Ising, Heisenberg and spin glass model simulations on GPU

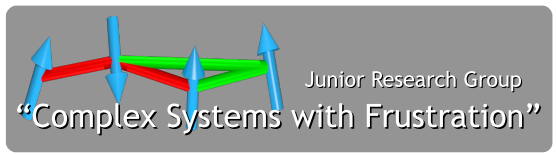
Martin Weigel

Theoretische Physik, Universität des Saarlandes, Saarbrücken, Germany

and

Institut für Physik, Johannes-Gutenberg-Universität Mainz, Germany

Mini-Workshop “Simulations on GPU”
Leipzig, June 11–12, 2009



GPU computation frameworks

GPGPU = General Purpose Computation on Graphics Processing Unit

"Old" times: use original graphics primitives

- OpenGL
- DirectX

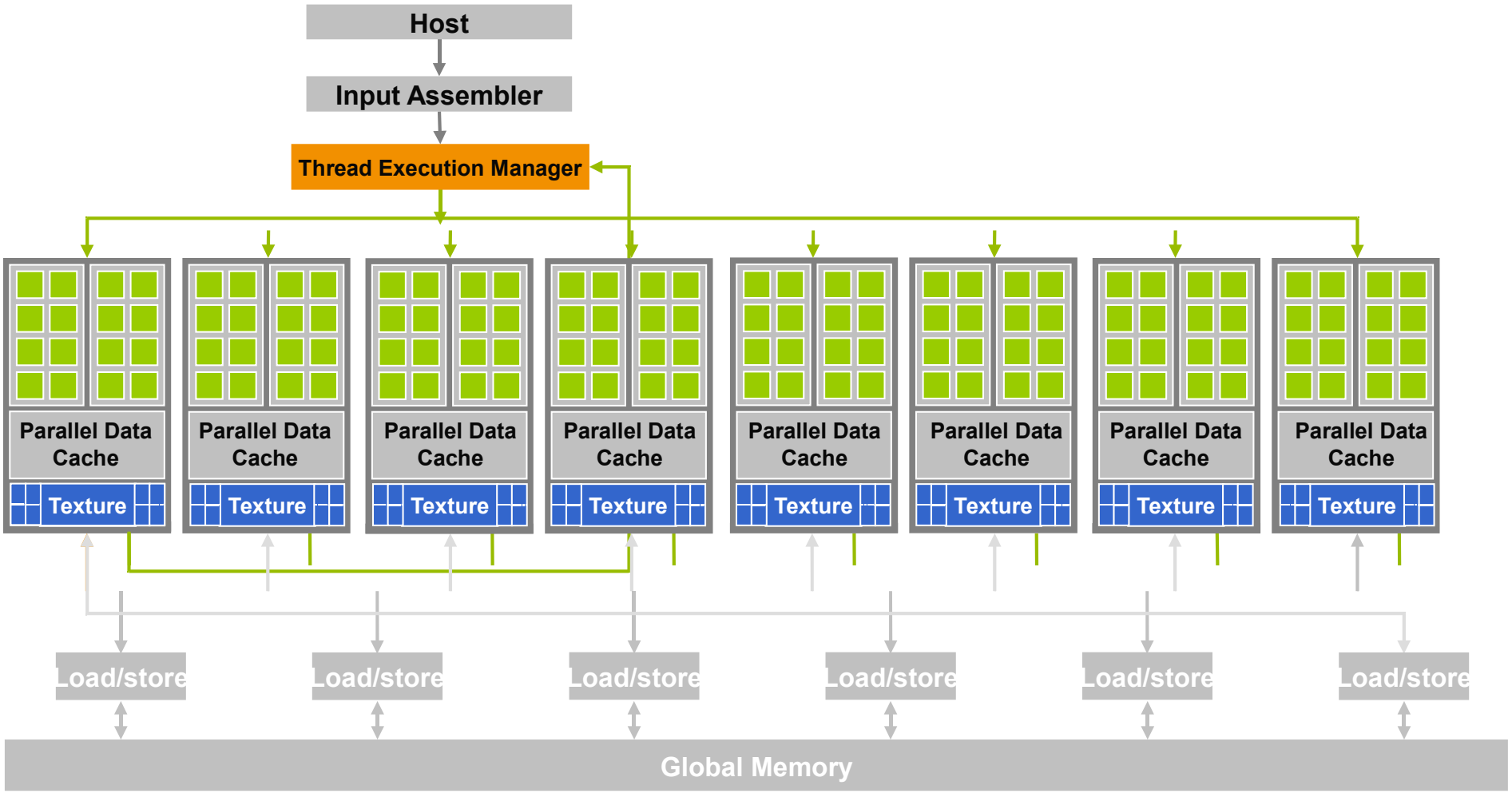
Vendor specific APIs for GPGPU:

- NVIDIA CUDA: library of functions performing computations on GPU (C, C++, Fortran), additional preprocessor with language extensions
- ATI/AMD Stream: similar functionality for ATI GPUs

Device independent schemes:

- BrookGPU (Standford University): compiler for the "Brook stream program language" with backends for different hardware; now merged with AMD Stream
- Sh (University of Waterloo): metaprogramming language for programmable GPUs
- OpenCL (Open Computing Language): open framework for parallel programming across a wide range of devices, ranging from CPUs, Cell processors and GPUs to handheld devices

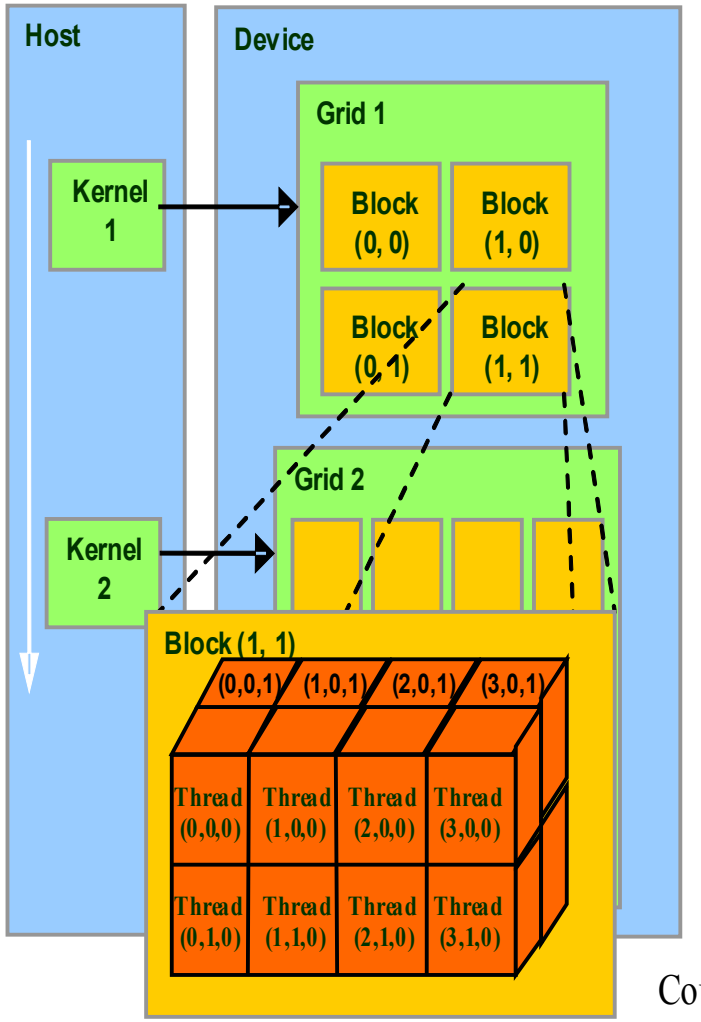
NVIDIA architecture



NVIDIA architecture

Hierarchical organization:

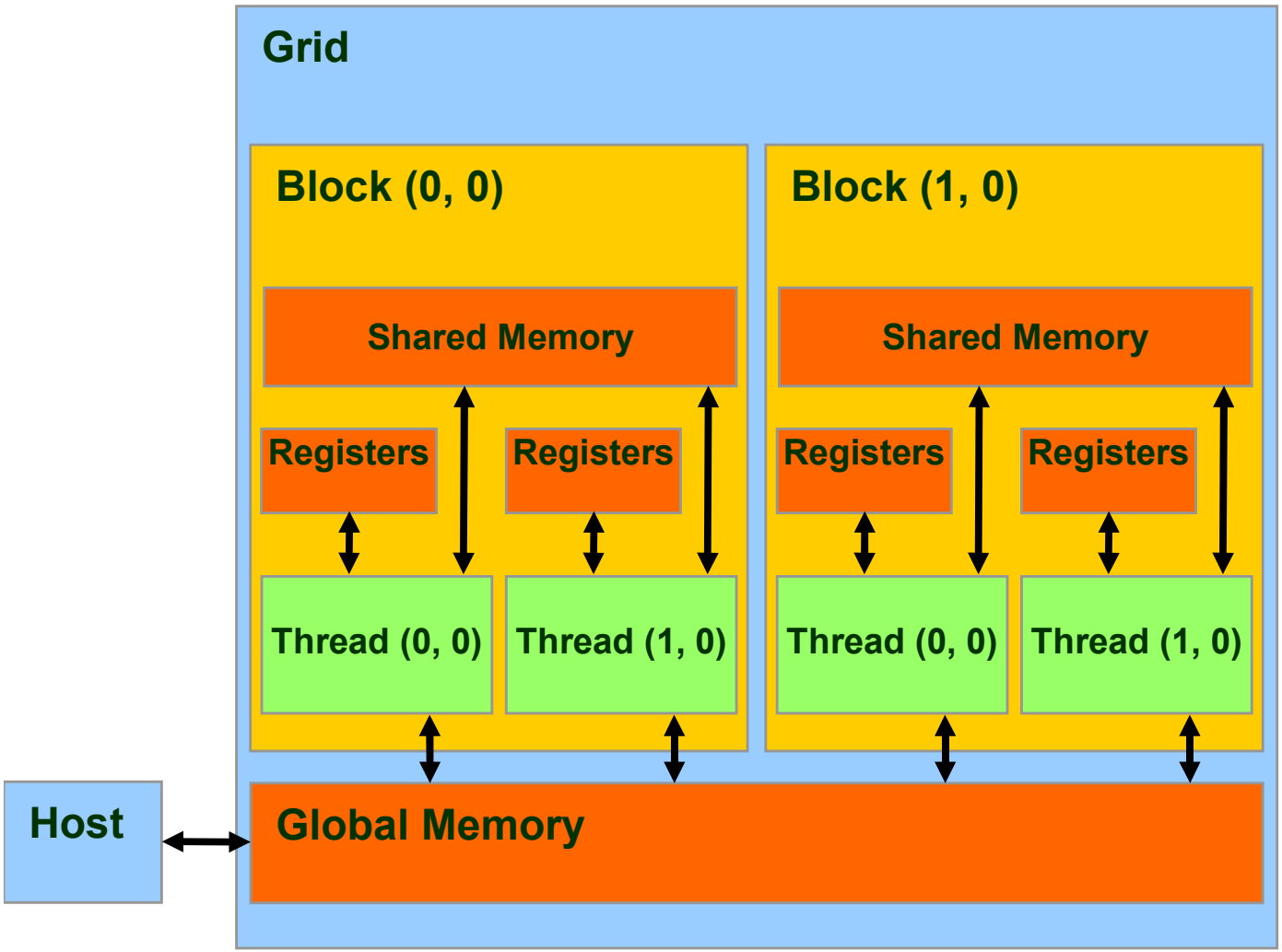
- host code ↔ device code
- grid of blocks are scheduled independently
- each block consists of threads that can synchronize



