Performance potential for simulating spin models on GPU

Martin Weigel

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

11th International NTZ-Workshop on New Developments in Computational Physics
Leipzig, November 25–27, 2010

Emmy Noether-Programm
Deutsche Forschungsgemeinschaft
DFG

Junior Research Group
"Complex Systems with Frustration"

univer
sität
mainz

GPU computation frameworks

GPGPU = General Purpose Computation on Graphics Processing Unit

“Old” times: use original graphics primitives

- OpenGL
- DirectX

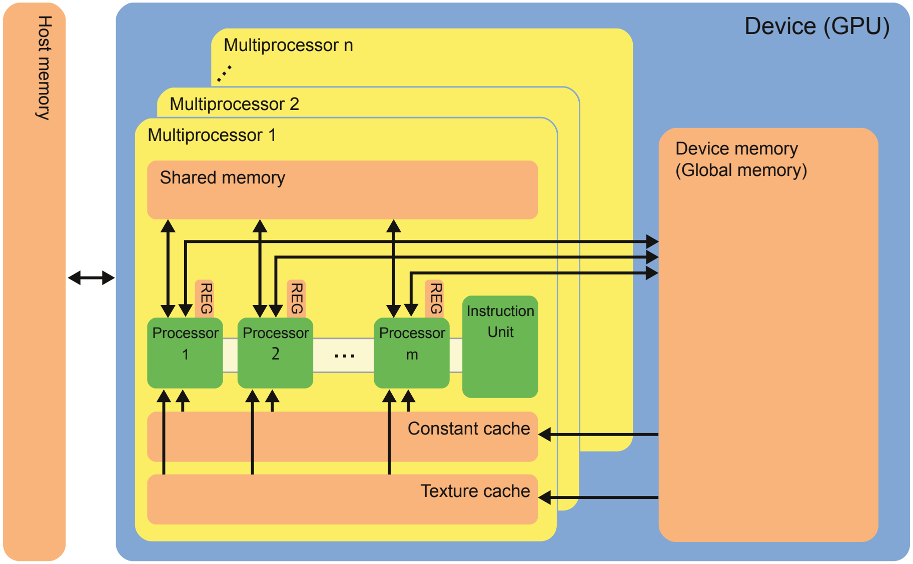
Vendor specific APIs for GPGPU:

- 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

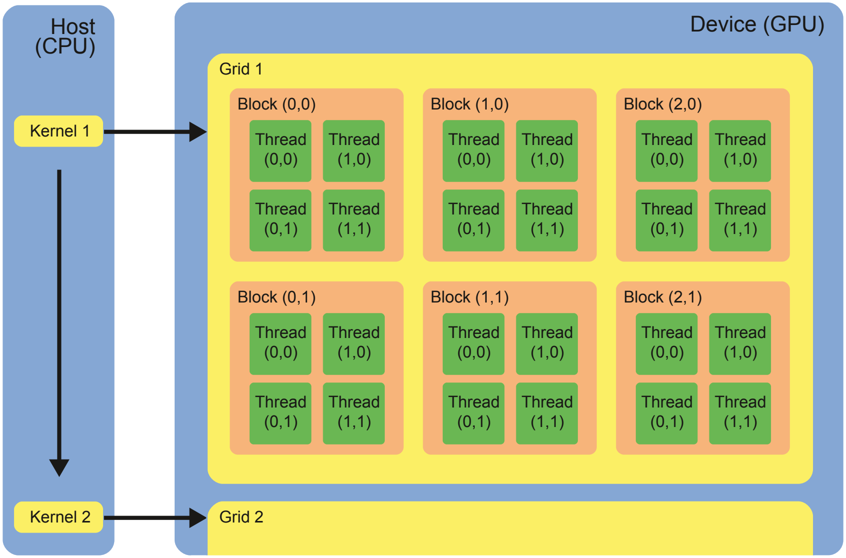
Device independent schemes:

- 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



NVIDIA architecture

Memory layout:

- *Registers*: each multiprocessor is equipped with several thousand registers with local, zero-latency access
- *Shared memory*: processors of a multiprocessor have access a small amount (16 KB for Tesla, 48 KB for Fermi) of on chip, very small latency shared memory
- *Global memory*: large amount (currently up to 4 GB) of memory on separate DRAM chips with access from every thread on each multiprocessor with a latency of several hundred clock cycles
- *Constant and texture memory*: read-only memories of the same speed as global memory, but cached
- *Host memory*: cannot be accessed from inside GPU functions, relatively slow transfers

