Performance-/Cache-Optimierungen in der numerischen Simulation am Beispiel von Lattice-Boltzmann-Verfahren

M. Kowarschik, T. Pohl, U. Rüde,
K. Iglberger, J. Wilke, N. Thürey (LSS)
G. Wellein, F. Deserno, G. Hager,
F. Brechtefeld (RRZE)

Outline

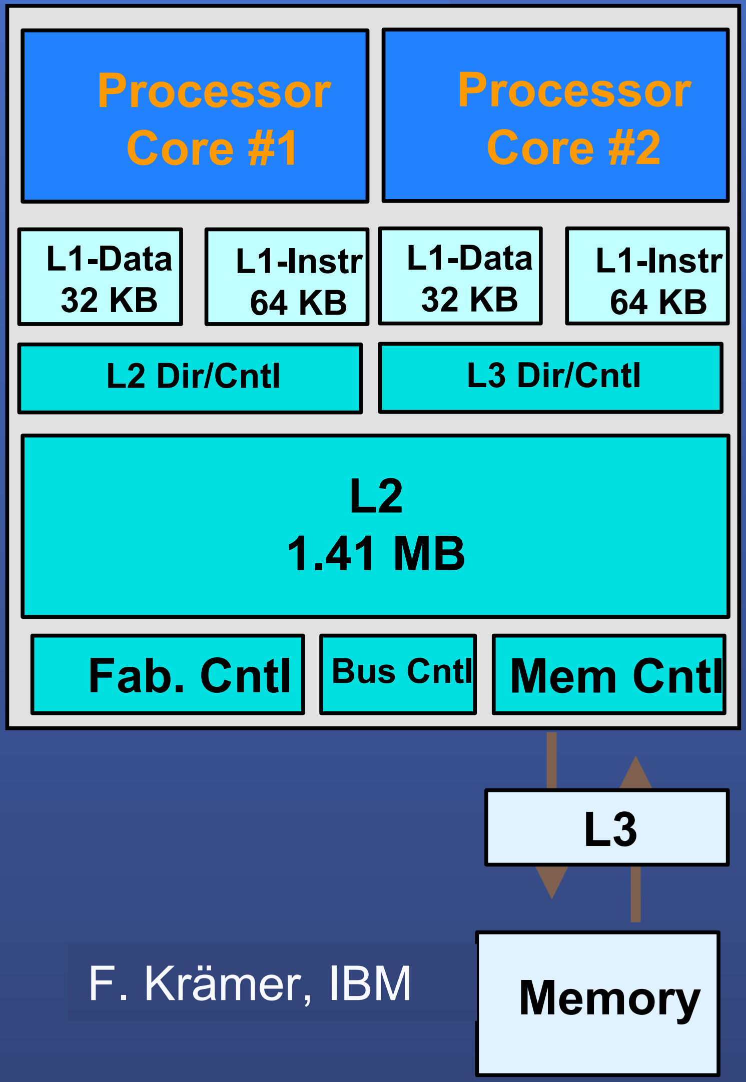
- High performance computer architectures
- Performance optimization techniques
  - Parallelization: shared/distributed memory
  - Optimization of single-CPU efficiency: memory hierachies (caches)
- Programming methodology to achieve
  - Flexibility/Portability
  - Efficiency

State of the art processors for HPC

		Intel Xeon DP	IBM Power4	Intel Itanium 2	NEC SX6
Peak Perf. Core-Frequency		6.1 GFlop/s 3.06 GHz	5.2 GFlop/s 1.3 GHz	4.0 GFlop/s 1.0 GHz	8 GFlop/s 0.5 GHz
Instruction Set		CISC/RISC	RISC	EPIC	Vector
# Float. Reg.		8 / 16	32	128	8 x 256 (Vector)
L1	Size	8 kB	32 KB	16 KB	---
	BW	96 GB/s	31.2 GB/s	32 GB/s	---
	Latency	2 cycles	4 cycles	1 cycle	---
L2	Size	512 kB	1.44 MB	256 KB	---
	BW	96 GB/s	124 GB/s	32 GB/s	---
	Latency	7 cycles	14 cycles	5-6 cycles	---
L3	Size	---	32 MB	3 MB	---
	BW	---	11.7 GB/s	32 GB/s	---
	Latency	---	≤ 340 cyc.	12-13 cyc.	---
Mem	BW	4.3 GB/s r/w	6.9 GB/s r&w	6.4 GB/s r/w	32 GB/s r/w
	Latency	~150 ns	~200 ns	~200 ns	Vectorization

