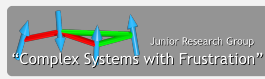


A tutorial on GPU simulations (for spin models)

Martin Weigel

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

CSC@lunch seminar, Centre for Scientific Computing,
University of Warwick, June 1, 2015



GPU computing



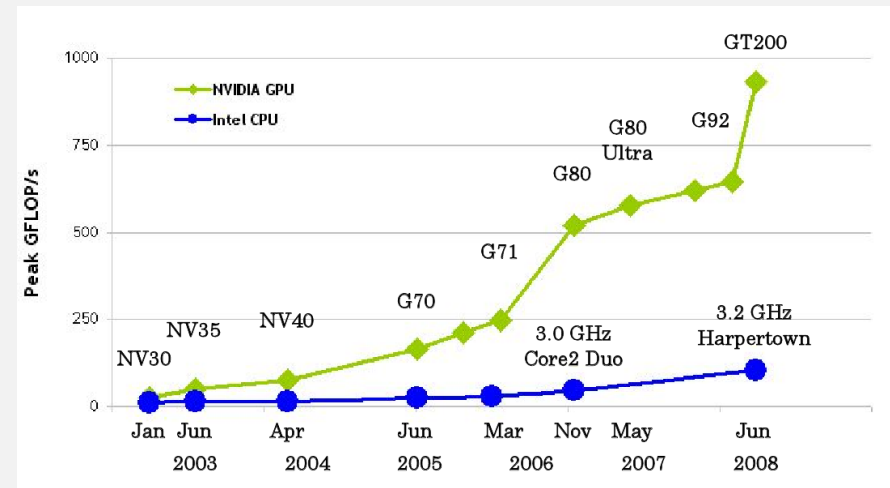
traditional interpretation of GPU computing

GPU computing



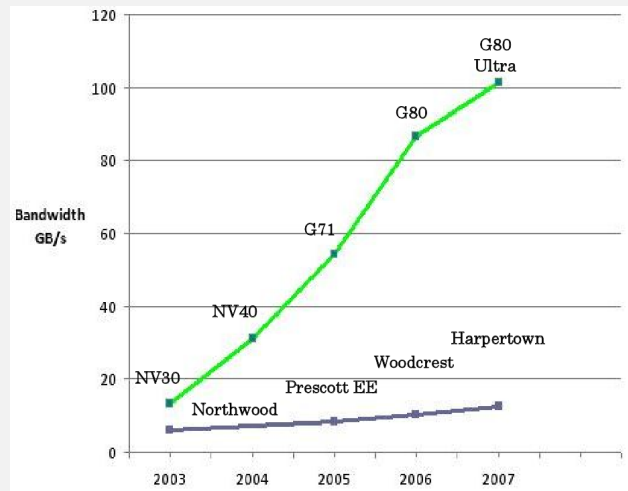
traditional interpretation of GPU computing

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: ≈ 21 GB/s
- NVIDIA Tesla K10: 320 GB/s

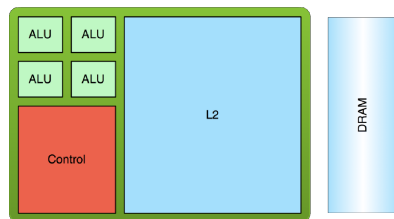
Outline

- 1 GPU architecture
- 2 Spin models
- 3 Single spin flips
- 4 Further systems and algorithms

GPU architecture

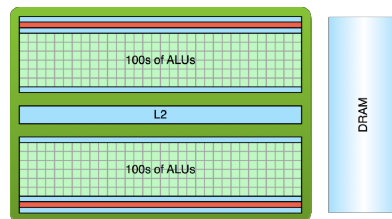
CPU vs. GPU hardware

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

GPU architecture

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

GPGPU history

NVIDIA

- introduced CUDA in 2007
- developed into a fully blown ecosystem
- series of computing cards
- academic support programs
 - CUDA professorships
 - CUDA research centers
 - CUDA teaching centers

ATI

- ATI Stream introduced in 2007
- less viral marketing
- higher peak performance
- somewhat less flexible architecture

Other architectures

- Intel MIC
 - started as Larabee in 2006
 - prototype board Knight's Ferry (2010)
 - 32 cores with 4 threads/core
 - 2 GB DDR5 memory
 - Knight's Corner developed in connection with SGI in 2012
 - more than 50 cores per chip
 - supposedly used in a number of new supercomputers
- (some) special-purpose machines
 - ANTON by D. E. Shaw research, composed of purpose-built ASICs
 - Janus, FPGA based machine for spin-model simulations
 - QPace, based on the Cell processor known from Playstation 3
 - GRAPE, based on FPGAs and used for astrophysical N-body simulations

GPU architecture: main components

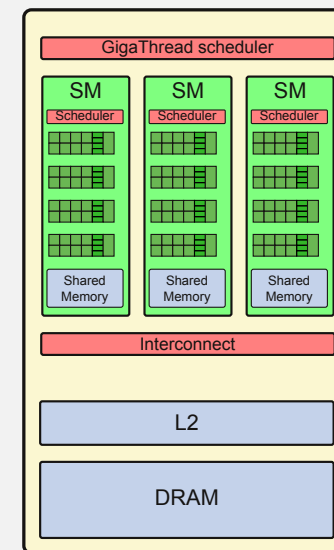
Main memory

- up to 8GB in current GPUs
- maximum bandwidth approx. 160 GB/s (Fermi) resp. 320 GB/s (Kepler)
- accessible from CPU and GPU sides
- large latency (see below)
- optional error correction (ECC on/off, Fermi onwards)

Several multiprocessors

- similar to a multi-core CPU
- each has its own set of registers, scheduler, caches etc.

