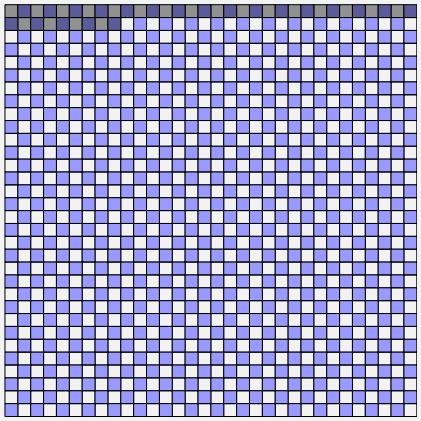
$$T(\{s_i'\} \to \{s_i\})p_\beta(\{s_i'\}) = T(\{s_i\} \to \{s_i'\})p_\beta(\{s_i\}),$$

only holds for *random* updates.

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

$$\sum_{\{s_i'\}} T(\{s_i'\} \to \{s_i\})p_\beta(\{s_i'\}) =$$

$$\sum_{\{s_i'\}} T(\{s_i\} \to \{s_i'\})p_\beta(\{s_i\}) = p_\beta(\{s_i\})$$



Is it correct?

Detailed balance,

$$T(\{s_i'\} \to \{s_i\})p_\beta(\{s_i'\}) = T(\{s_i\} \to \{s_i'\})p_\beta(\{s_i\}),$$

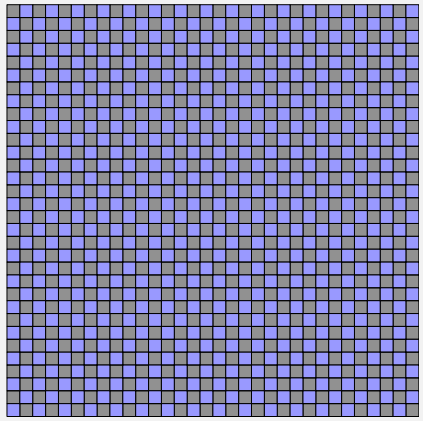
only holds for *random* updates.

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

$$\sum_{\{s_i'\}} T(\{s_i'\} \to \{s_i\})p_\beta(\{s_i'\}) =$$

$$\sum_{\{s_i'\}} T(\{s_i\} \to \{s_i'\})p_\beta(\{s_i\}) = p_\beta(\{s_i\})$$

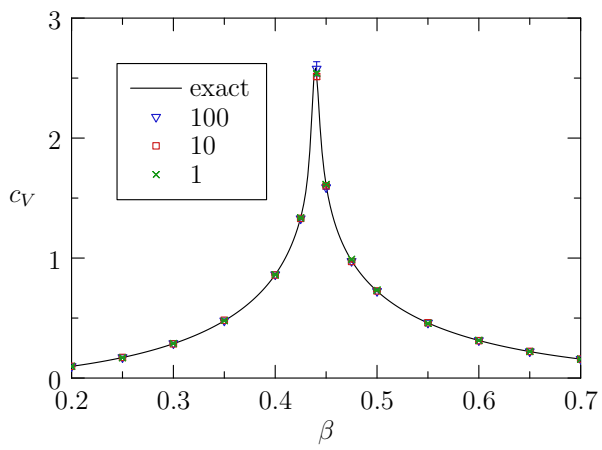
Similarly for checkerboard update. Could restore detailed balance on the level of several sweeps, though:



$$AAAA(M)AAAABBBB(M)BBBBAAAA(M)AAAABBBB(M)BBBB\cdots$$

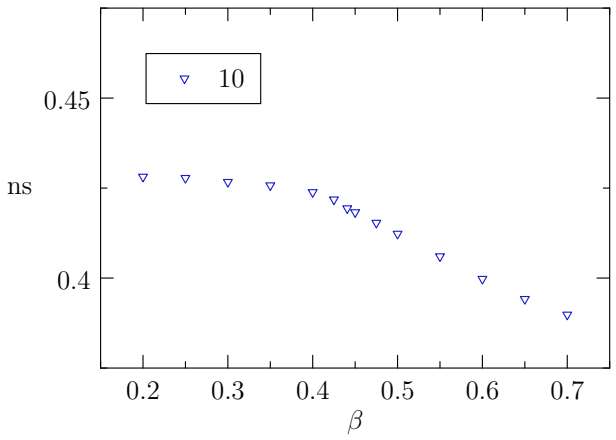
A closer look

Comparison to exact results:



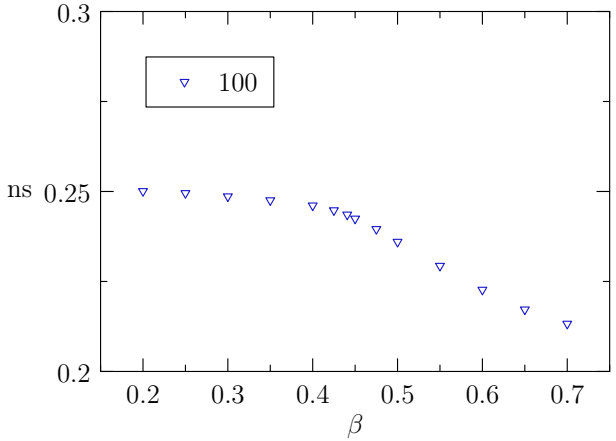
A closer look

Temperature dependent spin flip times:



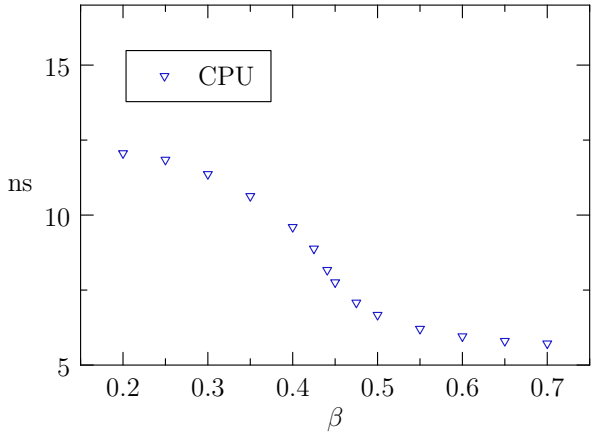
A closer look

Autocorrelation times:



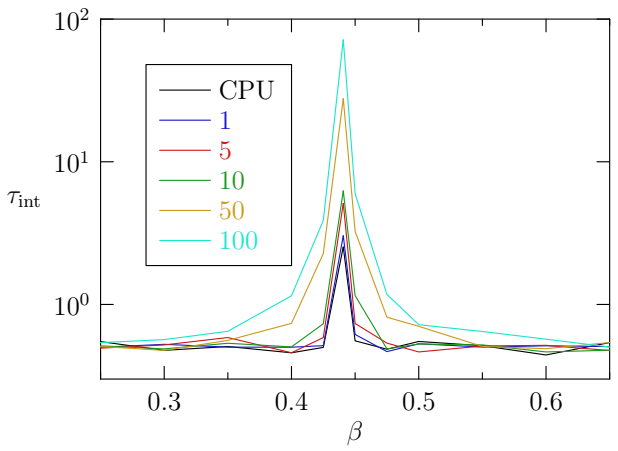
A closer look

Real time to create independent spin configuration:



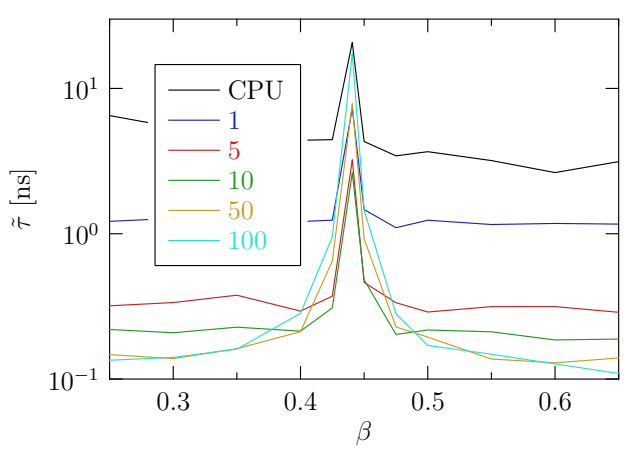
A closer look

Real time to create independent spin configuration:



A closer look

Real time to create independent spin configuration:



Heisenberg model

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

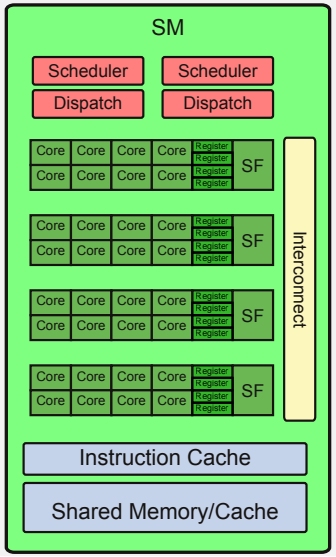
⇒ use same decomposition, but now floating-point computations are dominant:

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

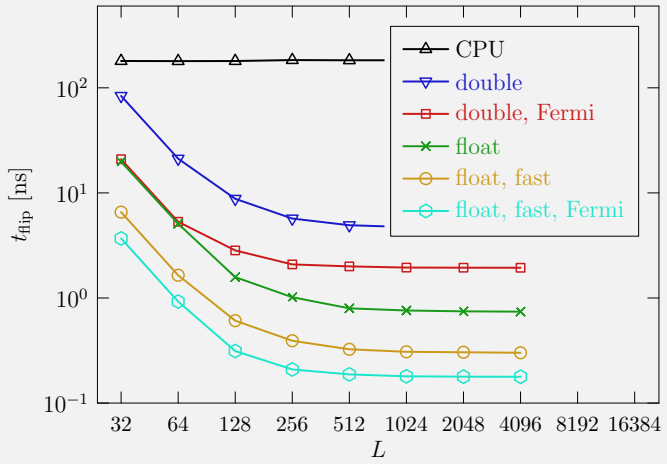
Special function units

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



Heisenberg model: performance

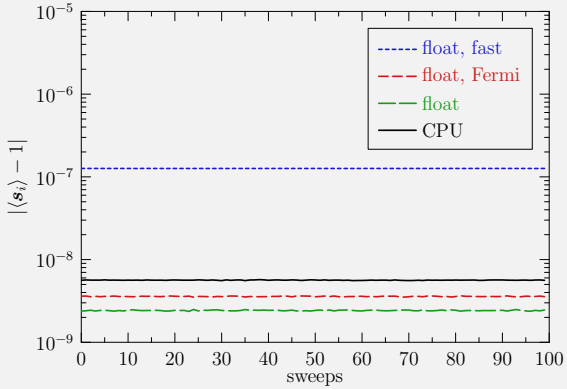


Heisenberg model: stability

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

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. You should be able to write a CUDA program with decent performance for a problem in your field.

Next lecture

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

Reading

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

- M. Weigel, Monte Carlo methods for massively parallel computers [arXiv:1709.04394].
- M. Weigel, Comput. Phys. Commun. **182**, 1833 (2011) [arXiv:1006.3865].
- M. Weigel, J. Comp. Phys. **231**, 3064 [arXiv:1101.1427]

Ising code

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

Tasks:

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