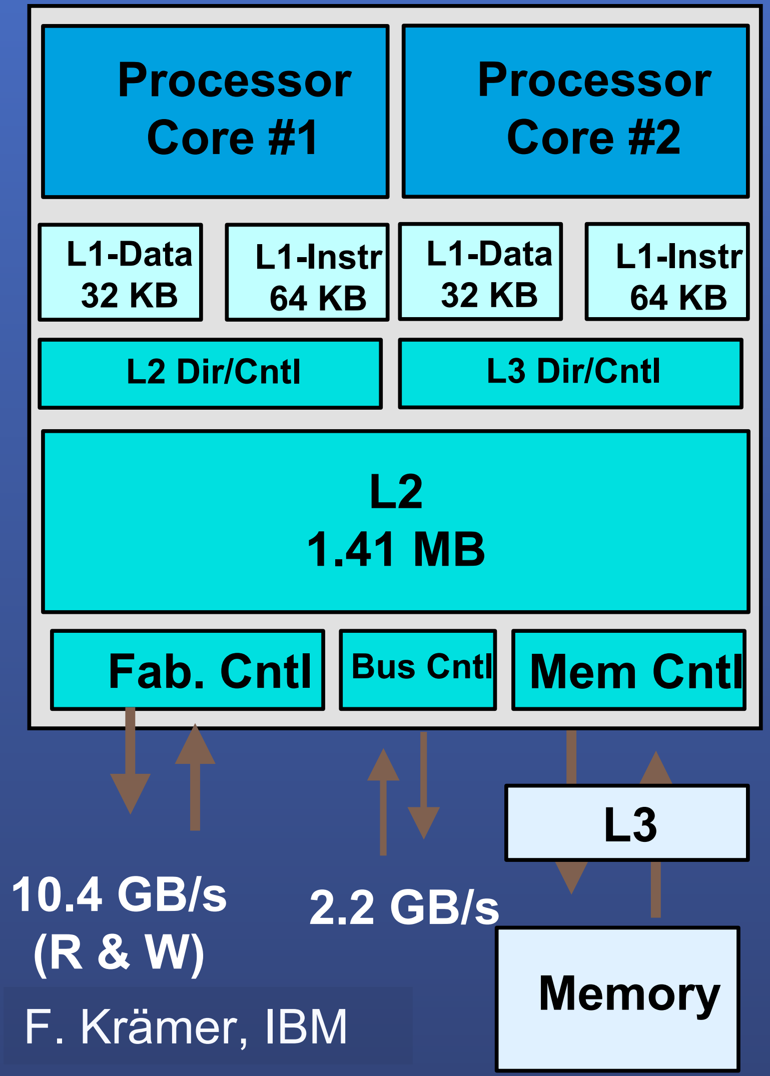
Example: POWER4 - "System On Chip" design

- Each core:
  - Up to 1.3 GHz/5.2 GFlop/s
  - 8 Prefetch Streams from L2, L3, mem.
  - 8 outstanding cache misses
- L1:
  - 64KB instruction, direct, 128B lines
  - 32KB data, 2-way associative,
  - 128B lines, store through
  - 1 load/store per cycle
- L2:
  - 3 slices of .47 MB
  - 8 way associative, 128B lines
  - 1 load + 1 store / CPU
  - 3* 32B*1.3 Ghz= 124 GB/s
  - 10-14 cycles latency (9.1-10.9 ns)