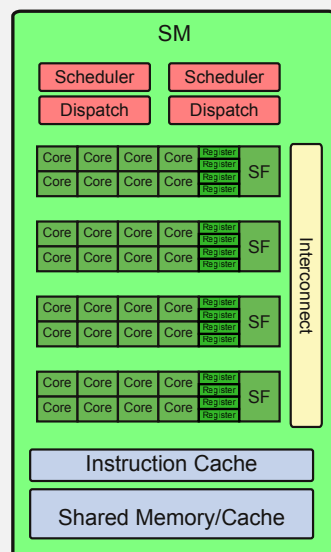
+ scheduling units, PCIe logic etc.



GPU architecture: streaming multiprocessor (Fermi)

SM components

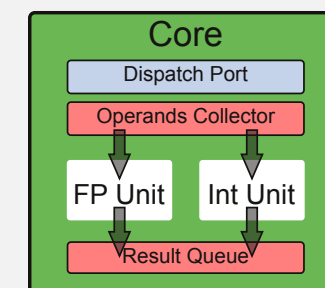
- 32 cores per SM (Fermi)
 - 32 fp32 ops/s
 - 16 fp64 ops/s (Tesla)
 - 32 int32 ops/s
- 2 warp schedulers (warp = 32 threads)
 - up to 1536 threads resident in total
- extra special function units (4 for Fermi)
- 64 KB cache on die, re-configurable as 16 KB cache + 32 KB shared memory or vice versa
- 32K 32-bit registers



GPU architecture: computing core

Computing core

- Integer and floating-point units:
 - IEEE-2008 compliant floating point arithmetic (starting from Fermi)
 - Fused multiply-and-add instruction in hardware
- Logic unit
- Move, compare unit
- Branch unit



Definitions

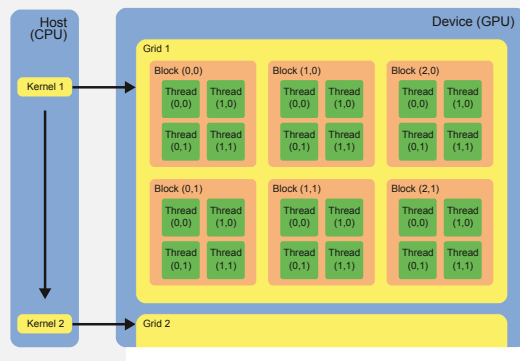
Kernel GPU program that runs on a grid of threads

Thread scalar execution unit

Warp block of 32 threads executed in lockstep

Block a set of warps executed on the same SM

Grid a set of blocks usually executed on different SMs



Threading hierarchy

- Parallel portions of an application are executed on GPU as kernels.
 - one kernel is executed at a time (later on modified in concept of streams)
 - each kernel executes in many threads, but on one device
- Compare CUDA threads to CPU threads
 - CUDA threads are lightweight
 - very little creation overhead
 - low cost of thread switching
 - CUDA needs thousands of threads for efficiency
- Kernels are executed by an array of threads
 - all threads run the same code
 - each thread has a unique `threadid` for control decisions and memory access
- Kernel launched are in grid of thread blocks
 - threads within block cooperate via shared memory
 - threads within a block can synchronize
 - threads within different blocks cannot cooperate (or only via global memory)
 - block executes on one SM and does not migrate
- allows programs to transparently scale to GPUs with different numbers of cores

Memory hierarchy

Per thread

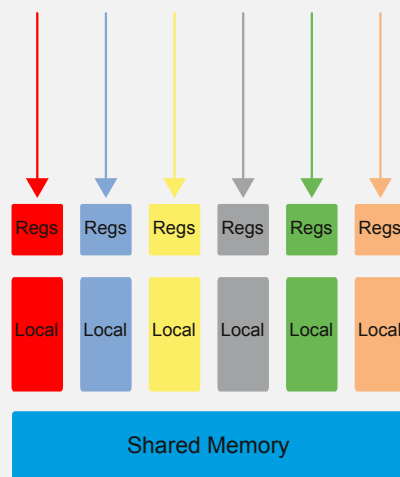
- Registers (extra fast, no copy for ops)
- Local memory

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



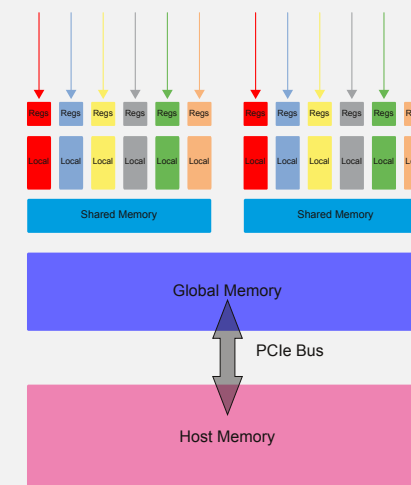
Memory hierarchy

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  $\approx 160$  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

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

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

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

Monte Carlo simulations

Monte Carlo simulations of spin models:

- Metropolis algorithm:

- 1 select a random spin
- 2 compute the induced energy change ΔE
- 3 draw a random number $r \in [0, 1]$ and flip spin if

$$r < \exp(-\Delta E / k_B T)$$

- 4 repeat

- cluster algorithms
- parallel tempering
- multicanonical/Wang-Landau simulations
- ...

CPU implementation

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

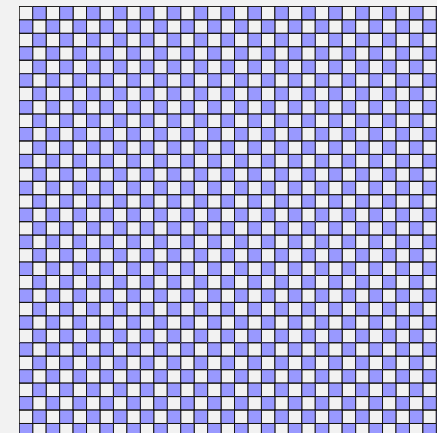
CPU code

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

- 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

Checkerboard decomposition

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



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

GPU simulation: first version

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

GPU code v1 - driver

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

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

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

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

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

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

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

GPU simulation: first version

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

GPU code v1 - kernel

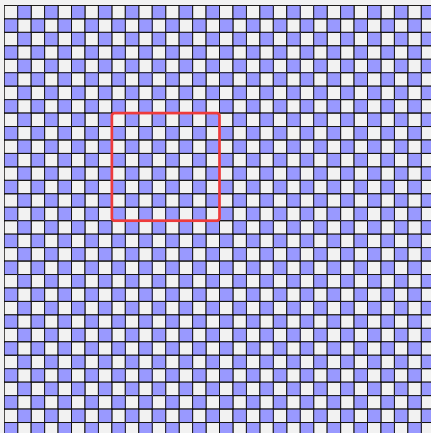
```
typedef int spin_t;

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

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

Checkerboard decomposition

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



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

GPU simulation: first version

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

GPU code v1 - kernel