Courtesy: NDVIA

NVIDIA architecture

Memory layout:



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

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

Design goals

Computations need to be organized to suit the GPU layout for maximum performance:

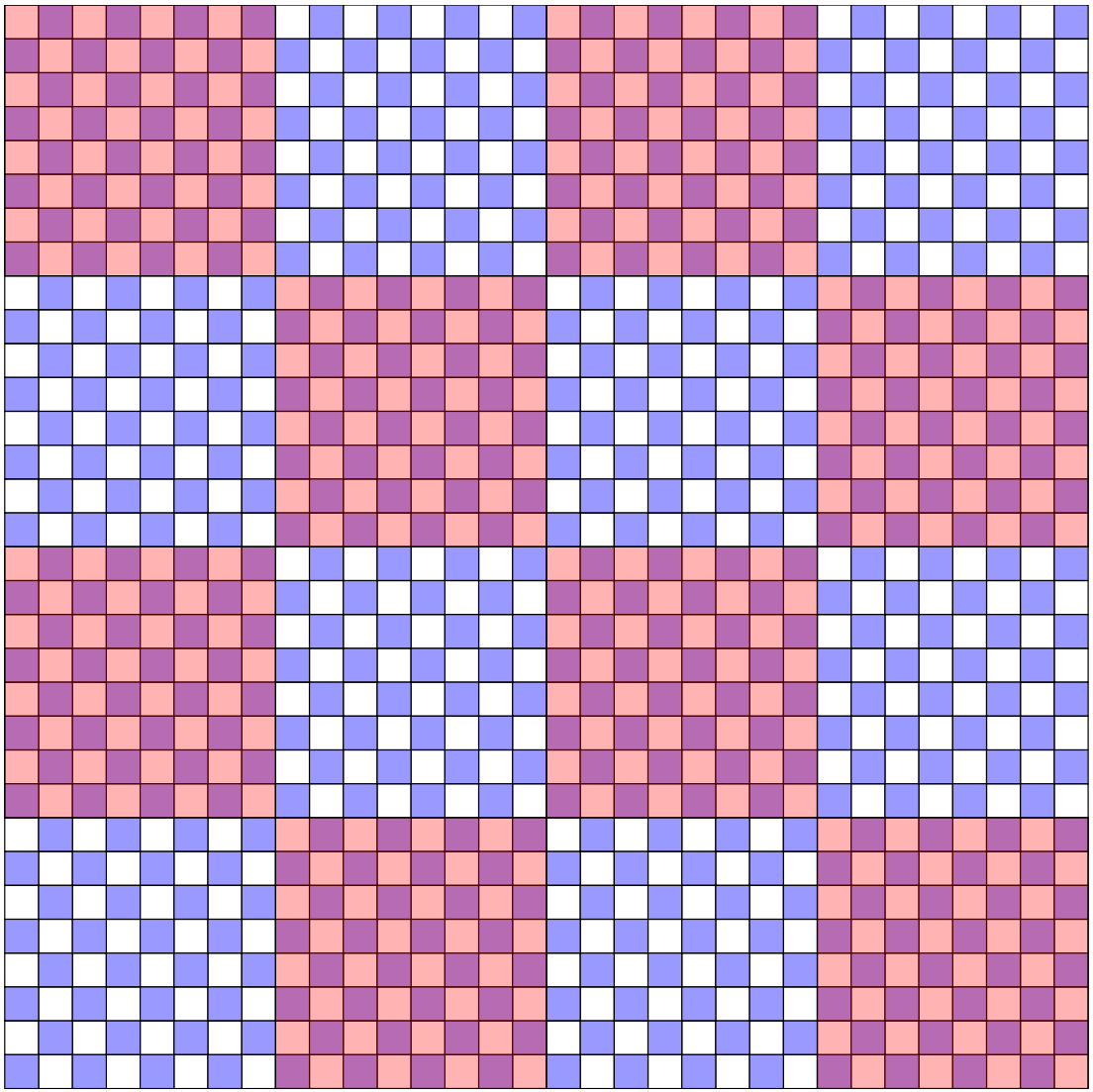
- many threads/blocks required to alleviate memory latencies
- per block shared memory much faster than global memory $\Rightarrow$ define sub-problems that fit in to the (currently 16KB) shared memory
- threads within a block can synchronize and reside on the same processor, blocks must be independent (arbitrary execution order)

Consequences for (Metropolis) simulations:

- best to use an independent RNG per thread $\Rightarrow$ need to make sure that sequences are uncorrelated
- divide system into tiles that can be worked on independently $\Rightarrow$ level-1 checkerboard
- each tile should fit into shared memory
- divide tile (again) in checkboard fashion for parallel update with different threads $\Rightarrow$ level-2 checkerboard

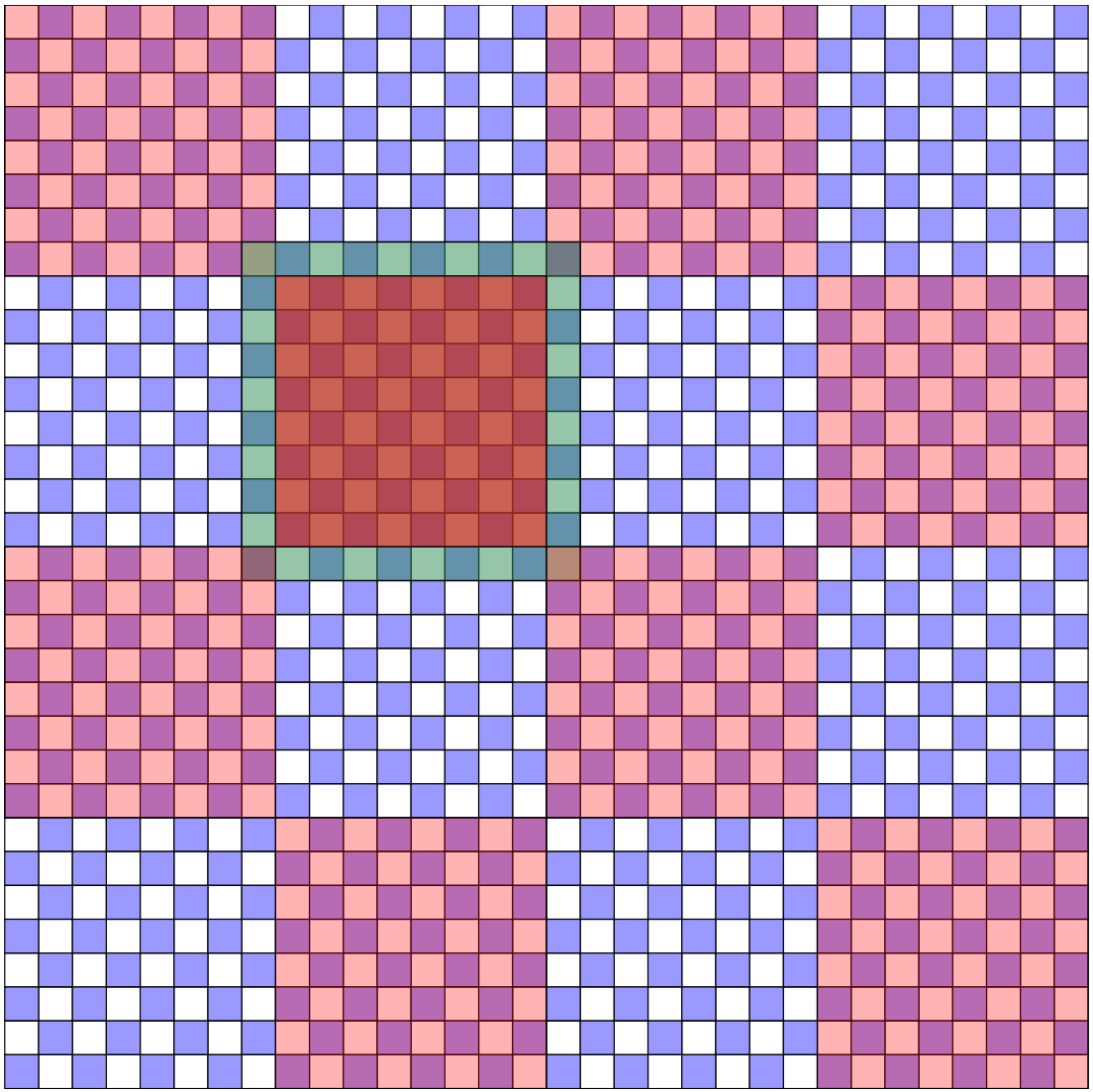
Checkerboard decomposition

- (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



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



Implementation

```
__global__ void metro_sublattice_shared(spin_t *s, int *ranvec, int offset)
{
  int n = threadIdx.y*(BLOCKL/2)+threadIdx.x;

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

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

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

Implementation

