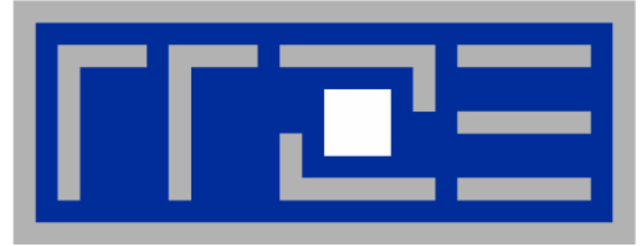


# Efficient implementation of simple lattice Boltzmann kernels

**G. Wellein, T. Zeiser, G. Hager, S. Donath**

*HPC Services*

*Regionales Rechenzentrum Erlangen*

*Friedrich-Alexander-University Erlangen-Nuremberg*

*Germany*

*International Conference for Mesoscopic Methods in Engineering and Science 2006 – Short Course*

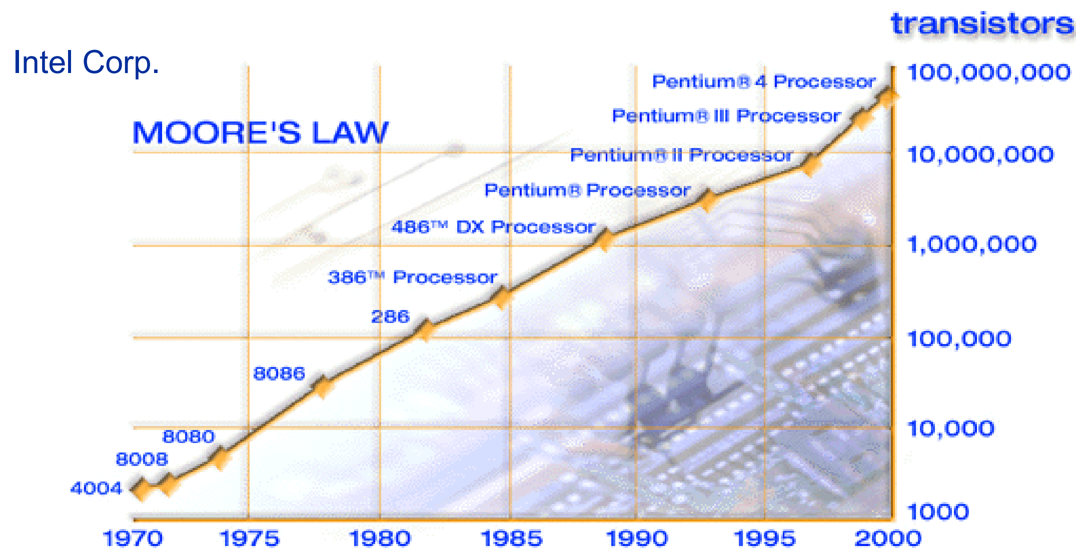
CX HPC

# Survey

- Introduction
- Memory hierarchies of modern processors
- Implementation of lattice Boltzmann method
- Optimization of data access for 3D lattice Boltzmann method
- Summary
- Literature

# Introduction
# Ever growing processor speed & Moore´s Law

- **1965 G. Moore (co-founder of Intel) claimed**
  #transistors on processor chip doubles every 12-24 months