```
typedef int spin_t;

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

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

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

Improving the code

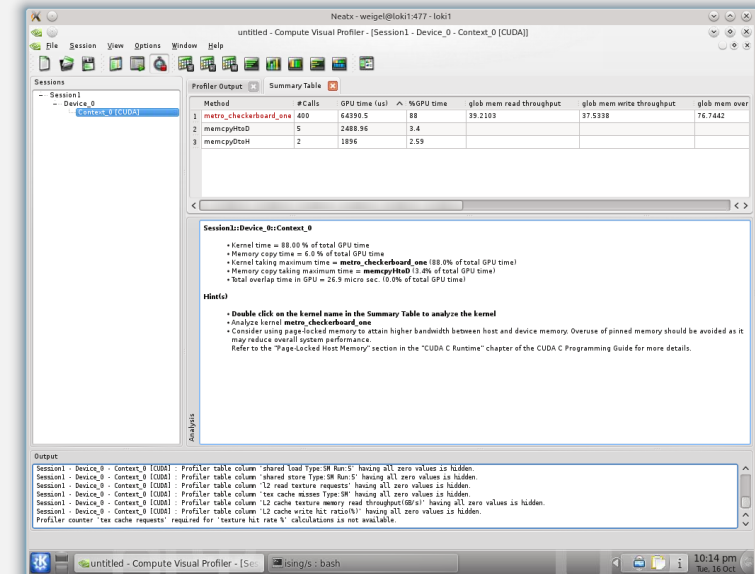
Compare the serial CPU code and the first GPU version:

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

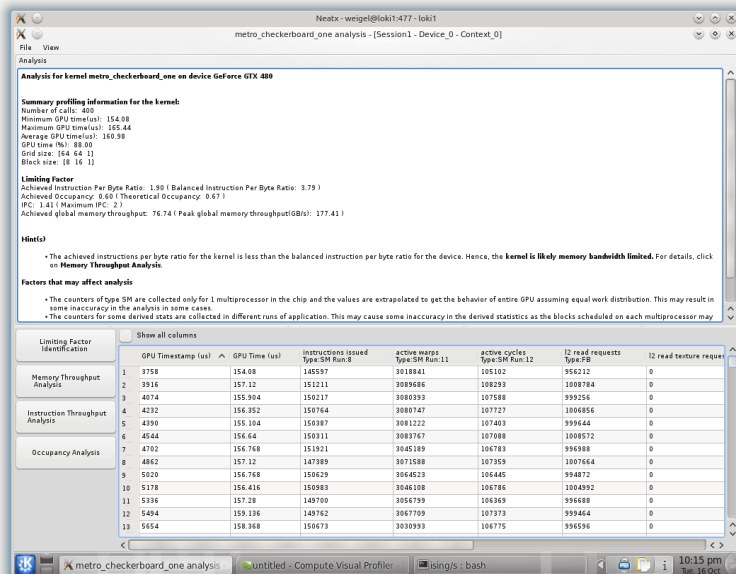
How to improve on this?

- 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

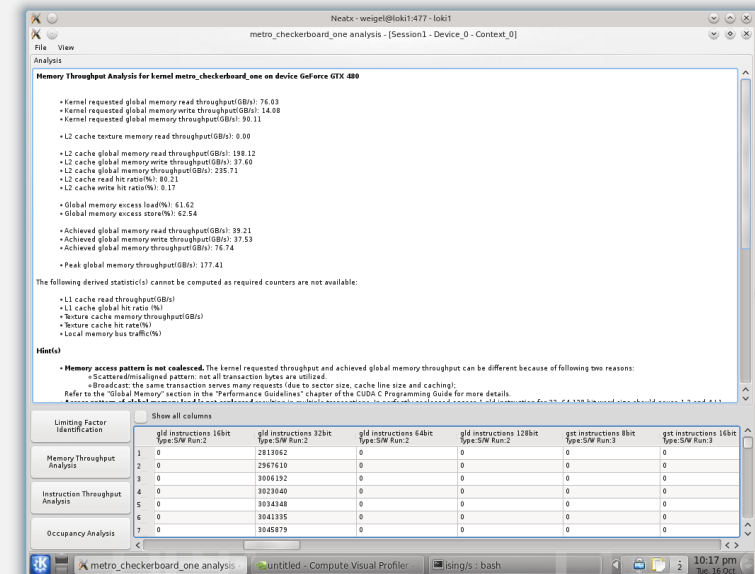
Compute visual profiler



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. ”

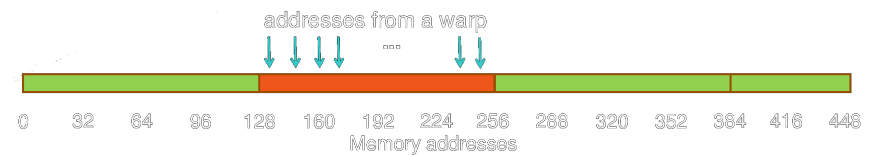
In Fermi and Kepler:

- 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

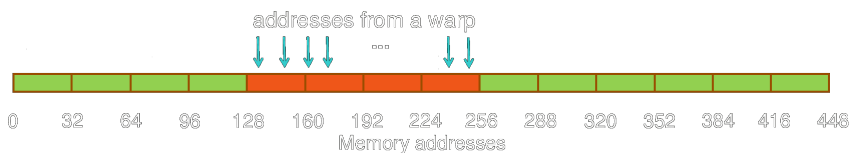
- cached load**
- warp requests aligned to 32 bytes
- accessing consecutive 4-byte words
- addresses lie in one cache line
- efficiency:
 - warp needs 128 bytes
 - 128 bytes are transferred on a cache miss
 - bus utilization 100%