```
  __shared__ int ranvecS[THREADS];
  ranvecS[n] = ranvec[(blockIdx.y*(GRIDL/2)+blockIdx.x)*THREADS+n];

  __syncthreads();

  int x1 = (threadIdx.y%2)+2*threadIdx.x;
  int x2 = ((threadIdx.y+1)%2)+2*threadIdx.x;
  int y = threadIdx.y;

  for(int i = 0; i < SWEEPS_LOCAL; ++i) {
    int ide = sS(x1,y)*(sS(x1-1,y)+sS(x1+1,y)+sS(x1,y-1)+sS(x1,y+1));
    if(ide <= 0 || RAN(ranvecS[n]) < boltzD[ide]) sS(x1,y) = -sS(x1,y);

    __syncthreads();

    ide = sS(x2,y)*(sS(x2-1,y)+sS(x2+1,y)+sS(x2,y-1)+sS(x2,y+1));
    if(ide <= 0 || RAN(ranvecS[n]) < boltzD[ide]) sS(x2,y) = -sS(x2,y);

    __syncthreads();
  }

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

Performance

How to assess performance?

- what to compare to (one CPU core, whole CPU, SMP system, ...)
- 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

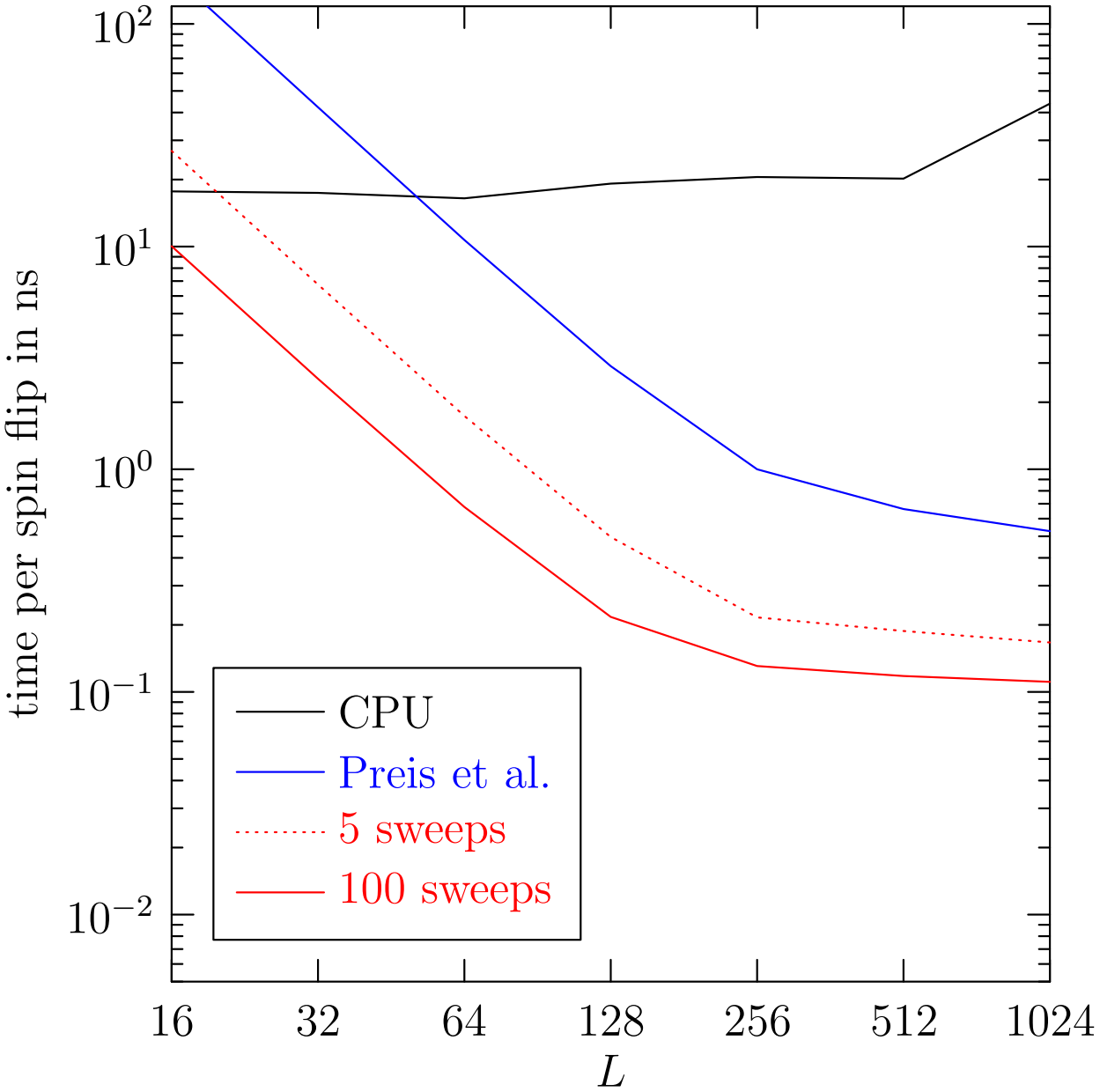
How to assess performance?

- what to compare to (one CPU core, whole CPU, SMP system, ...)
- 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

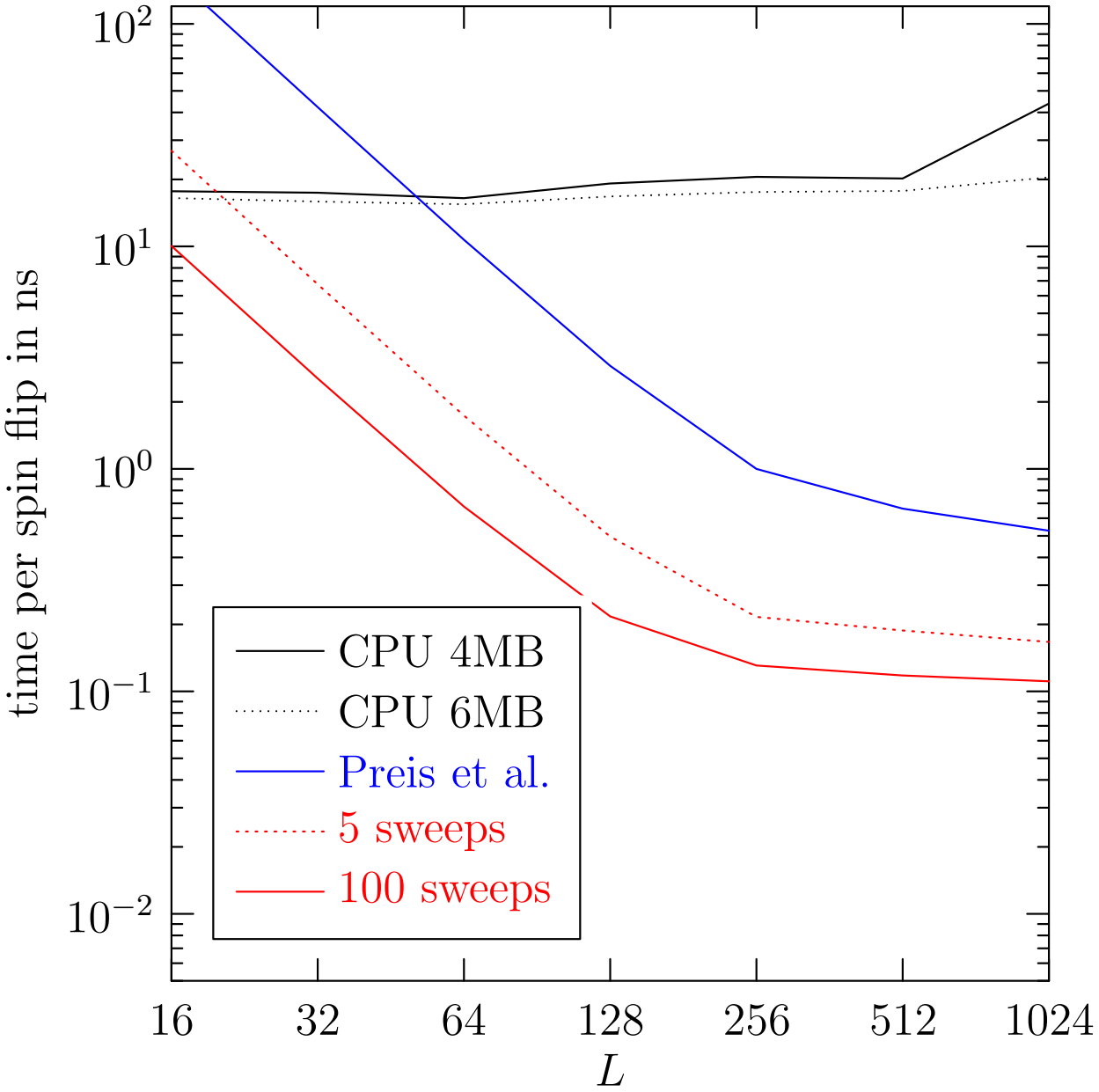
Example: Metropolis simulation of 2D Ising system