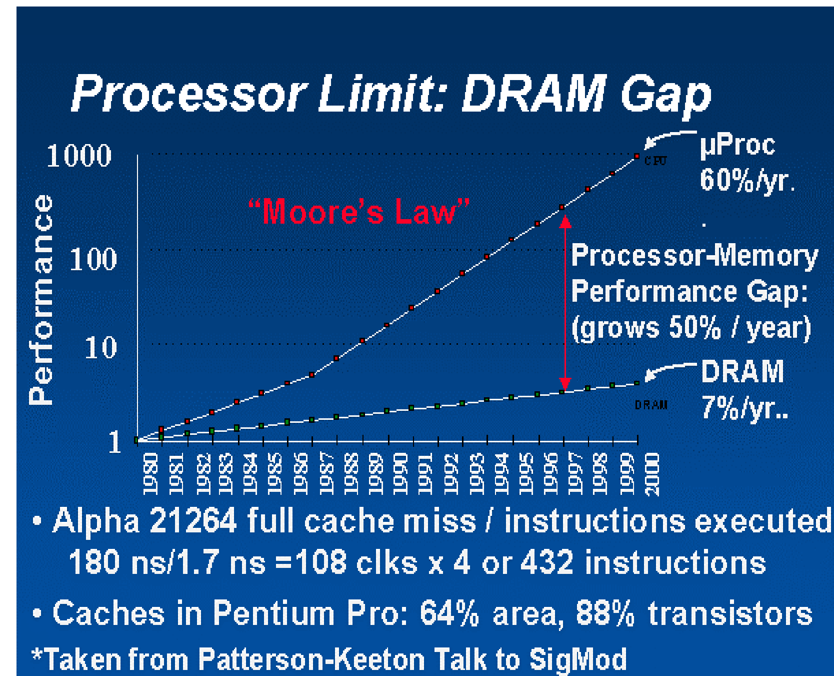


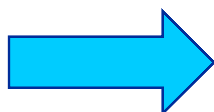

- **Processor speed grew roughly at the same rate**
  My computer: 350 MHz (1998) – 3,000 MHz (2004)
  Growth rate: 43 % p.y. -> doubles every 24 months

- **Why worry about performance?**

# Why worry about performance: Memory gap

- **Memory (DRAM) Gap**
  - Memory bandwidth grows only at a speed of 7% a year
  - Memory latency remains constant / increases in terms of processor speed
  - Loading a single data item from main memory can cost 100s of cycles on a 3 GHz CPU

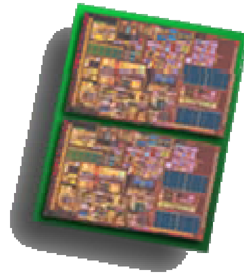
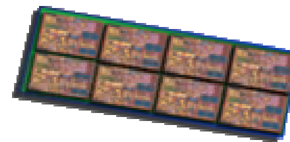
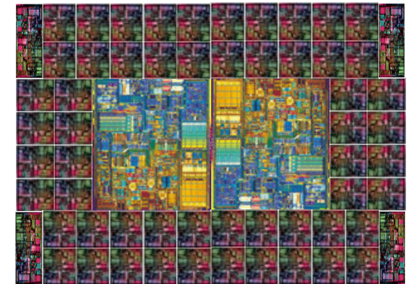


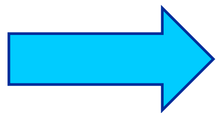



**Optimization of main memory access is mandatory for most applications**

## Why worry about performance: Multi-core

- **Multi-Core Processors – The game is over…**
  - Problem: Moore's law is still valid but increasing clock speed hits a technical wall (heat)
  - Solution: Reduce clock speed of processor but put 2 (or more) processors (cores) on a single silicon die
  - We will have to use many less powerful processors in the future