Memory coalescence

Access pattern

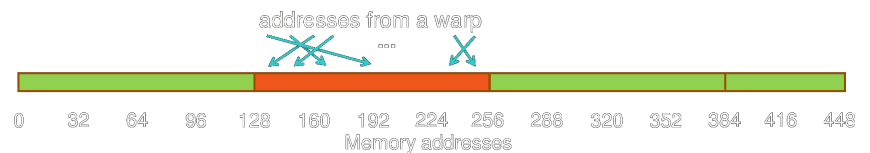
- L2 only load**
- warp requests aligned to 32 bytes
- accessing consecutive 4-byte words
- addresses lie in 4 adjacent segments
- efficiency:
 - warp needs 128 bytes
 - 128 bytes are transferred on a cache miss
 - bus utilization 100%



Memory coalescence

Access pattern

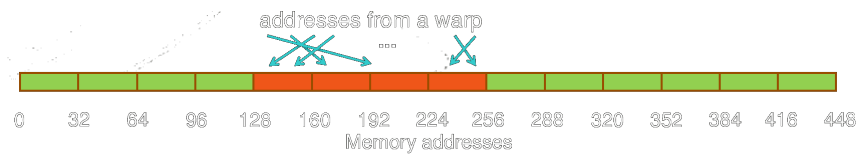
- cached load**
- warp requests aligned to 32 bytes
- accessing permuted 4-byte words
- addresses lie in one cache line
- efficiency:
 - warp needs 128 bytes
 - 128 bytes are transferred on a cache miss
 - bus utilization 100%



Memory coalescence

Access pattern

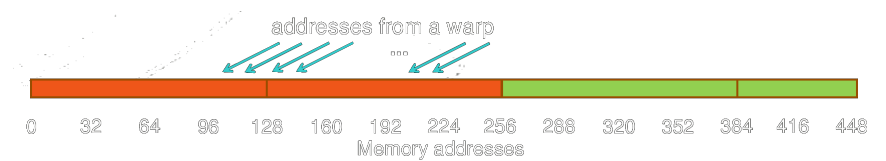
- **L2 only load**
- warp requests aligned to 32 bytes
- accessing permuted 4-byte words
- addresses lie in 4 adjacent segments
- efficiency:
 - warp needs 128 bytes
 - 128 bytes are transferred on a cache miss
 - bus utilization 100%



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

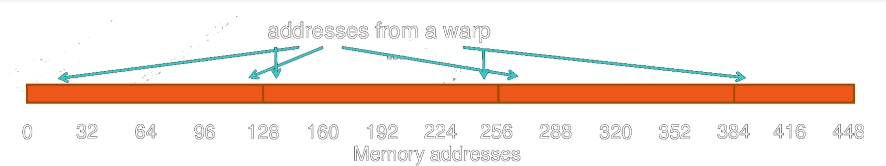
- **L2 only load**
- warp requests misaligned to 32 bytes
- accessing consecutive 4-byte words
- addresses fall in at most 5 segments
- efficiency:
 - warp needs 128 bytes
 - 160 bytes are transferred on cache misses
 - bus utilization at least 80% (100% for some patterns)



Memory coalescence

Access pattern

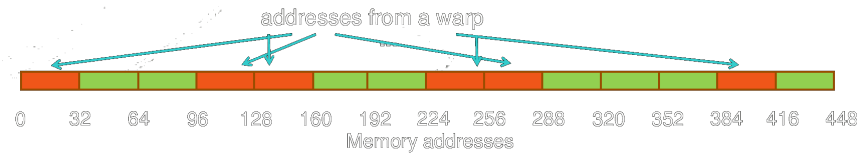
- **cached load**
- warp requests are 32 scattered 4-byte words
- addresses fall in N cache lines
- efficiency:
 - warp needs 128 bytes
 - $N \times 128$ bytes are transferred on cache misses
 - bus utilization $1/N$



Memory coalescence

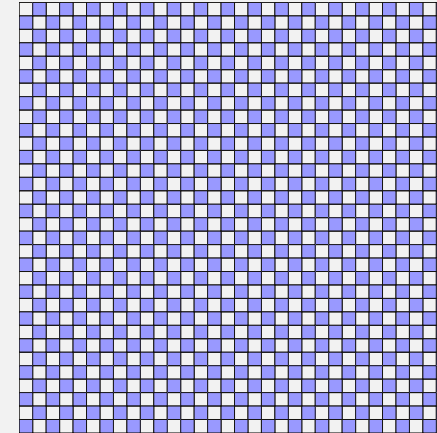
Access pattern

- **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

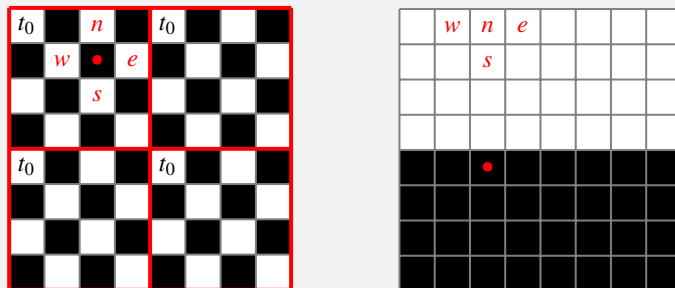
We need to perform updates on non- (or weakly) interacting sub-domains. For bi-partite lattices and nearest-neighbor interactions, this leads to a checkerboard decomposition.