F. Krämer, IBM

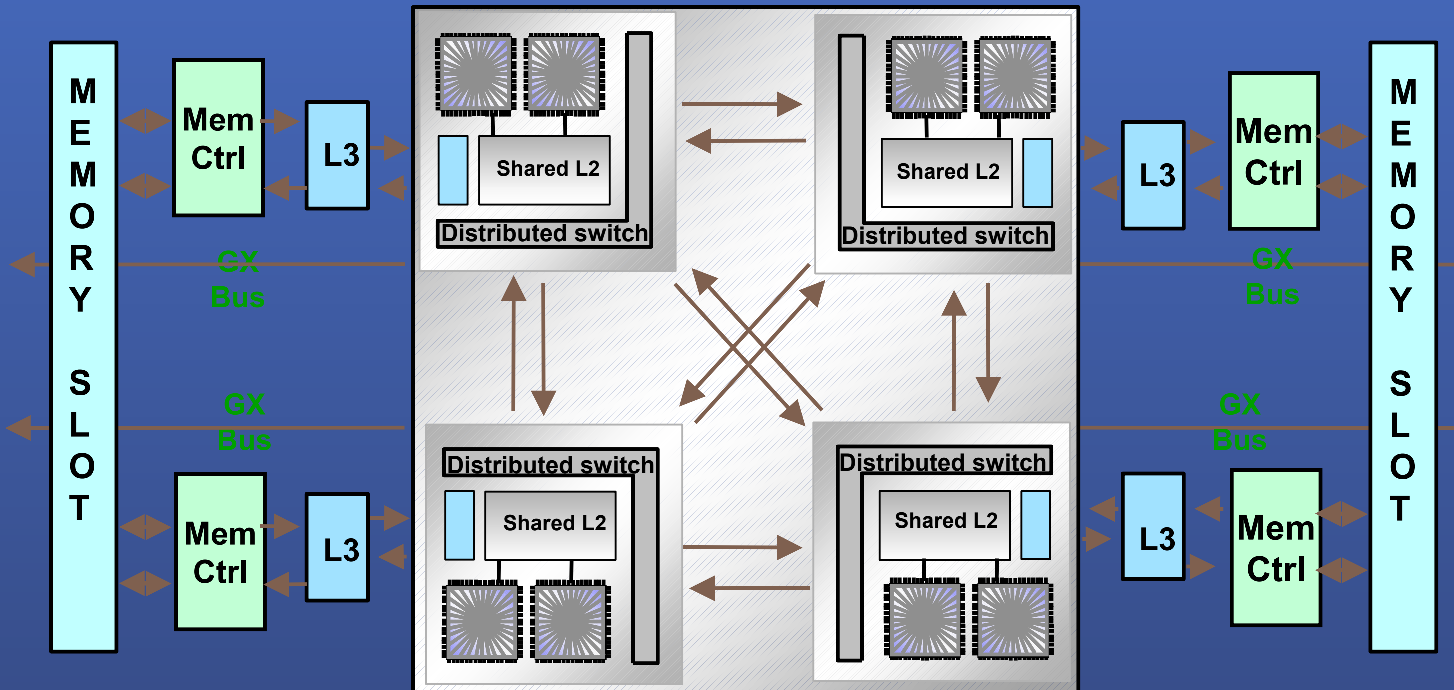
Example: POWER4 - "System On Chip" design



F. Krämer, IBM

- L3:
  - 32 MB/chip (external)
  - 8-way associative, 512B lines
  - 16 B 1:3= 6.9 GB/s read
  - 16 B 1:3= 6.9 GB/s write
  - total BW/chip 13.8 GBs
  - 92/100 cycles latency (local/remote)
- Memory:
  - up to 64/128 GB/ MCM
  - 1 or 2 port memory cards:
  - 6.4 GB/s read per port
  - 6.4 GB/s write per port
  - 252 cycles latency