- use Knuth's linear congruential RNG
- no neighbor table since integer multiplies and adds are very cheap (4 instructions per clock cycle and processor)
- need to play with tile sizes to achieve best throughput

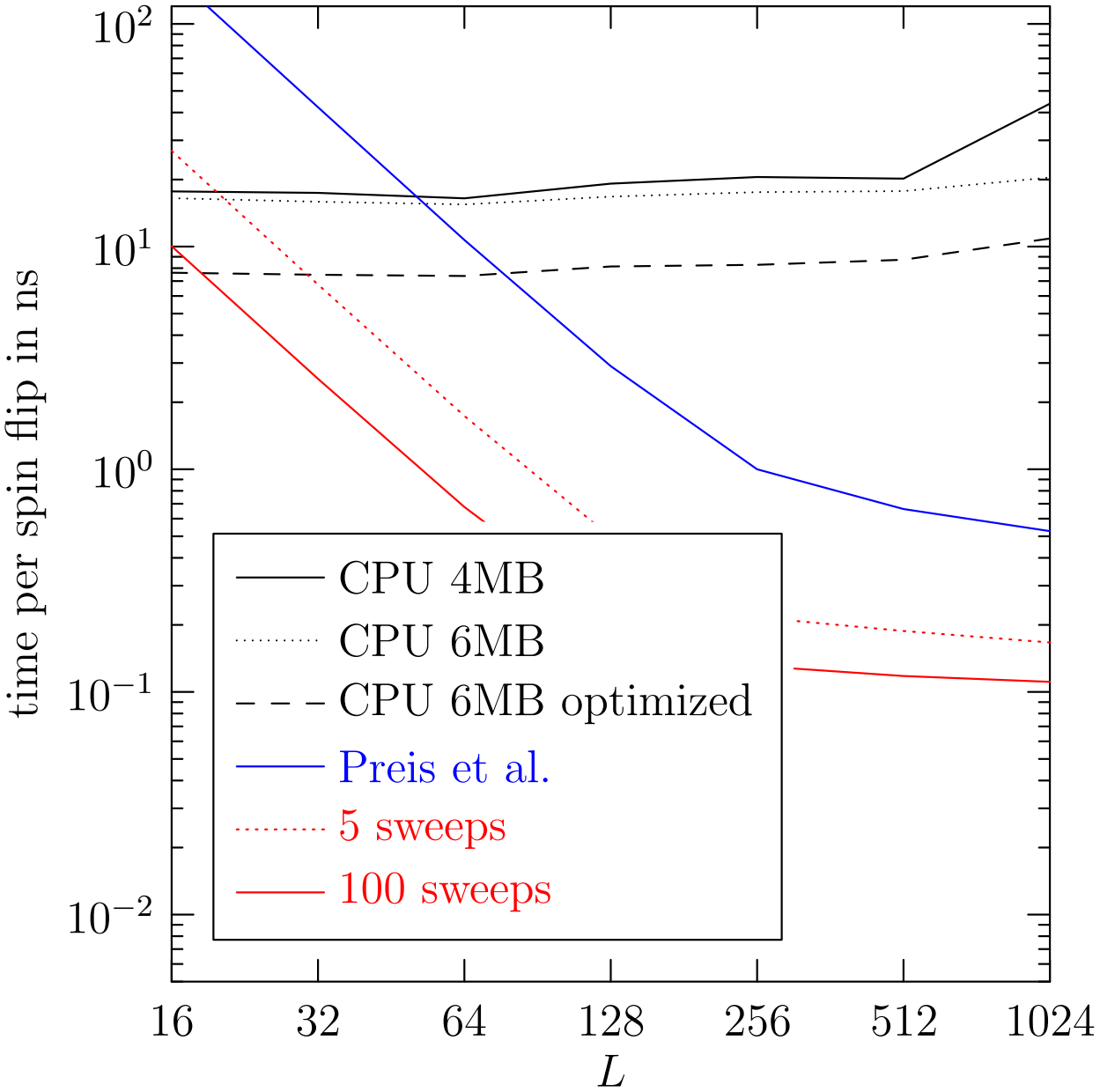
2D Ising ferromagnet



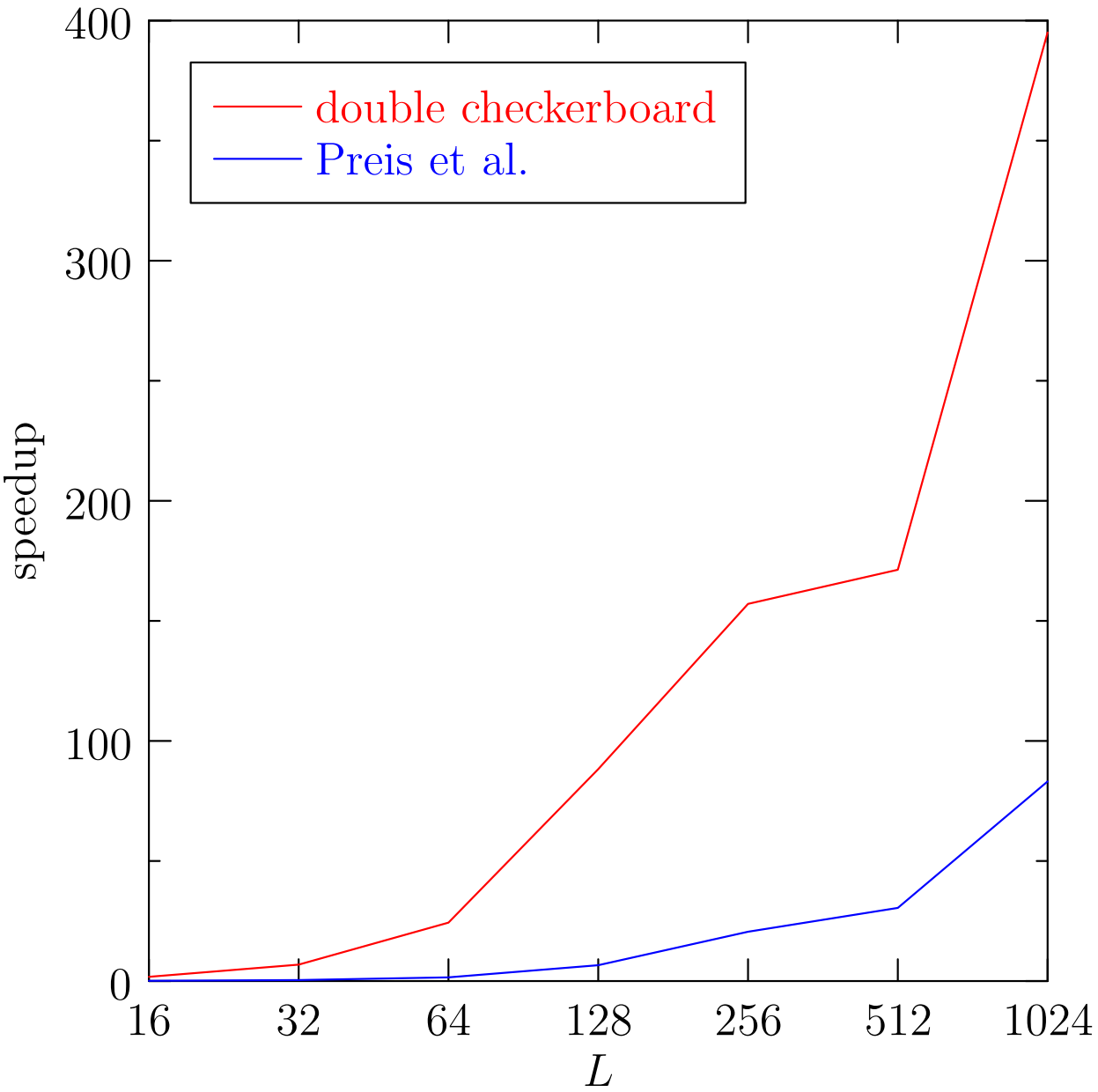
2D Ising ferromagnet



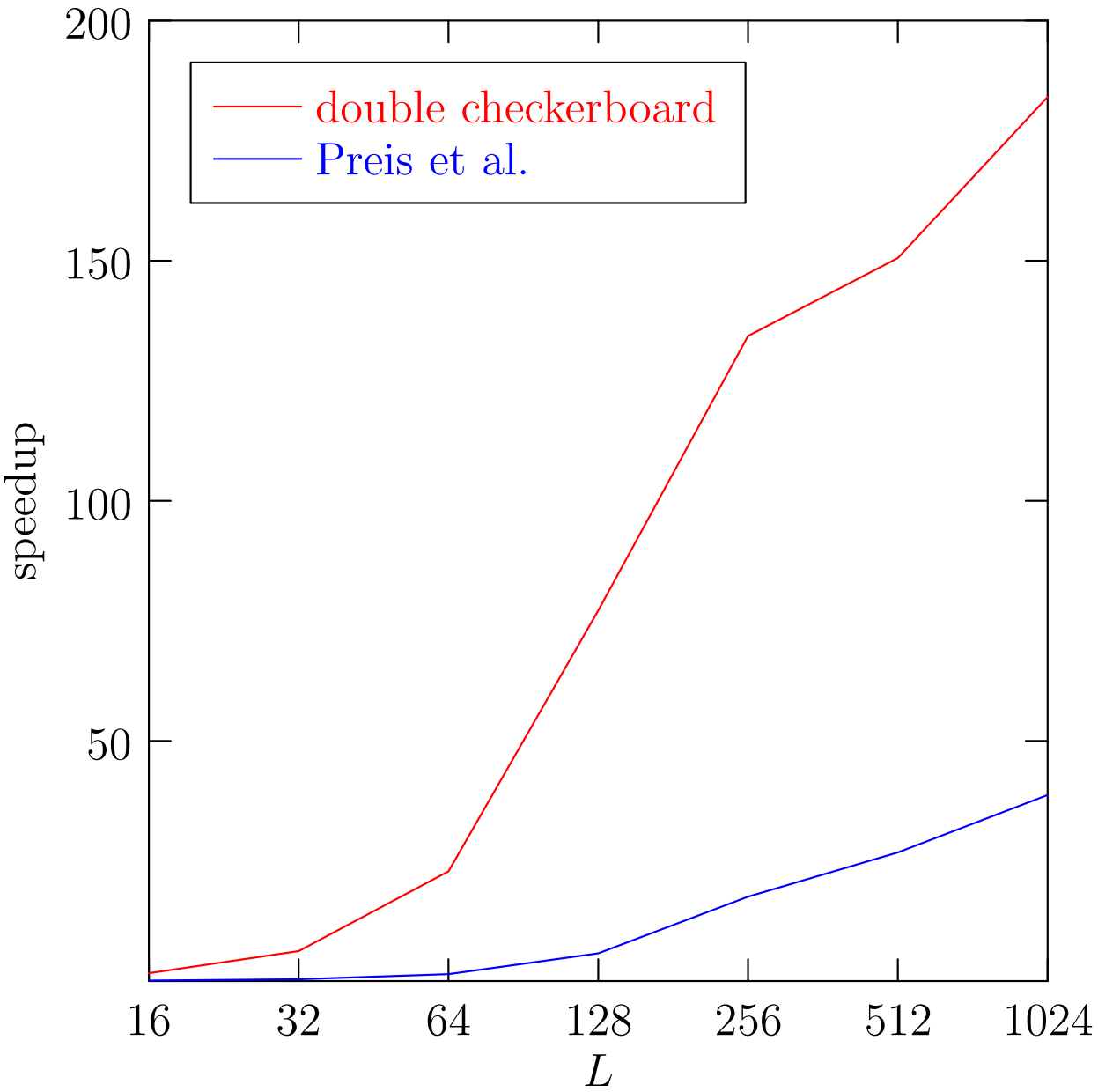
2D Ising ferromagnet