Spin models

Consider classical spin models with nn interactions, in particular

Ising model

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

Heisenberg model

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

Edwards-Anderson spin glass

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

Metropolis simulations

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

- 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

Metropolis simulations

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

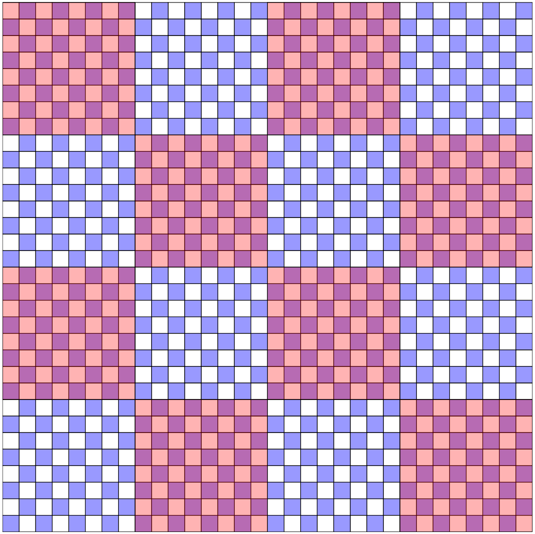
- a large degree of locality of the calculations, reducing the need for communication between threads
- a large coherence of calculations with a minimum occurrence of divergence of the execution paths of different threads
- a total number of threads significantly exceeding the number of available processing units
- a large overhead of arithmetic operations and shared memory accesses over global memory accesses

Consequences for (Metropolis) simulations:

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

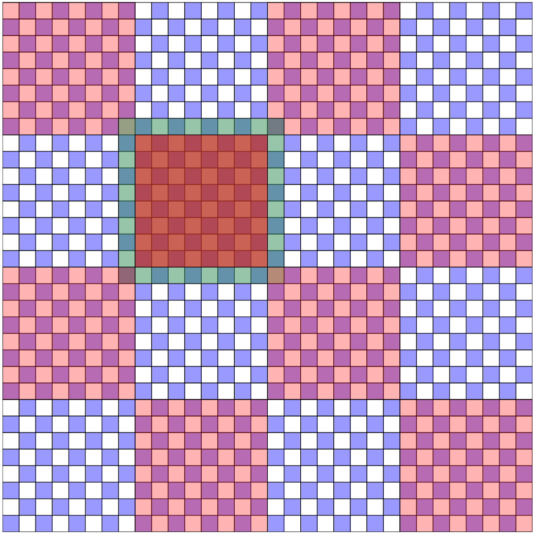
Checkerboard decomposition

- (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



Performance

How to assess performance?

- what to compare to (one CPU core, whole CPU, SMP system, ...)
  here: Tesla C1060 vs. Intel QuadCore (Yorkfield) @ 3.0 GHz/6 MB
- for really fair comparison: optimize CPU code for cache alignment, use SSE instructions etc.
- ignore measurements, since spin flips per μs, (ns, ps) is well-established unit for spin systems

Performance

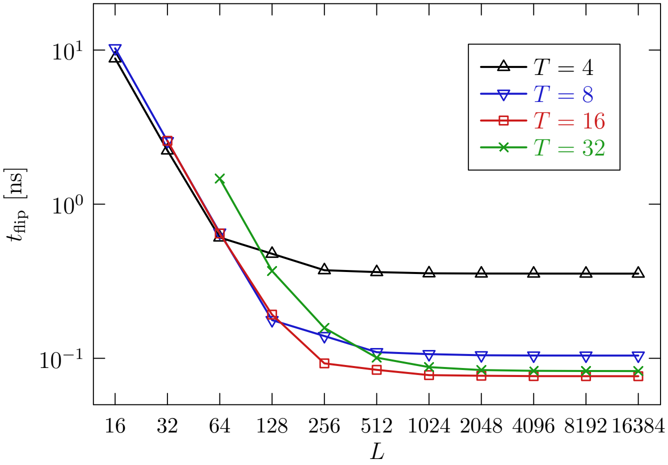
How to assess performance?