Example: 8-way SMP "System-on-MCM" design

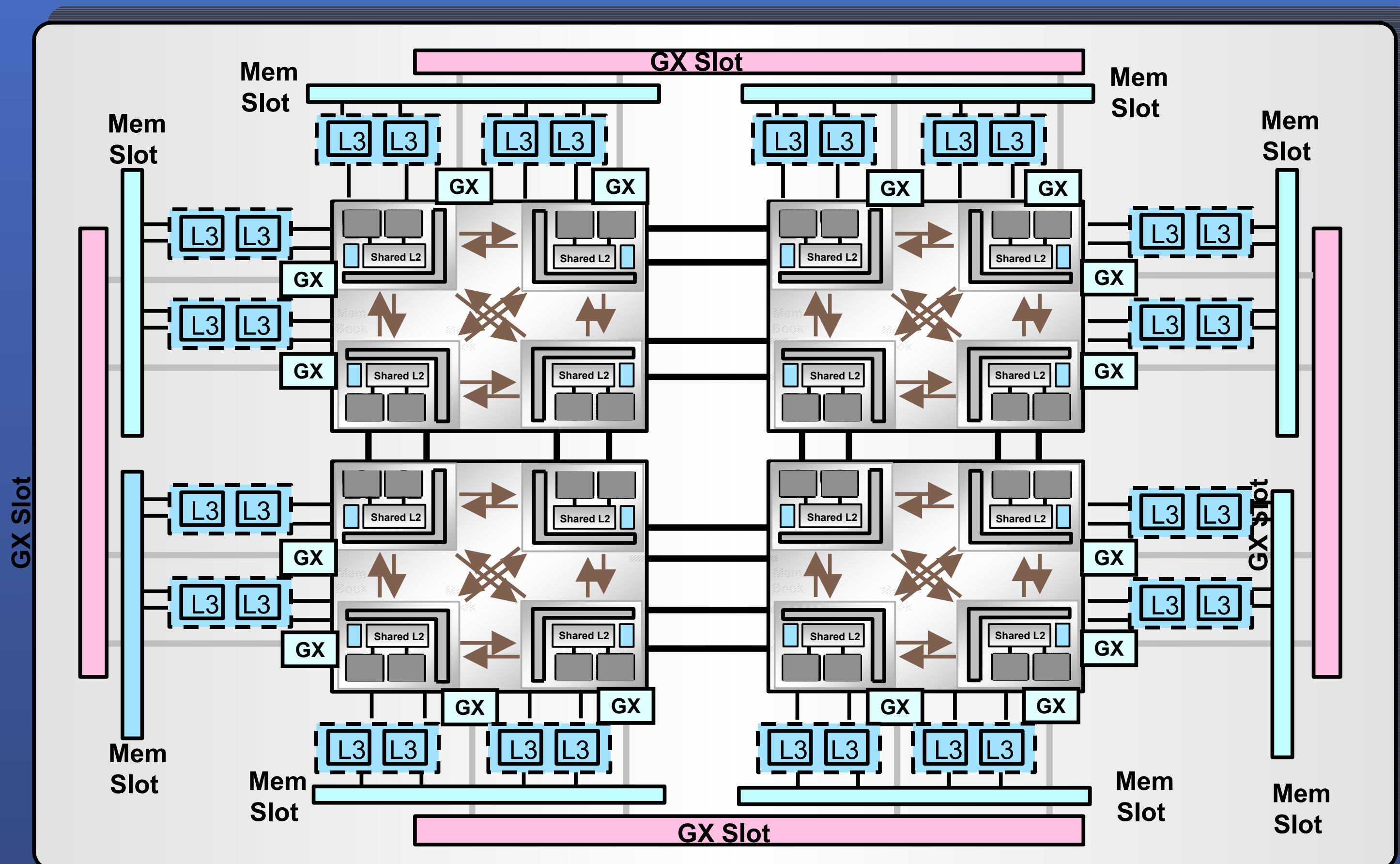


L3 cache shared across all processors

4** GX Bus **links for external connections

F. Krämer, IBM

Example: 32-way SMP p690, system design



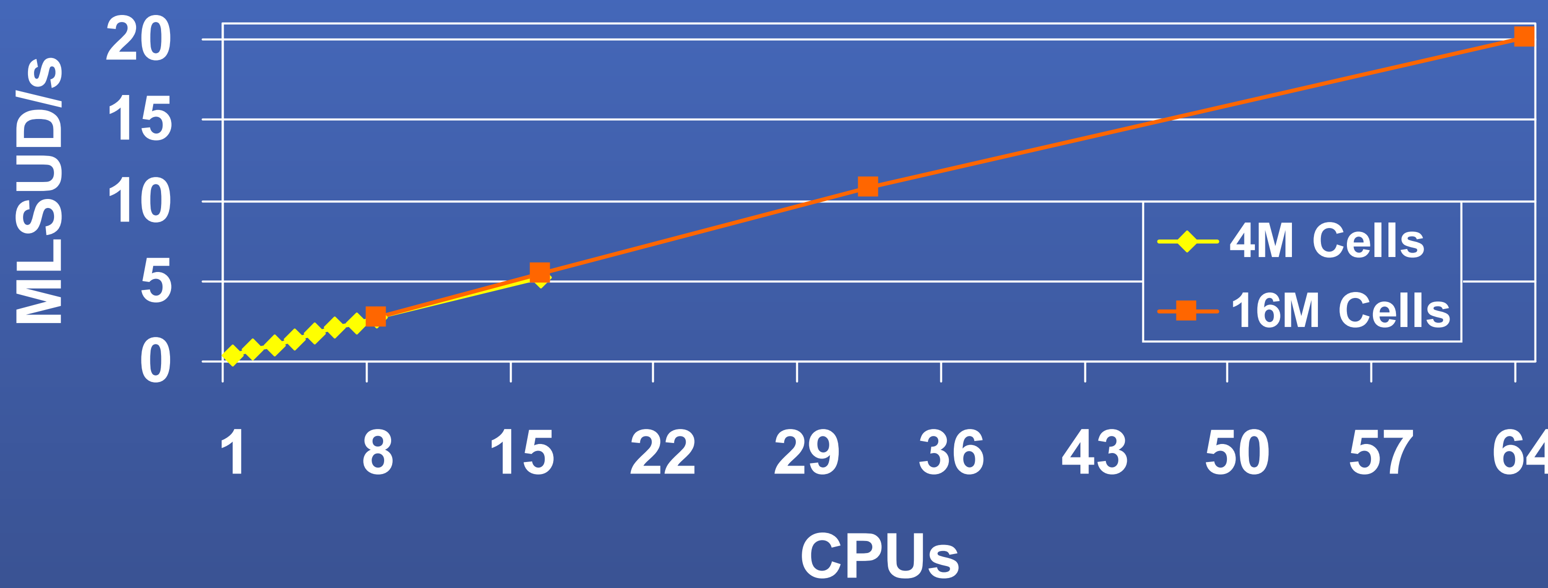
- 205 GB/s Memory-BW
- 16 GB/s IO-BW

F. Krämer, IBM

Hitachi SR8000-F1 at the Leibniz Comuting Center in Munich

Number of SMP Nodes	168
CPUs per Node	8
Number of Processors	1344
Peak Performance per CPU	1.5 GFlop/s
Peak Performance per Node	12 GFlop/s
Peak performance of the whole System	2016 GFlop/s
Memory per Node	> 8 GByte
Memory of the whole System	1376 GByte
Top 500 Ranking (Nov. 2003)	64

Performance on the SR8000 – Part I



- Parallelization by Domain Decomposition using MPI
- Almost linear Speed-up behavior
- Very bad single-CPU performance because of unoptimized LBM Code

Performance on the SR8000 – Part II

- Use COMPAS instead of OpenMP
- Add COMPAS directives to “explain” data dependencies to the compiler