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:

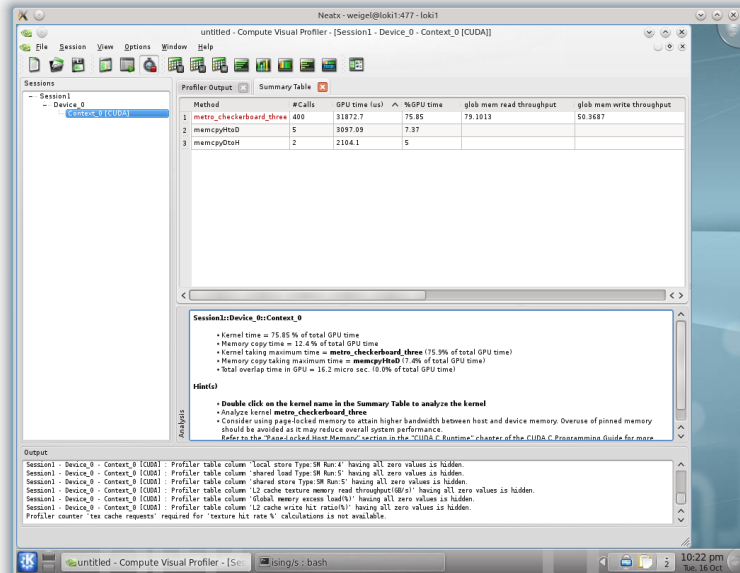
GPU code v2 - kernel

```
__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



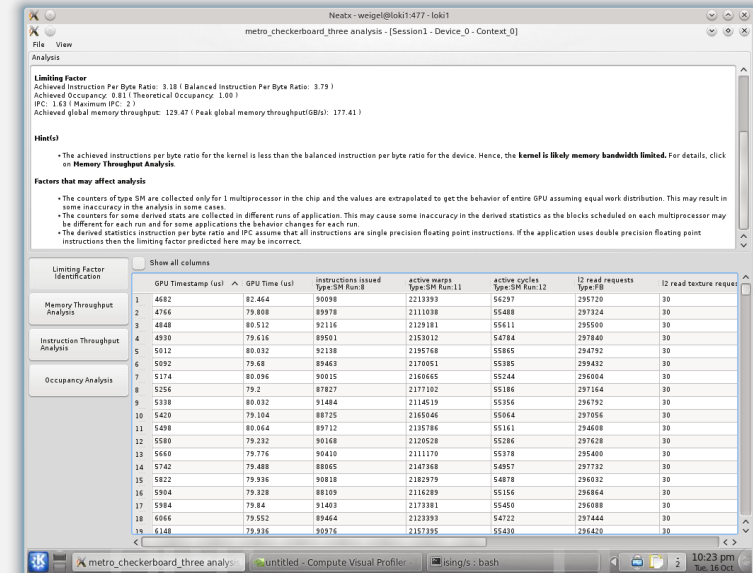
M. Weigel (Coventry/Mainz)

spin models on GPU

01/06/2015

30 / 53

GPU simulation: second version



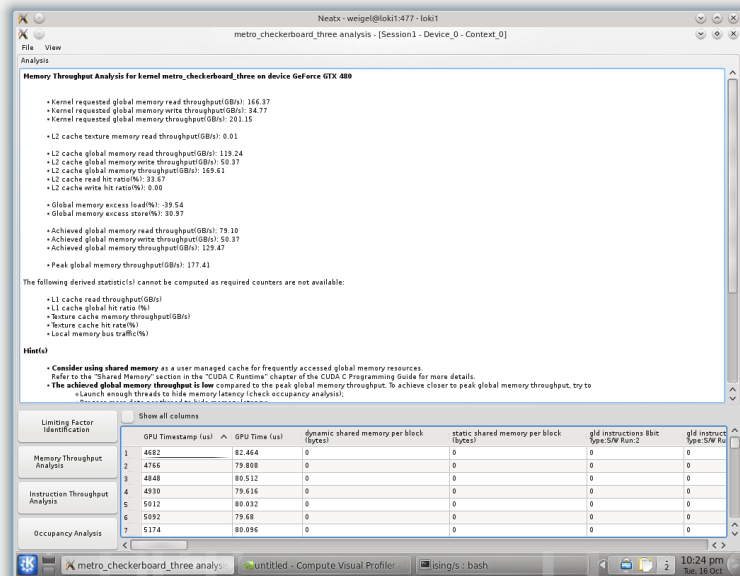
M. Weigel (Coventry/Mainz)

spin models on GPU

01/06/2015

30 / 53

GPU simulation: second version



M. Weigel (Coventry/Mainz)

spin models on GPU

01/06/2015

30 / 53

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., `chars`:

GPU code v3 - kernel

```
typedef char spin_t;
```

M. Weigel (Coventry/Mainz)

spin models on GPU

01/06/2015

31 / 53

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

0					15
16					31

Warp 1

0					15
16					31

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

```
    do something
```

```
} else {
```

```
    do something else
```

```
}
```

```
...
```

Thread divergence

No serialization

Warp 0

0					15
16					31

Warp 1

0					15
16					31

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

```
    do something
```

```
} else {
```

```
    do something else
```

```
}
```

```
...
```

Thread divergence

No serialization

Warp 0

Warp 1

```

if(threadIdx.x < 32) {

    do something

} else {

    do something else

}

...

```

0					15
16					31

0					15
16					31

Thread divergence

With serialization

Warp 0

Warp 1

```

if(threadIdx.x < 16) {

    do something

} else {

    do something else

}

...

```

0					15
16					31

0					15
16					31

Thread divergence

With serialization

Warp 0

Warp 1

```

if(threadIdx.x < 16) {

    do something

} else {

    do something else

}

...

```

0					15
16					31

0					15
16					31

Thread divergence

With serialization

Warp 0

Warp 1

```

if(threadIdx.x < 16) {

    do something

} else {

    do something else

}

...

```

0					15
16					31

0					15
16					31

Thread divergence

With serialization

Warp 0

Warp 1

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

    do something

} else {

    do something else

}

...
```