- what to compare to (one CPU core, whole CPU, SMP system, ...)
  here: Tesla C1060 vs. Intel QuadCore (Yorkfield) @ 3.0 GHz/6 MB
- for really fair comparison: optimize CPU code for cache alignment, use SSE instructions etc.
- ignore measurements, since spin flips per μs, (ns, ps) is well-established unit for spin systems

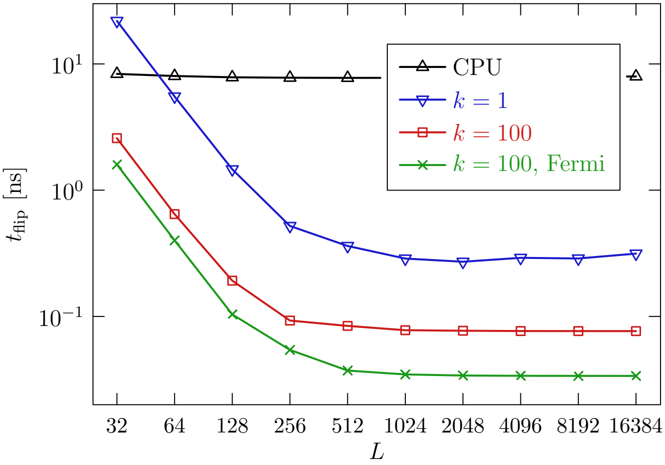
Example: Metropolis simulation of 2D Ising system

- use 32-bit linear congruential generator
- 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

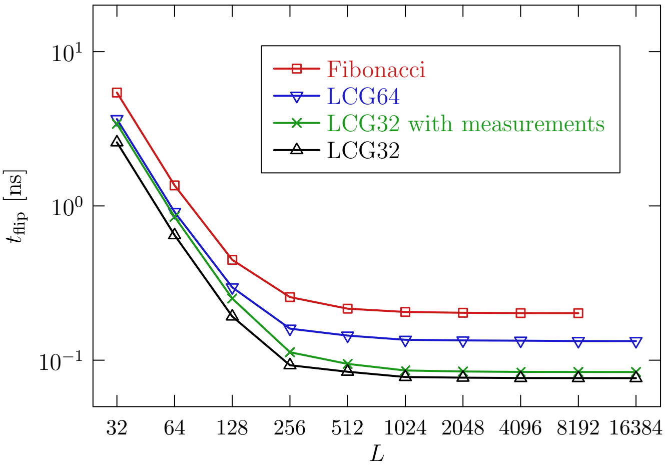
2D Ising ferromagnet



2D Ising ferromagnet

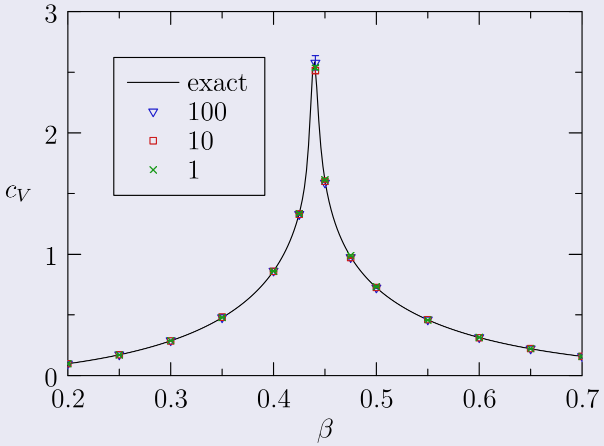


2D Ising ferromagnet



A closer look

Comparison to exact results:



A closer look

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

- checkerboard update uses random numbers in different way than sequential update
- linear congruential generators can skip ahead: “right” way uses non-overlapping sub-sequences
- “wrong” way uses sequences from random initial seeds, many of which must overlap

A closer look

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

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

A closer look

Speedups:

- In two dimensions:
  - 0.076 ns on GPU vs. 8 ns on CPU: factor **105**
  - 0.034 ns on Fermi GPU: factor **235**
  - CPU code up to 10 times faster, GPU code up to 9 times faster than that used in

    `T. Preis, P. Virnau, W. Paul, J. J. Schneider, J. Comput. Phys. 228, 4468 (2009)`
- In three dimensions:
  - 0.13 ns vs. 14 ns on CPU: factor **110**
  - 0.067 ns on Fermi GPU: factor **210**

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?