- Do loop fusion to replace the three spatial loops by one single loop
- Remove conditional branches which causes redundant calculations but improves pipelining

Unoptimized Code	2.8 MLSUD/s
Optimized Code	20.6 MLSUD/s

(for 8 CPUs)

Optimized Code runs with 4.3 GFlop/s (36% of peak performance)

Increasing single-CPU performance by optimizing data locality

Caches work due to the locality of memory accesses (instructions + data)

(Numerically intensive) codes exhibit:

- Spatial locality:

  Data items accessed within a short time period are located close to each other in memory

- Temporal locality:

  Data that has been accessed recently is likely to be accessed again in the near future

Goal: Increase spatial and temporal locality in order to enhance cache utilization (cache-aware progr.)

Cache performance optimizations

- **Data layout optimizations**:

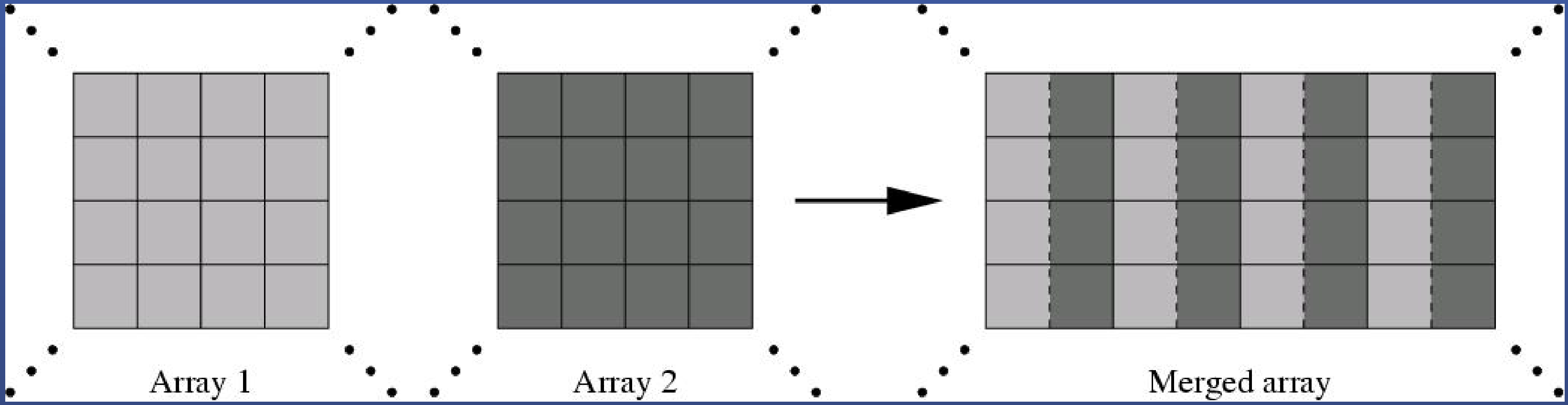
  Change the data layout in memory to enhance spatial locality
- **Data access optimizations**:

  Change the order of data accesses to enhance spatial and temporal locality

These transformations preserve numerical results and their introduction can (theoretically) be automated!

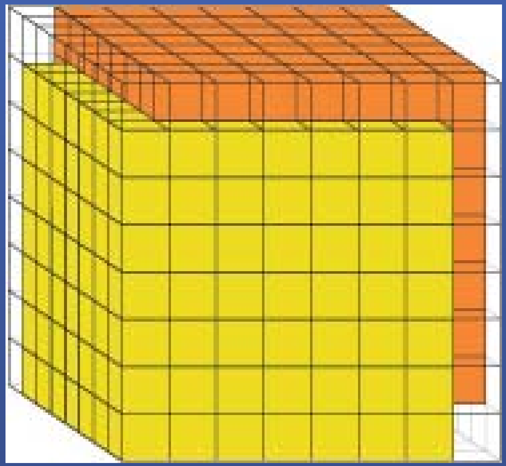
Data layout opt. for the LBM

- Grid merging: fusing the source grid of cells and the destination grid of cells to enhance spatial locality
- Works equivalently in 3D



Data layout opt. for the LBM

- Oberservation: no need to store two full grids
- Grid compression: save memory and enhance spatial locality by overlaying the source grid and the destination grid, thereby introducing diagonal shifts
- Works equivalently for the LBM in 2D and in 3D

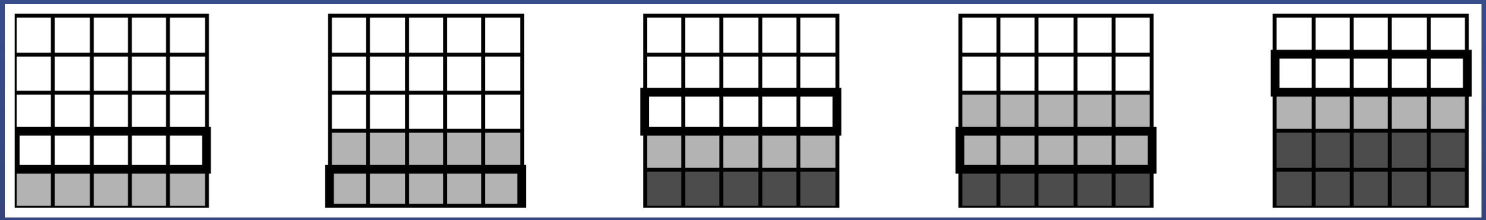


Data access opt. for the LBM

- Loop interchange: choose the most cache-conscious loop order, no influence on the numerical results
- Loop fusion: fuse the streaming step and the collision step into a single pass through the grid, save half of the passes through the grid