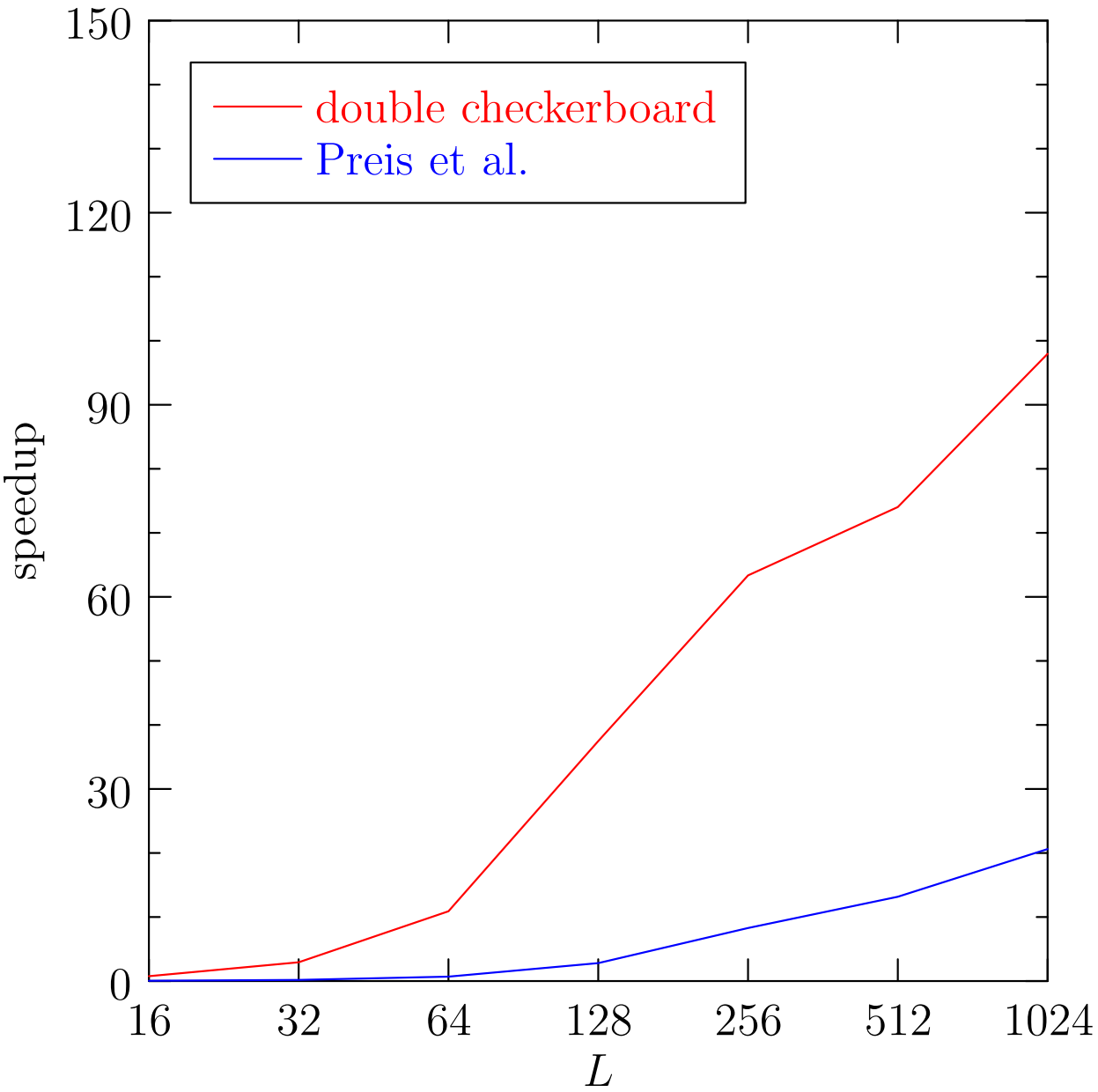
2D Ising ferromagnet



2D Ising ferromagnet



2D Ising ferromagnet



Heisenberg model

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

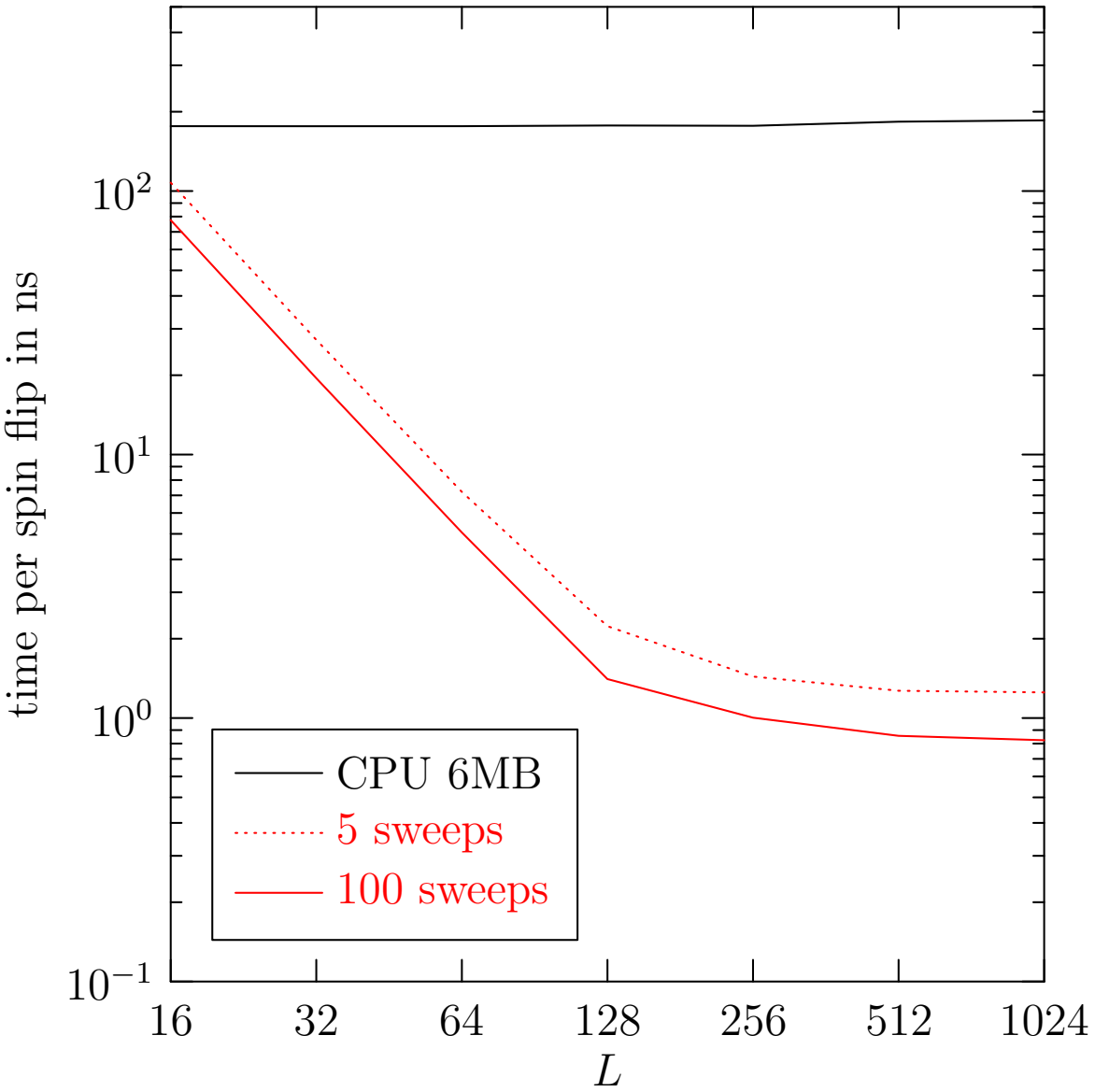
Heisenberg model

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

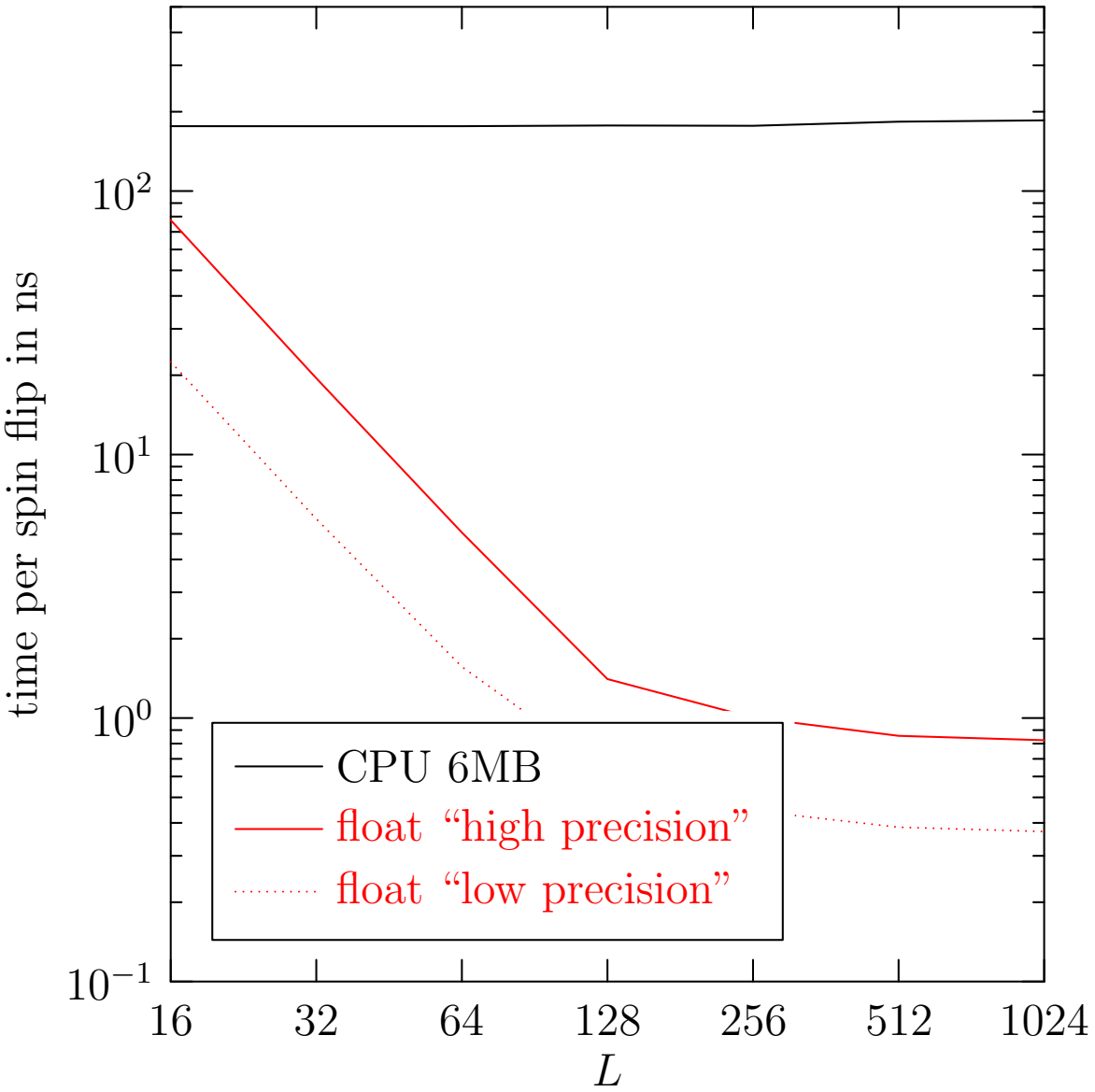
$\Rightarrow$ use same decomposition, but now floating-point computations are dominant:

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

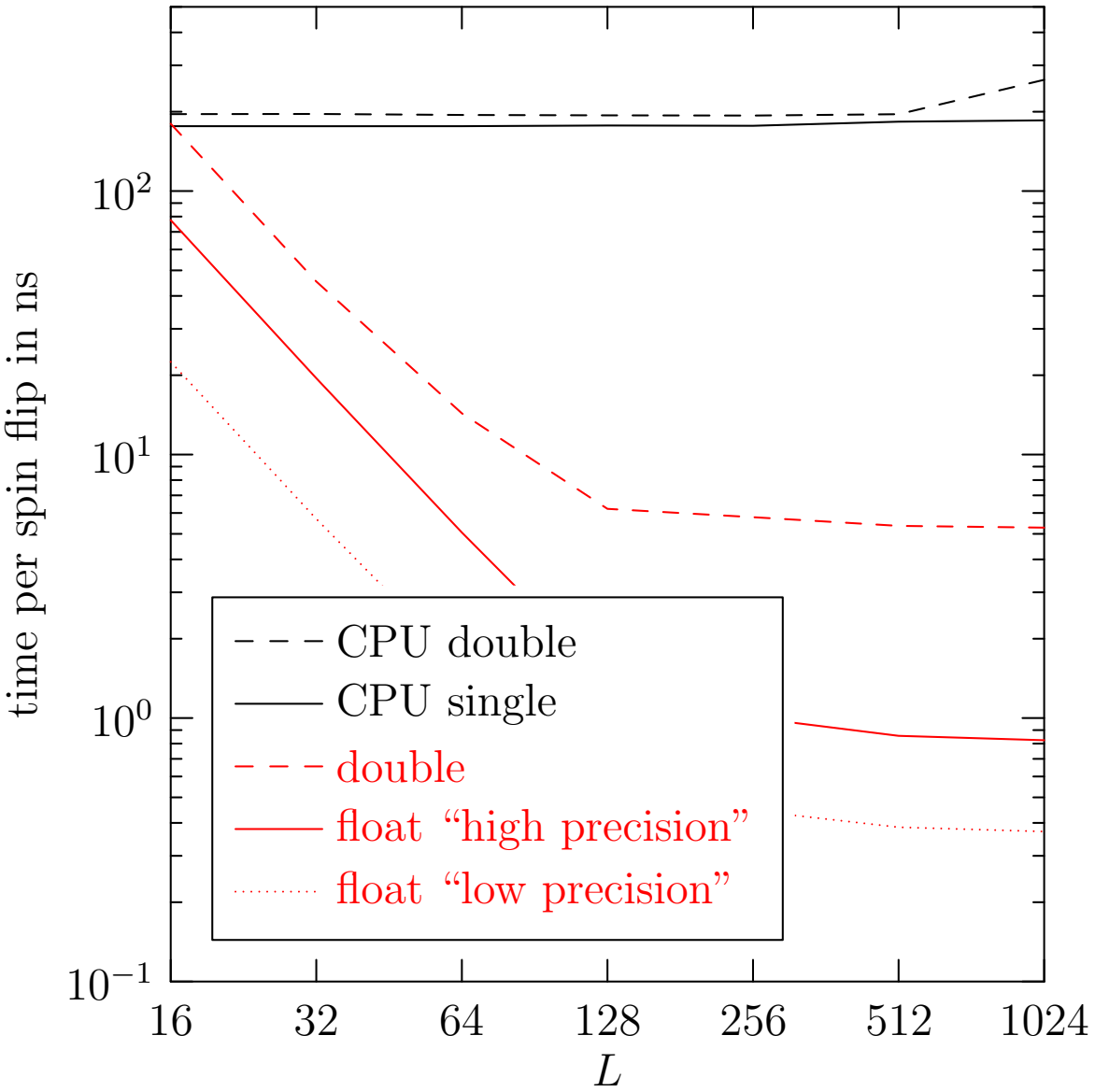
Heisenberg model: performance



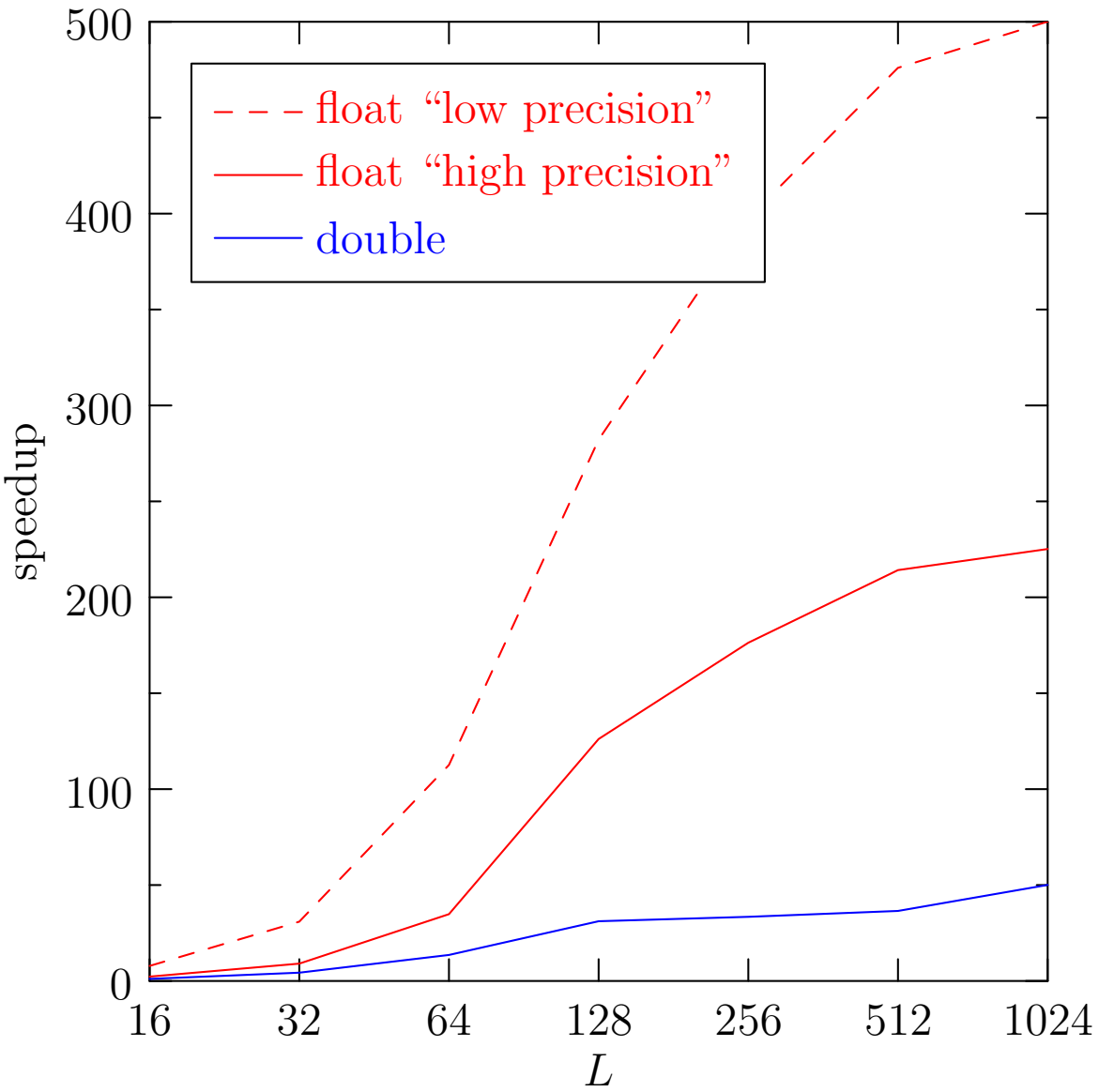
Heisenberg model: performance



Heisenberg model: performance



Heisenberg model: performance



Heisenberg model: stability

Performance results:

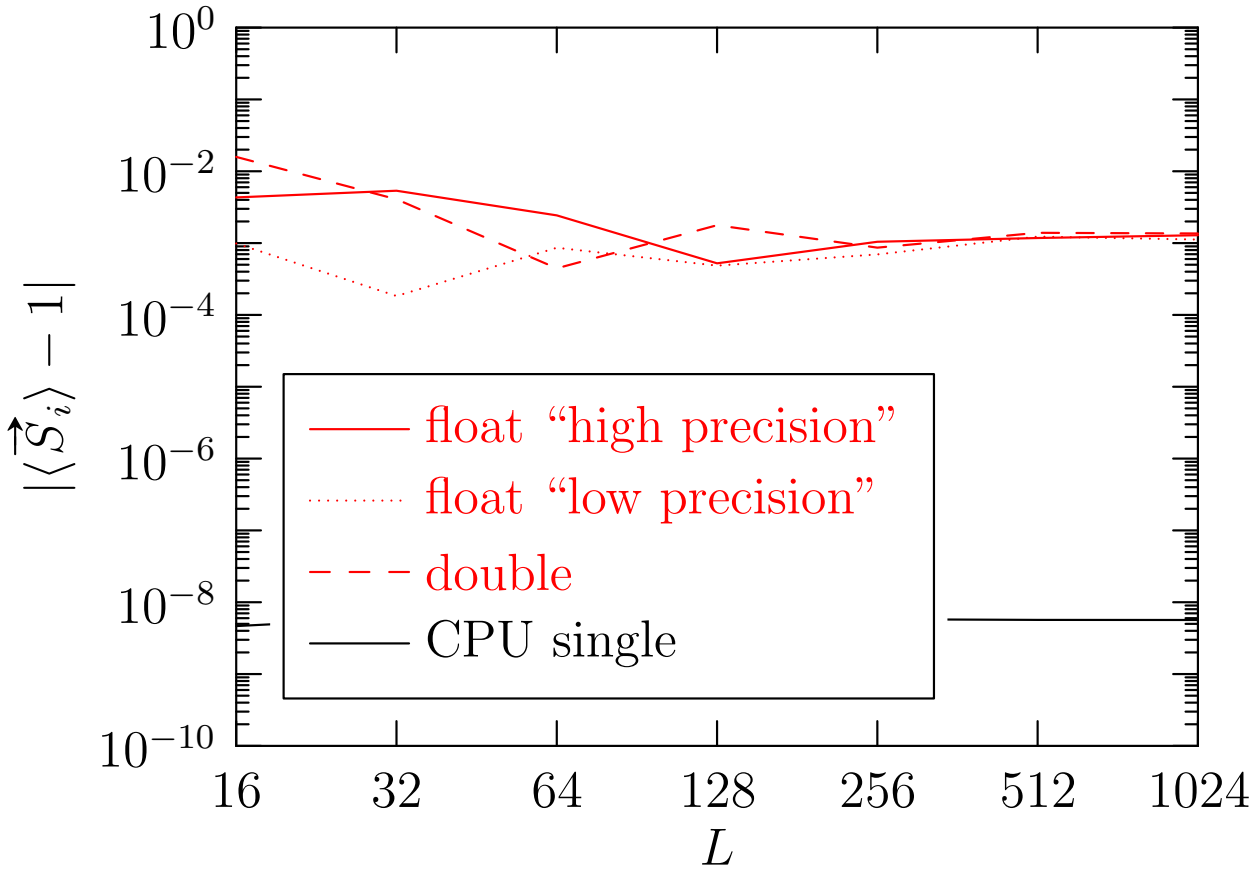
- CPU: 185 ns (single) resp. 264 (double) per spin flip
- GPU: 0.8 ns (single), 0.4 ns (fast single) resp. 5.3 ns (double) per spin flip

Heisenberg model: stability

Performance results:

- CPU: 185 ns (single) resp. 264 (double) per spin flip