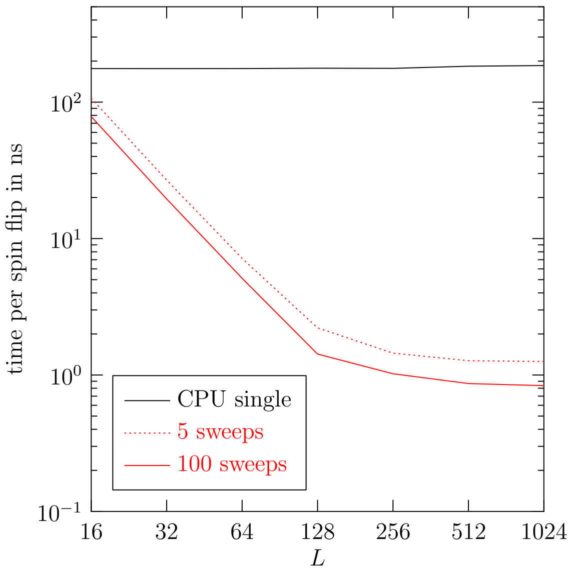
Heisenberg model

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?

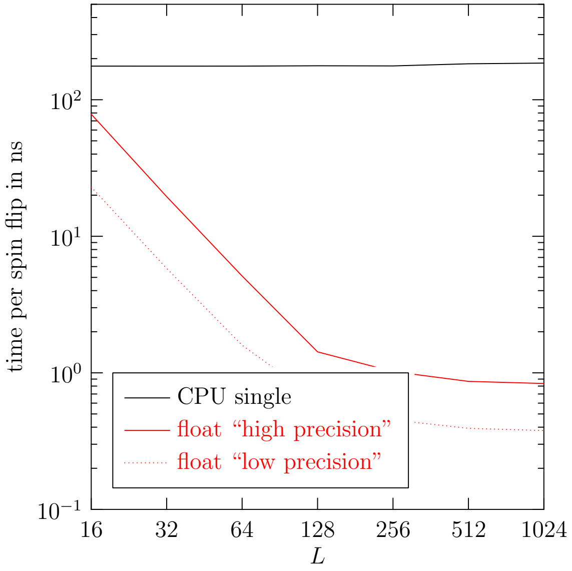
$\Rightarrow$ use same decomposition, but now floating-point computations are dominant:

- 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

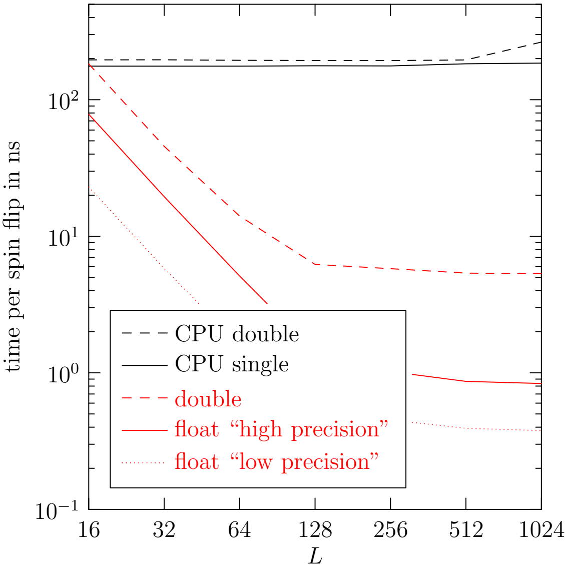
Heisenberg model: performance



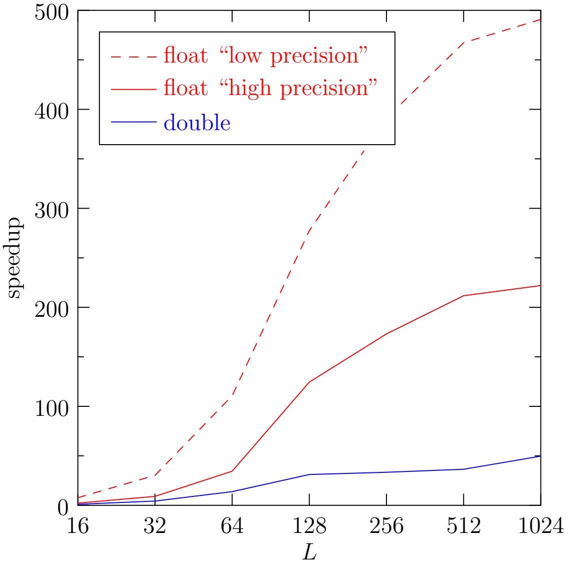
Heisenberg model: performance



Heisenberg model: performance



Heisenberg model: performance



Heisenberg model: stability

Performance results:

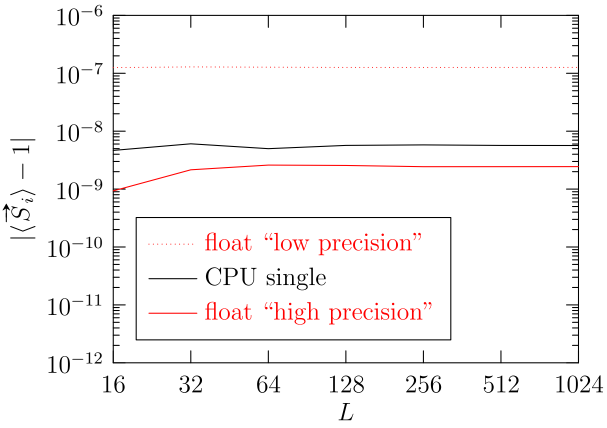
- CPU: 185 ns (single) resp. 264 (double) per spin flip
- GPU: 0.8 ns (single), 0.4 ns (fast single) resp. 5.3 ns (double) per spin flip

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?

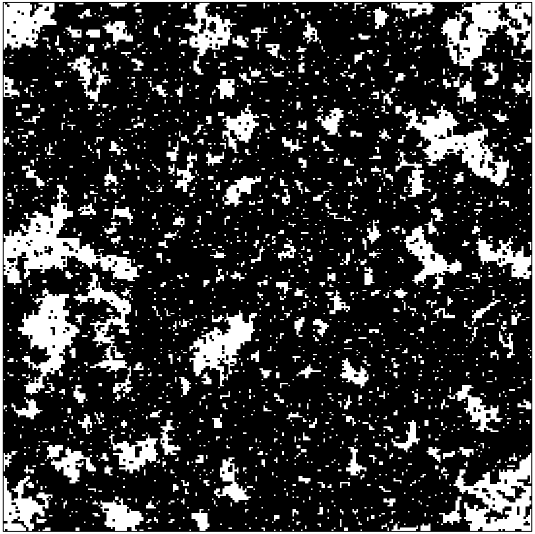


Cluster algorithms

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.

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.

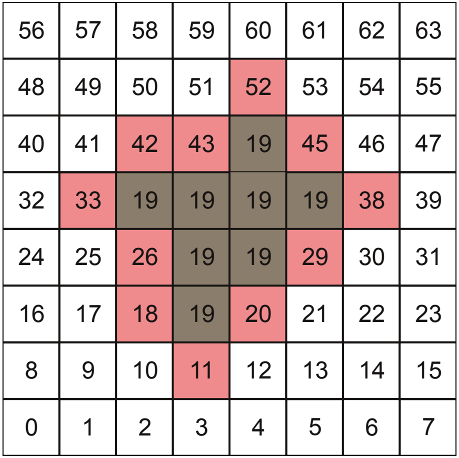
labeling *inside* of domains

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

relabeling *across* domains

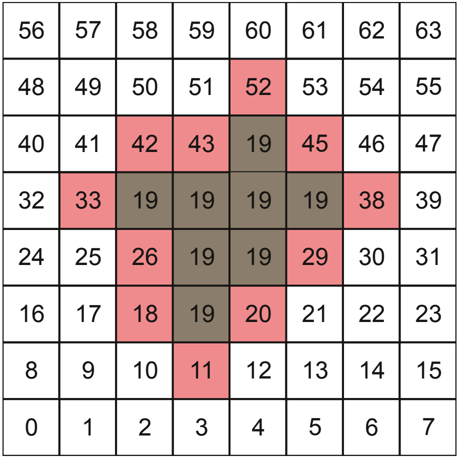
- self-labeling
- hierarchical approach
- iterative relaxation