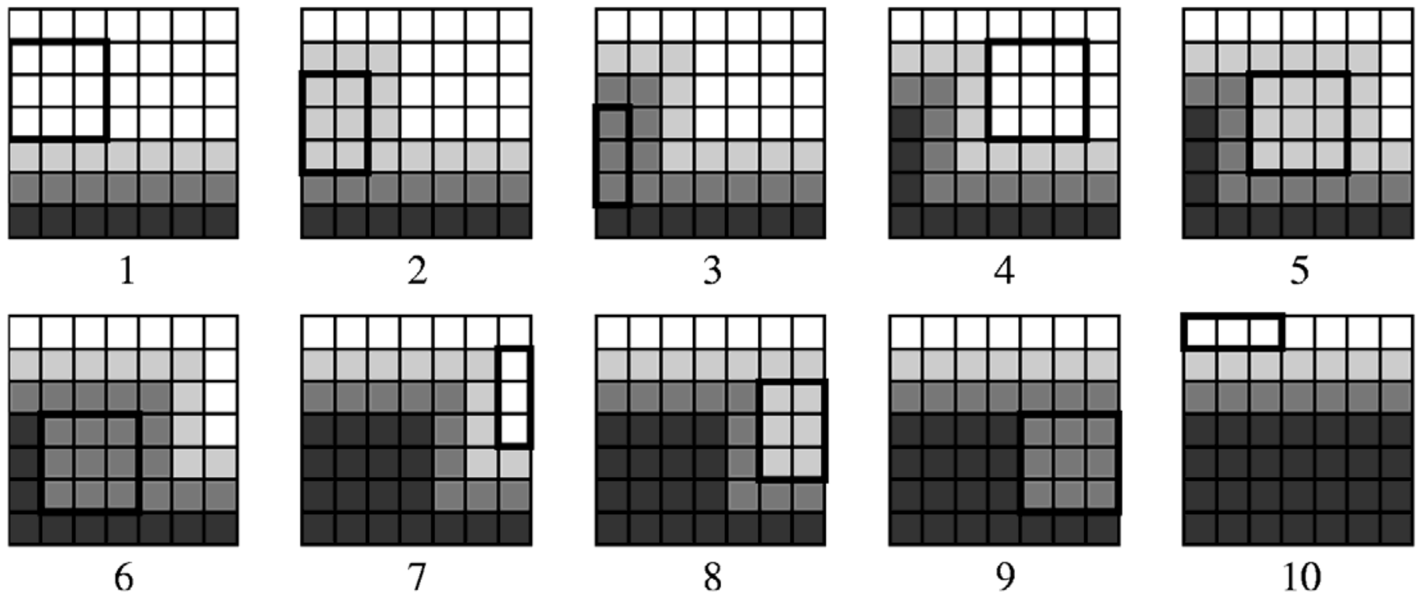
Data access opt. for the LBM

- Loop blocking: split the iteration space of a loop into blocks (that fit in cache) and perform as much work as possible on the block currently in cache
- Blocking the time loop of the LBM results in several time steps being executed during a single pass through the grid: 1-way blocking
- In 2D: blocking *n* time steps requires data from *n*+2 rows to fit in cache



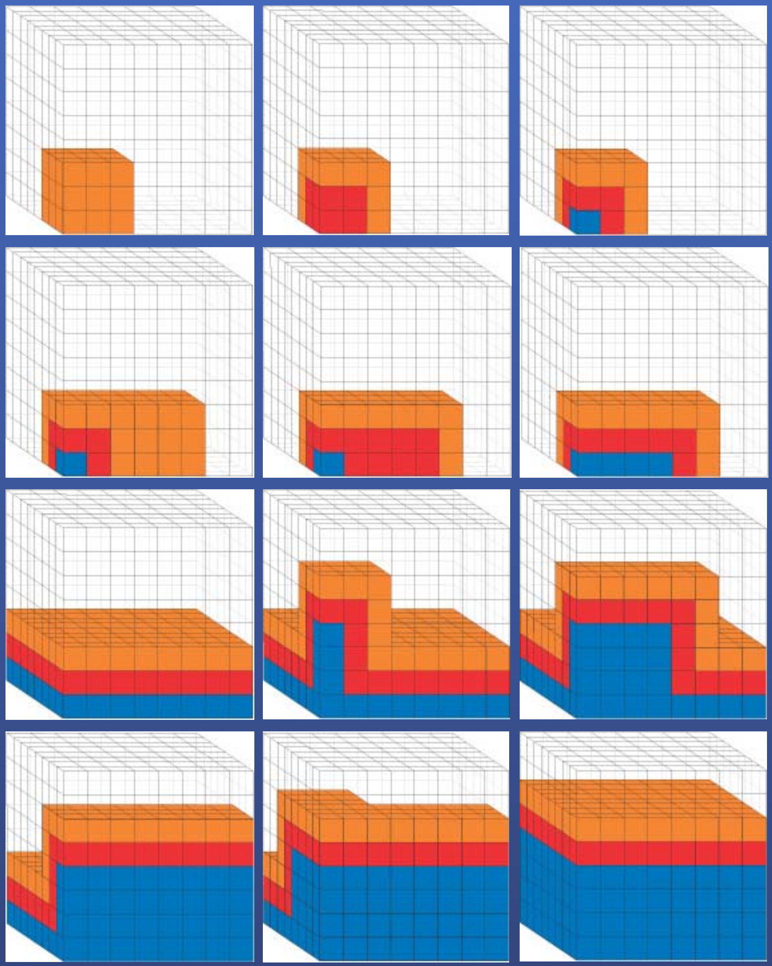
Data access opt. for the LBM

- Additionally, both loops along the spatial dimensions can be blocked (2D LBM): 3-way blocking
- Cache efficiency independent of the problem size, because block size does no longer depend on the problem size