- GPU: 0.8 ns (single), 0.4 ns (fast single) resp. 5.3 ns (double) per spin flip

How about stability?



Heisenberg model: stability

Performance results:

- CPU: 185 ns (single) resp. 264 (double) per spin flip
- GPU: 0.8 ns (single), 0.4 ns (fast single) resp. 5.3 ns (double) per spin flip

How about stability?

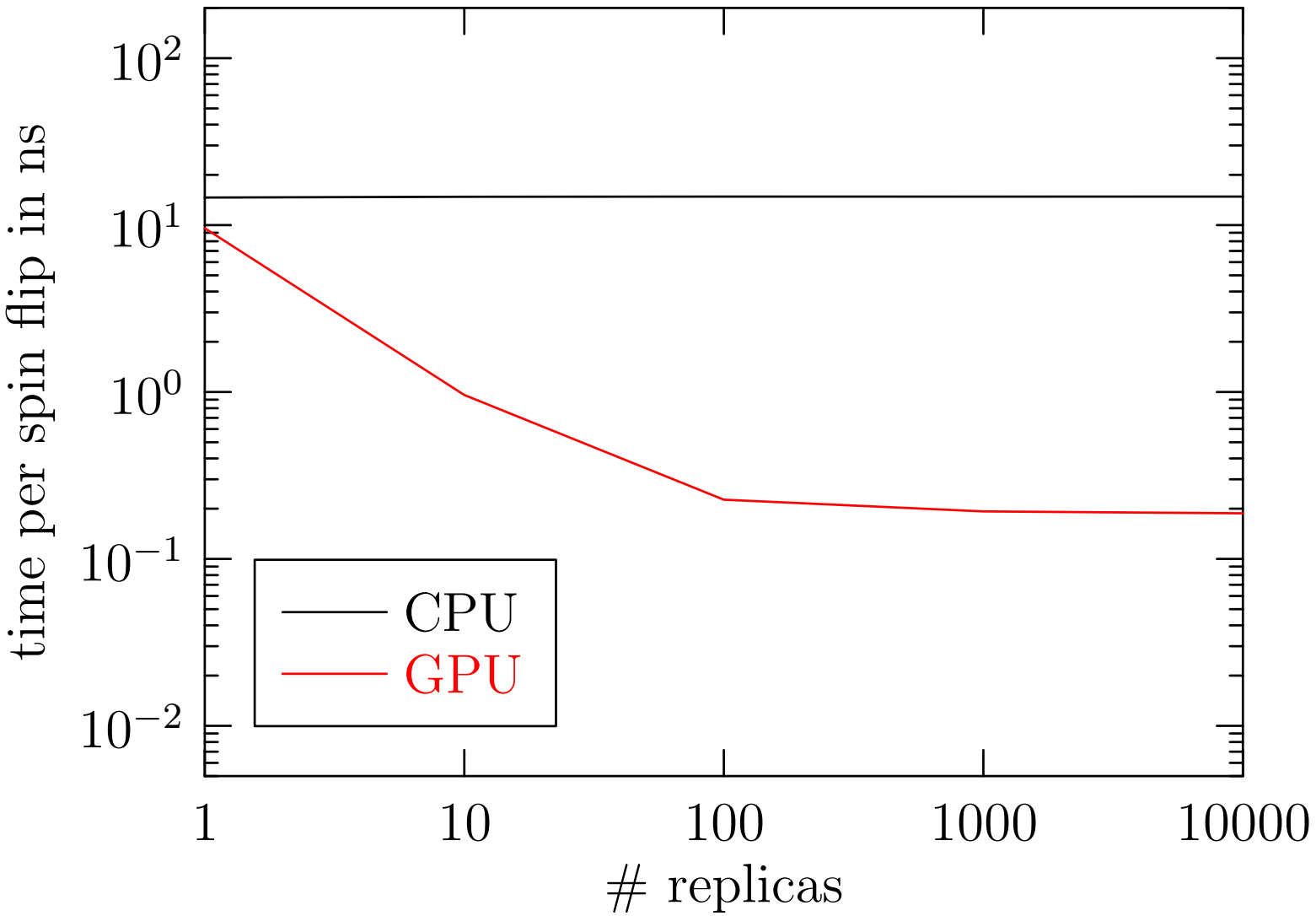
- no drift of spin normalization
- no deviations in averages from reference implementation (at least at low precision)
- more subtle effects: non-uniform trial vectors etc.
- needs closer look