BFS or Ants in the Labyrinth



56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	19	45	46	47
32	33	19	19	19	19	38	39
24	25	26	19	19	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

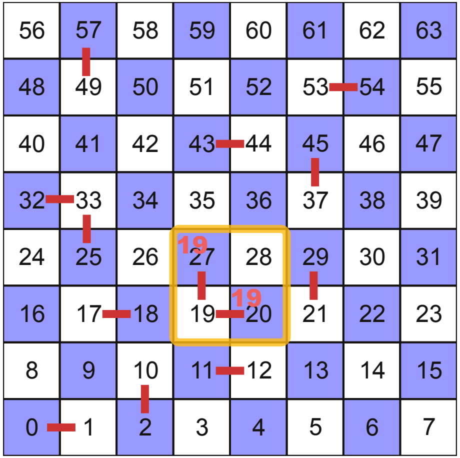
BFS or Ants in the Labyrinth

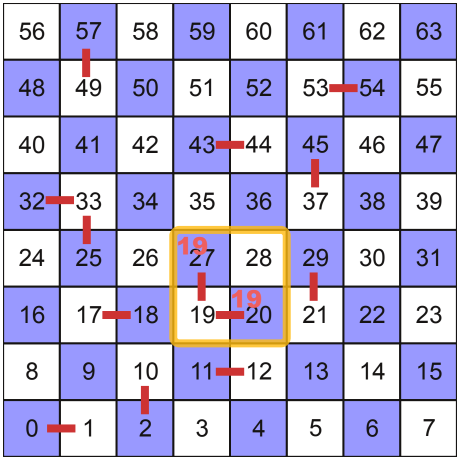


56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	19	45	46	47
32	33	19	19	19	19	38	39
24	25	26	19	19	29	30	31
16	17	18	19	20	21	22	23
8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7

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

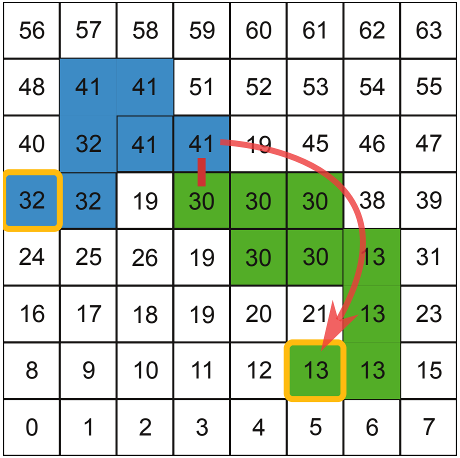
Self-labeling





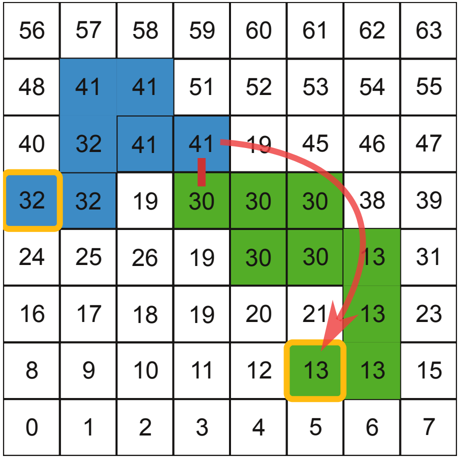
effort is O(L^3) at the critical point, but can be vectorized with O(L^2) threads

Union-find



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

Union-find

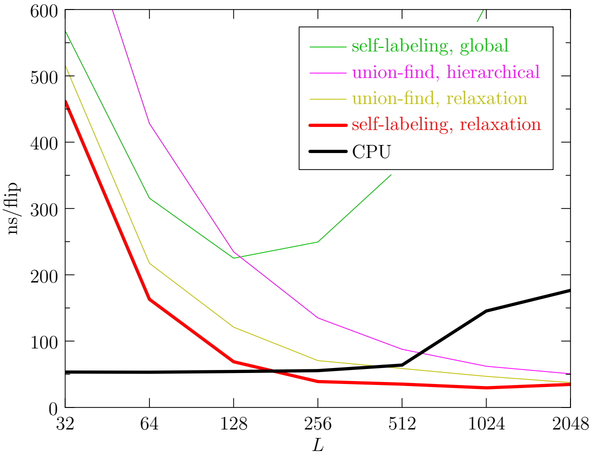


tree structure with two optimizations:

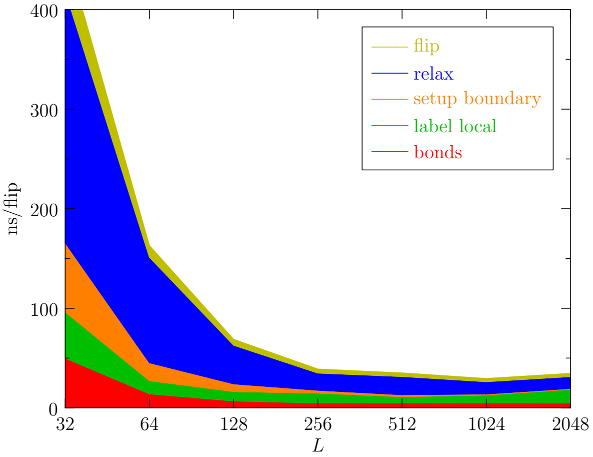
- balanced trees
- path compression

⇒ root finding and cluster union essentially O(1) operations

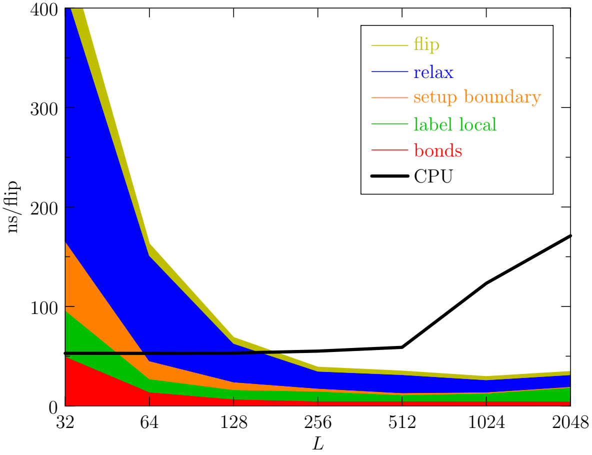
Performance



Performance



Performance



Performance

Problems with cluster labeling on GPU:

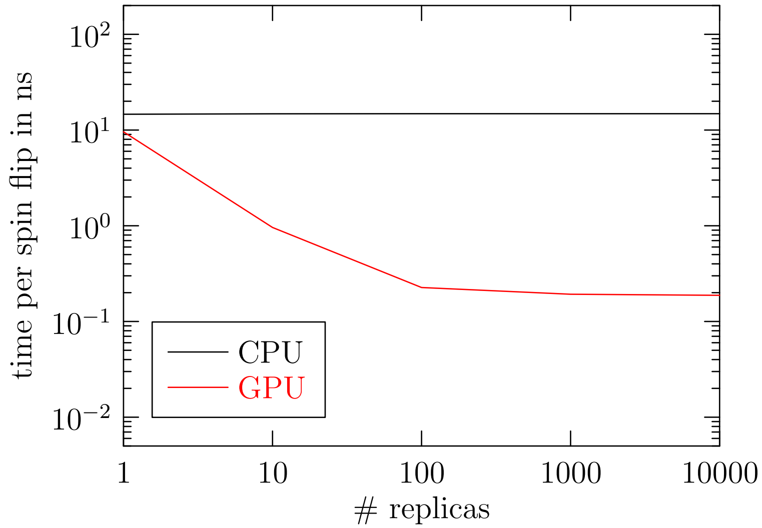
- overhead from parallelization (relaxation steps)
- lack of thread-level parallelism
- idle threads in hierarchical schemes
- best performance about 29 ns per spin flip, improvements possible
- problems *not* due to type of computations: 2.9 ns per spin flip for SW simulations of several systems in parallel