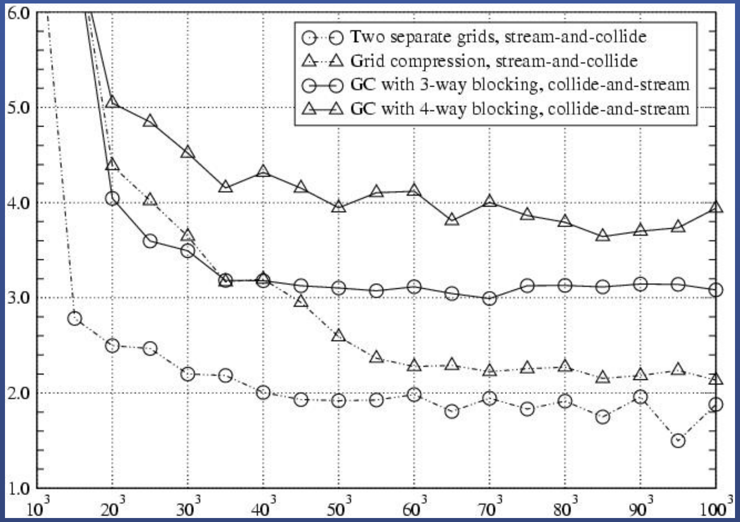
Data access opt. for the LBM

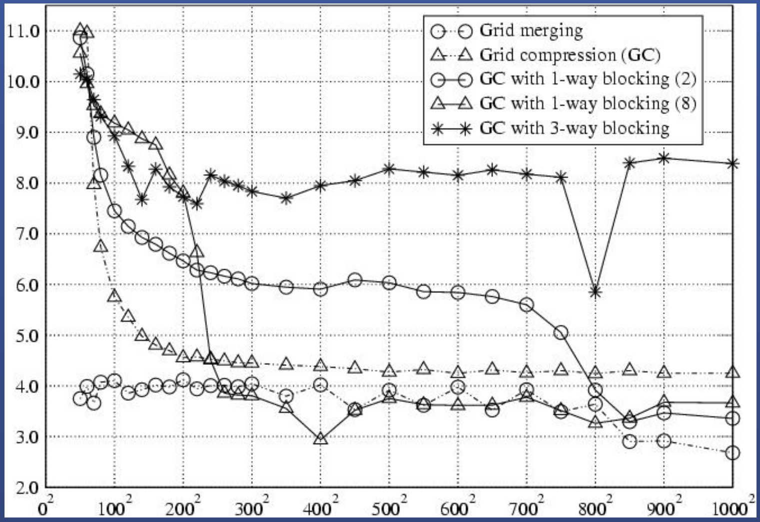
- In 3D, blocking the time loop and the three loops along the spatial dimensions results in the 4-way blocking technique
- 3D analogue to 3-way blocking in 2D



Performance speedups - examples

- Left: LBM-2D, C(++), AMD Athlon XP 2400+, Linux, gcc 3.2.1, right: LBM-3D, C, AMD Opteron 1.6 GHz, Linux, gcc 3.2.2
- Measure: MLUP/s (i.e., cell updates per second)
- **http://www10.informatik.uni-erlangen.de/dime**





Programming techniques

Seemingly conflicting goals:

- Portability/Flexibility:

  Code should run on a variety of (parallel) target platforms, including PC clusters, NUMA machines, etc.

- Efficiency:

  Code should run as efficiently as possible (in terms of MLUP/s) on each target platform

How can this conflict be solved?

Programming techniques

Work in progress covers:

- Parameter-based optimization (e.g., block sizes, array paddings)
- Exhaustive parameter search: AEOS paradigm, e.g., used for ATLAS
- Separation of data layout and data access patterns
- Domain-specific (i.e., LBM-specific) languages
- Automatized generation of optimized source code
- Etc.

Conclusions

- High performance architectures differ enormoulsy
- Code optimizations must address parallel efficiency and single-node performance
- Sophisticated programming techniques are required to develop portable and efficient codes
- http://www10.informatik.uni-erlangen.de/dime