Spin glasses

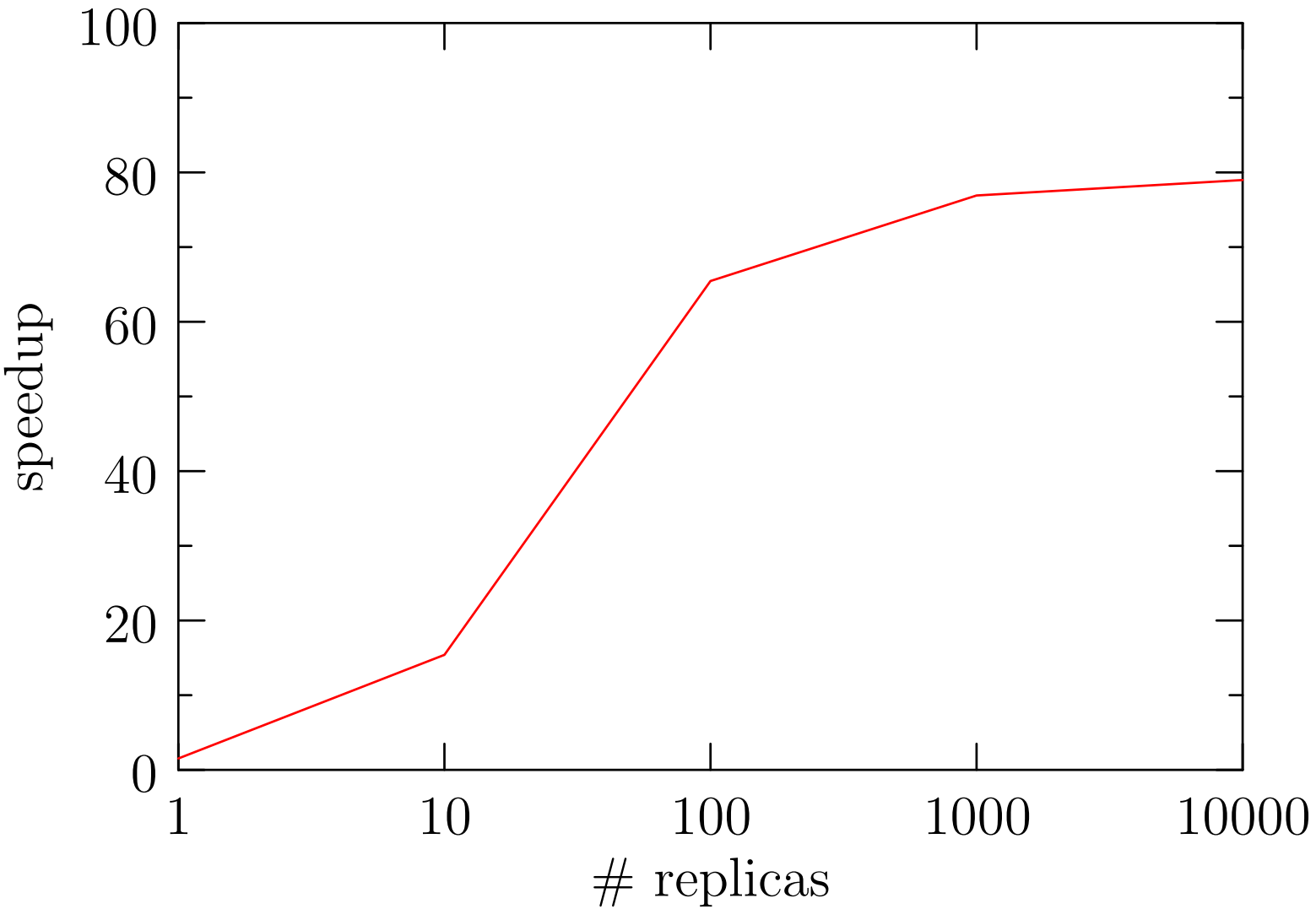
Simulate Edwards-Anderson model on GPU:

- 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

Spin glass: performance



Spin glass: performance



Spin glasses: continued

Seems to work well with

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

but not better than ferromagnetic Ising model.

Spin glasses: continued

Seems to work well with

- 15 ns per spin flip on CPU
- 180 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

Implementation

```
for(int i = 0; i < SWEEPS_LOCAL; ++i) {
  float r = RAN(ranvecS[n]);
  if(r < boltzD[4]) sS(x1,y) = ~sS(x1,y);
  else {
    p1 = JSx(x1m,y) ^ sS(x1,y) ^ sS(x1m,y); p2 = JSx(x1,y) ^ sS(x1,y) ^ sS(x1p,y);
    p3 = JSy(x1,ym) ^ sS(x1,y) ^ sS(x1,ym); p4 = JSy(x1,y) ^ sS(x1,y) ^ sS(x1,yp);
    if(r < boltzD[2]) {
      ido = p1 | p2 | p3 | p4;
      sS(x1,y) = ido ^ sS(x1,y);
    } else {
      ido1 = p1 & p2; ido2 = p1 ^ p2;
      ido3 = p3 & p4; ido4 = p3 ^ p4;
      ido = ido1 | ido3 | (ido2 & ido4);
      sS(x1,y) = ido ^ sS(x1,y);
    }
  }

  __syncthreads();

  r = RAN(ranvecS[n]);
  if(r < boltzD[4]) sS(x2,y) = ~sS(x2,y);
  else {
    p1 = JSx(x2m,y) ^ sS(x2,y) ^ sS(x2m,y); p2 = JSx(x2,y) ^ sS(x2,y) ^ sS(x2p,y);
    p3 = JSy(x2,ym) ^ sS(x2,y) ^ sS(x2,ym); p4 = JSy(x2,y) ^ sS(x2,y) ^ sS(x2,yp);
    if(r < boltzD[2]) {
      ido = p1 | p2 | p3 | p4;
      sS(x2,y) = ido ^ sS(x2,y);
    } else {
      ido1 = p1 & p2; ido2 = p1 ^ p2;
      ido3 = p3 & p4; ido4 = p3 ^ p4;
      ido = ido1 | ido3 | (ido2 & ido4);
      sS(x2,y) = ido ^ sS(x2,y);
    }
  }

  __syncthreads();
}
```

Spin glasses: continued

Seems to work well with

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

but not better than ferromagnetic Ising model.

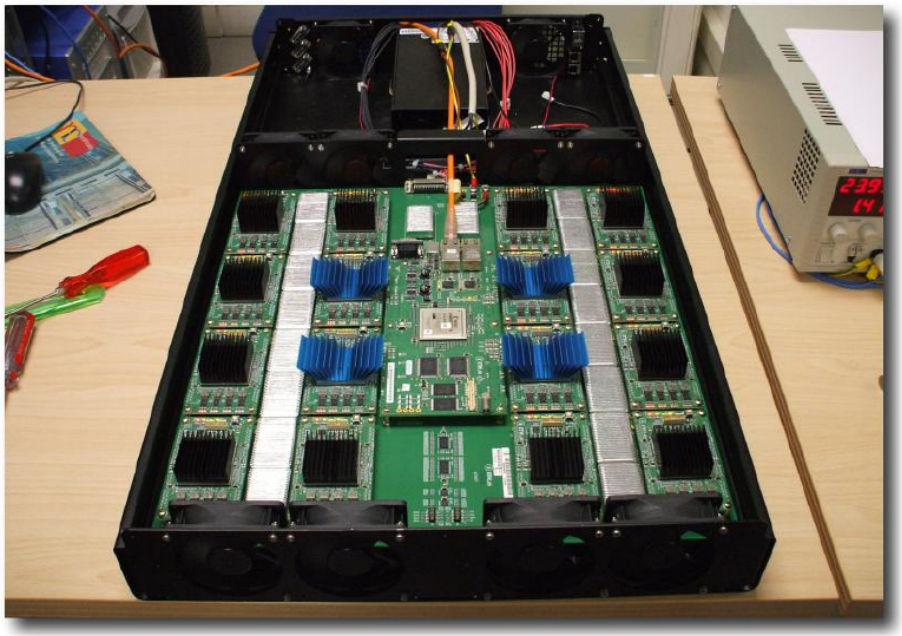
Further improvement: use multi-spin coding

- 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 15 ps per spin flip

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×	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×

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×	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

Outlook

Conclusions:

- GPGPU promises significant speedups at moderate coding effort
- need to find appropriate domain decomposition to leverage shared memory efficiency
- especially promising for problems with continous variables, but need to study precision effects
- effort significantly smaller than for (design and programming of) special-purpose machines
- GPGPU might be a fashion, but CPU computing goes the same way

Outlook:

- cluster algorithm is possible and looks promising
- parallel tempering no problem
- generalized-ensemble techniques might be more of a challenge