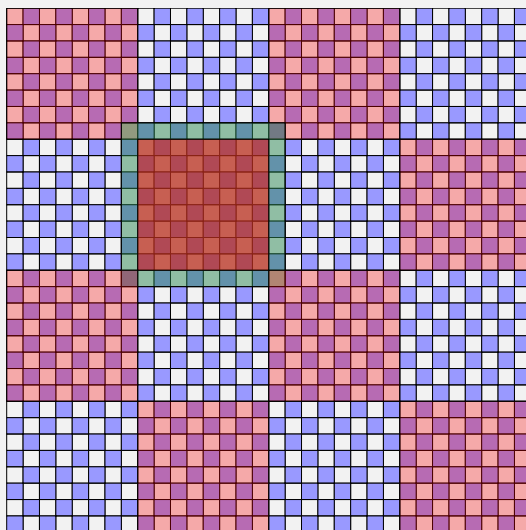
Global memory version: performance

- 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;

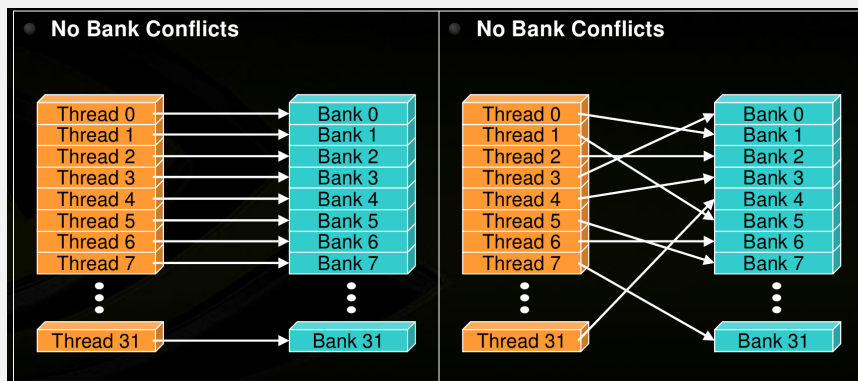
    __shared__ spin_t sS[(BLOCKL+2)*(BLOCKL+2)];

    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

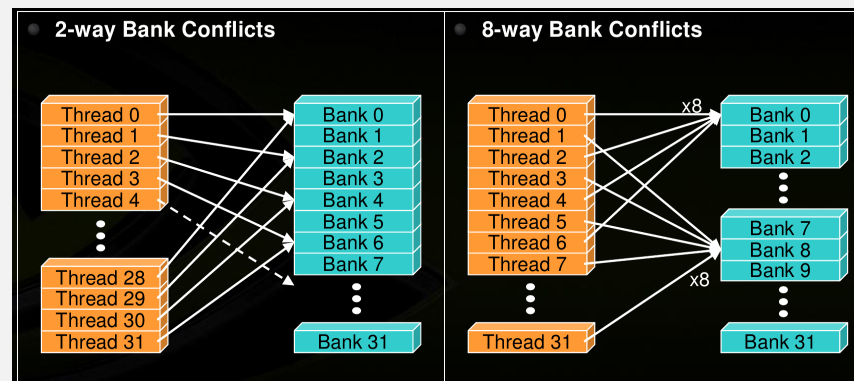
- organization:
 - pre-Fermi: 16 banks, 32-bit wide
 - Fermi: 32 banks, 32-bit wide
 - Kepler: 32 banks, 32-bit or 64-bit wide chunks
 - successive 4-byte words belong to different banks
- performance: 4 bytes per bank per two clock cycles per SM
- shared memory accesses are issued per warp (32 threads)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

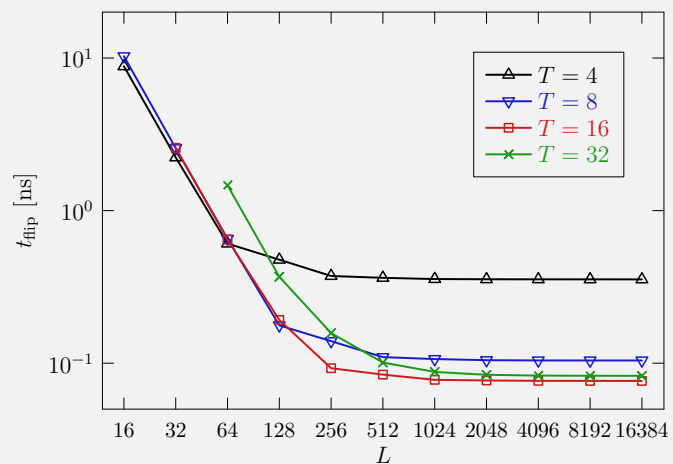
How to assess performance?

- 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

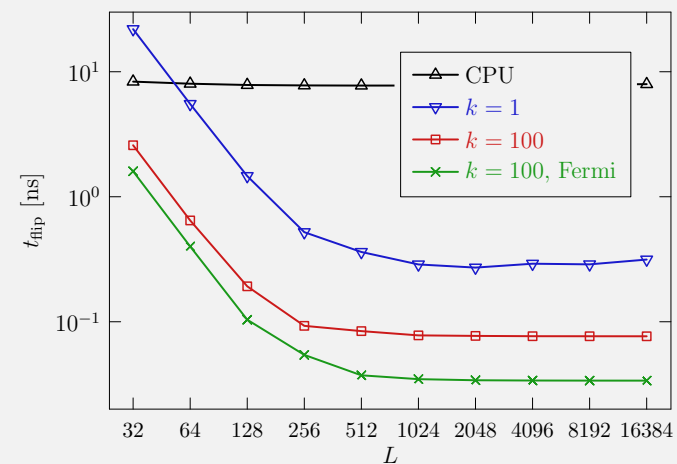
Example: Metropolis simulation of 2D Ising system