Spin glasses

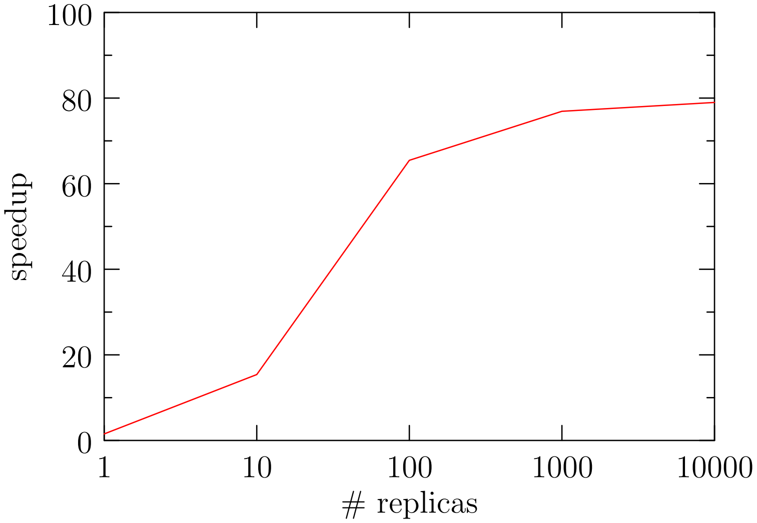
Simulate Edwards-Anderson model on GPU:

- 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

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.

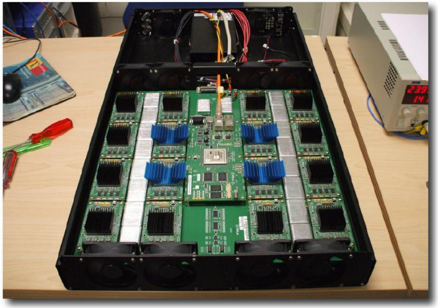
Further improvement: use multi-spin coding

- 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

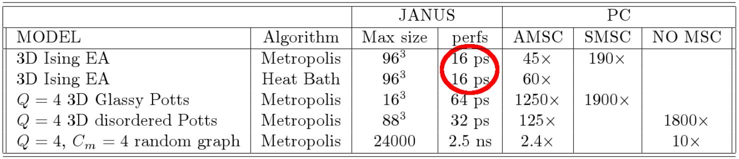
Janus

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



Janus

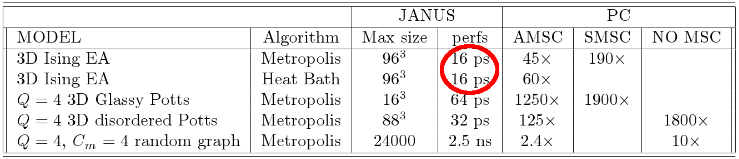
JANUS, a modular massively parallel and reconfigurable FPGA-based computing system.



		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	$45\times$	$190\times$	
3D Ising EA	Heat Bath	96^3	16 ps	$60\times$		
$Q = 4$ 3D Glassy Potts	Metropolis	16^3	64 ps	$1250\times$	$1900\times$	
$Q = 4$ 3D disordered Potts	Metropolis	88^3	32 ps	$125\times$		$1800\times$
$Q = 4$, $C_m = 4$ random graph	Metropolis	24000	2.5 ns	$2.4\times$		$10\times$

Costs:

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

Outlook

Conclusions:

- GPGPU promises significant speedups at moderate coding effort
- Requirements for good performance:
  - large degree of locality $\Rightarrow$ domain decomposition
  - suitability for parallelization (blocks) *and* vectorization (threads)
  - total number of threads much larger than processing units (memory latency)
  - opportunity for using shared memory $\Rightarrow$ performance is memory limited
  - ideally continuous variables

  $\Rightarrow$ maximum speed-up ~ 500
- effort significantly smaller than for special-purpose machines
- GPGPU might be a fashion, but CPU computing goes the same way

References:

- M. Weigel, *Simulating spin models on GPU*, Comput. Phys. Commun. (2010), in print, Preprint arXiv:1006.3865.
- M. Weigel, *Performance potential for simulating spin models on GPU*, Mainz preprint (2010).
- Code at `http://www.cond-mat.physik.uni-mainz.de/~weigel/GPU`.
- GPU workshop: late May, early June 2011, Mainz, Germany