- 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

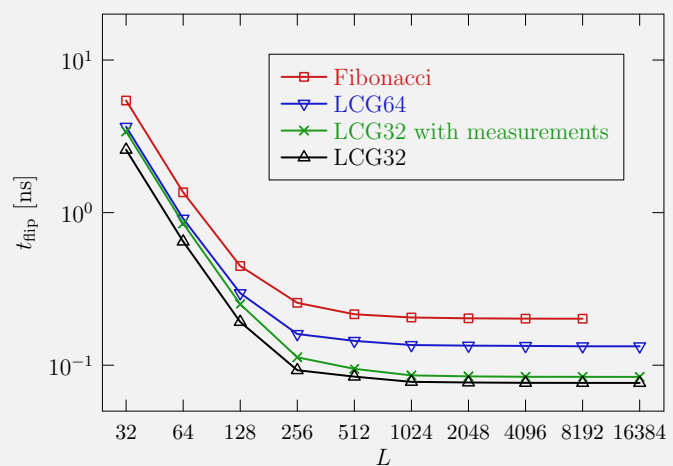
2D Ising ferromagnet



2D Ising ferromagnet



2D Ising ferromagnet

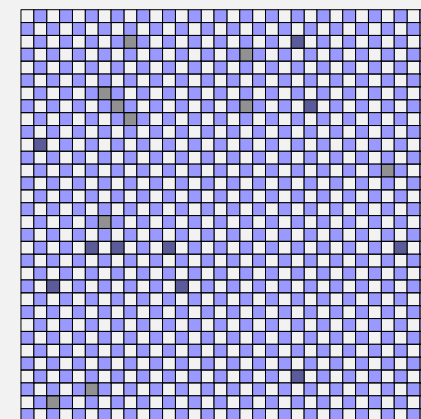


Is it correct?

Detailed balance,

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

only holds for *random* updates.



Is it correct?

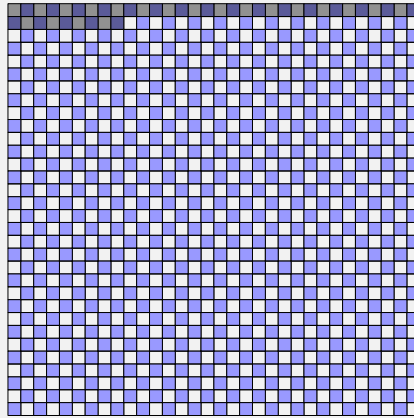
Detailed balance,

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

only holds for *random* updates.

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

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



Is it correct?

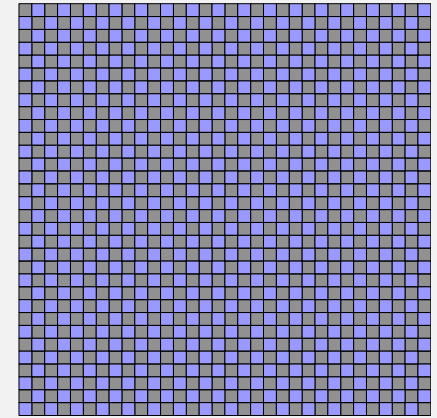
Detailed balance,

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

only holds for *random* updates.

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

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

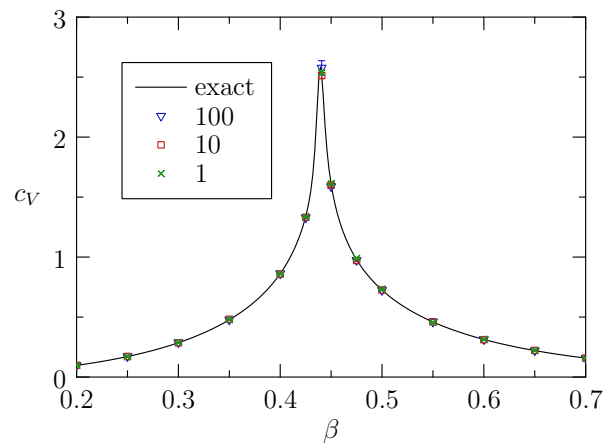


Similarly for checkerboard update. Could restore detailed balance on the level of several sweeps, though:

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

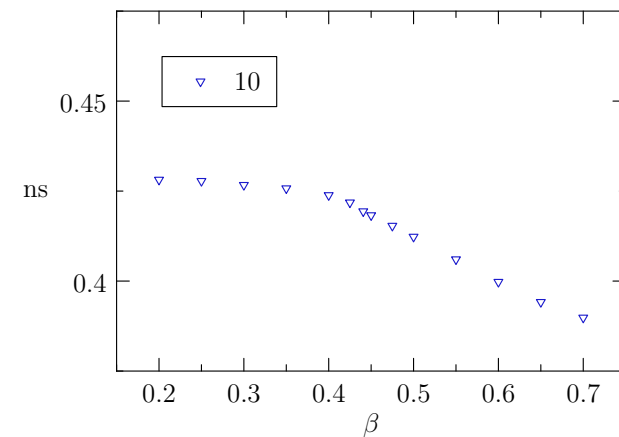
A closer look

Comparison to exact results:



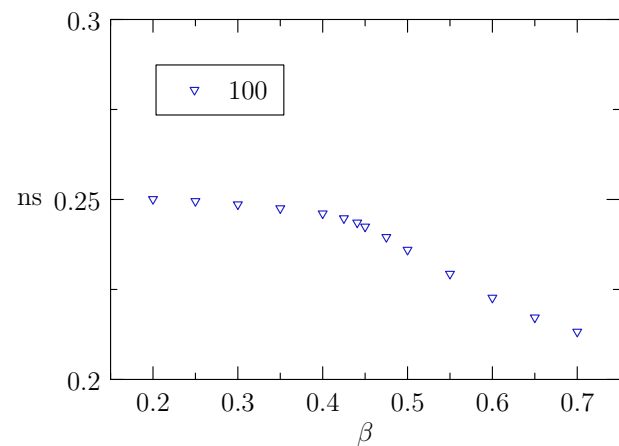
A closer look

Temperature dependent spin flip times:



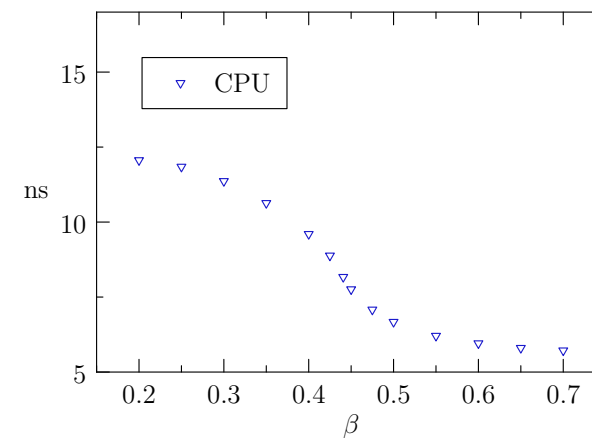
A closer look

Autocorrelation times:



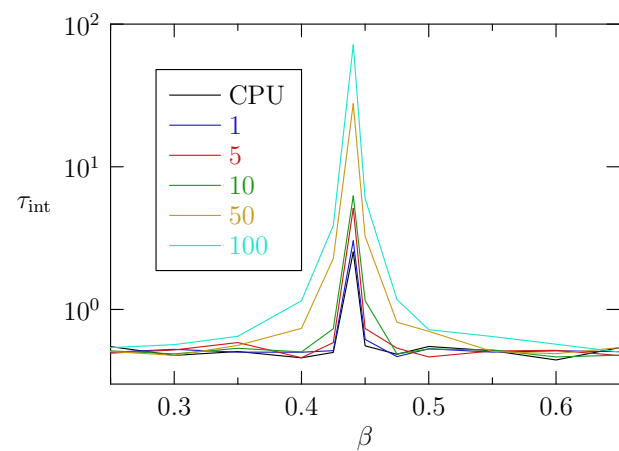
A closer look

Real time to create independent spin configuration:



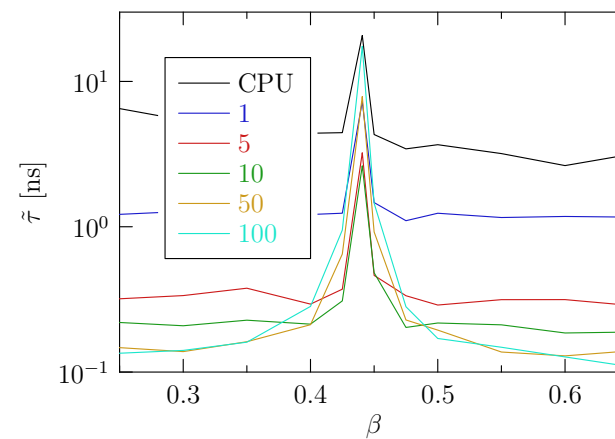
A closer look

Real time to create independent spin configuration:



A closer look

Real time to create independent spin configuration:



Ising Spin glass

Recall Hamiltonian:

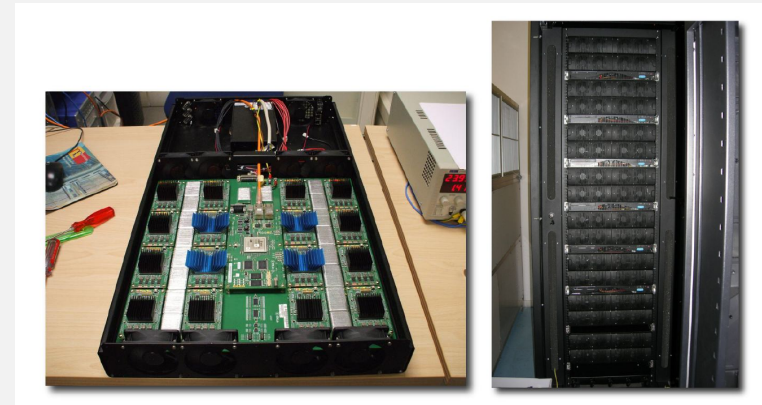
$$\mathcal{H} = - \sum_{\langle i,j \rangle} J_{ij} s_i s_j,$$

where J_{ij} are quenched random variables. For reasonable equilibrium results, average over thousands of realizations is necessary.

- 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

Janus

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



Janus

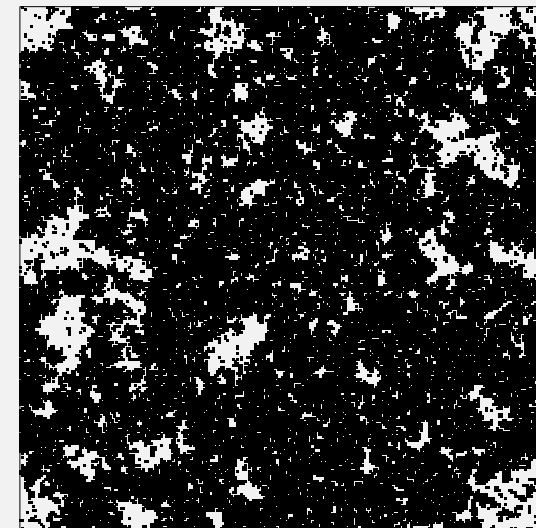
JANUS, a modular massively parallel and reconfigurable FPGA-based computing system.

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

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

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.

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

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

Summary and outlook

Take home message

- tailoring to the hardware needed for really good performance
- special, but general enough to provide significant speed-ups for most problems
- GPU computing might be a fashion, but CPU computing goes the same way

Reading

- Printable version of this lecture:

<http://goo.gl/ePh4sL>

- More information: <http://www.martin-weigel.org/research/gpu-computing/>
- Publications:

M. Weigel, Comput. Phys. Commun. **182**, 1833 (2011)

M. Weigel, J. Comp. Phys. **231**, 3064 (2012)

M. Weigel, Phys. Rev. E **84**, 036709 (2011)

M. Weigel and T. Yavors'kii, Physics Procedia **15**, 92 (2011)

T. Yavors'kii and M. Weigel, Eur. Phys. J. Special Topics **210**, 159 (2012)

M. Manssen, M. Weigel and A. K. Hartmann, Eur. Phys. J. Special Topics **210**, 53 (2012)

Some advertisement: EPJST issue



<http://epjst.epj.org/articles/epjst/abs/2012/10/contents/contents.html>