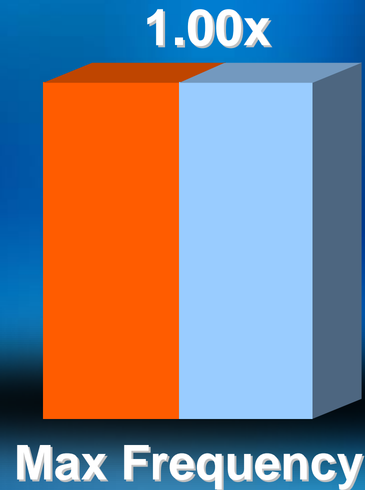






**Parallelization will be mandatory for most applications in the future**

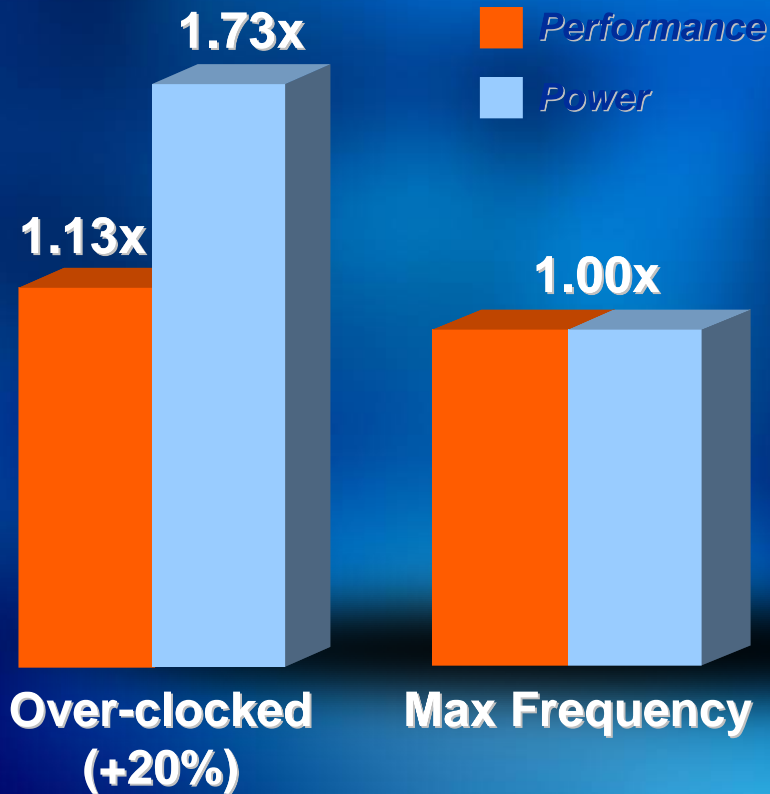
# Why worry about performance: Why multi-core?

*By courtesy of D. Vrsalovic, Intel*



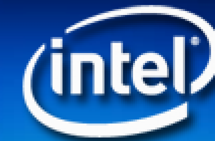
# Why worry about performance: Why Multi-Core?

*By courtesy of D. Vrsalovic, Intel*



# Why worry about performance: Why Multi-Core?

*By courtesy of D. Vrsalovic, Intel*




Performance

Power

1.13x

1.73x

Over-clocked (+20%)

1.00x

Max Frequency

0.87x

0.51x

Under-clocked (-20%)



gerhard.wellein@rrze.uni-erlangen.de

## Why worry about performance: Why Multi-Core?

*By courtesy of D. Vrsalovic, Intel*

Dual-Core
Performance
Power

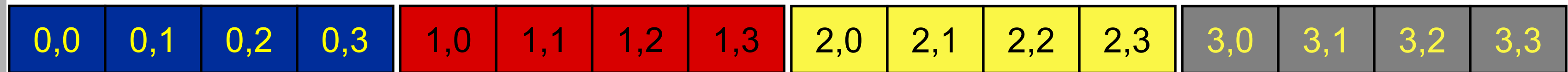
1.13x
1.73x
Over-clocked (+20%)

1.00x
Max Frequency

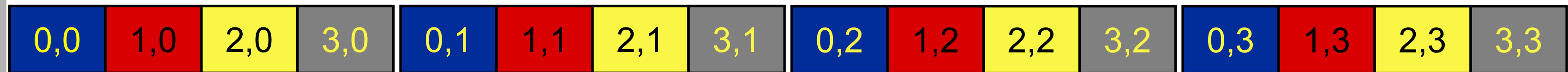
1.73x
1.02x
Dual-core (-20%)

- **FORTRAN or C is recommended**
- **Be aware of different memory mapping for FORTRAN & C:**

```
double a(4,4) // C version
```



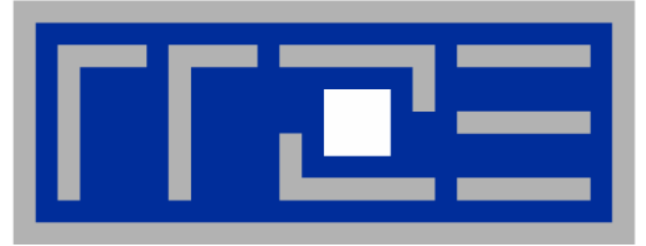

```
real*8 a(0:3,0:3)  ! FORTRAN Version
```




**Ordering of nested loops!**

- **Use appropriate compiler flags to ensure optimization,** e.g. for Intel compiler: `-O3 -xW -fno-alias` (if no pointer aliasing is used)

 gerhard.wellein@rrze.uni-erlangen.de 



# Memory hierarchies of modern processors

# Memory hierarchies

gerhard.wellein@rrze.uni-erlangen.de

# Memory hierarchies
## Cache-based dual-core architectures

| *SSE2 | | Intel Xeon5100 | AMD Opteron | Intel Itanium2 |
|---|---|---|---|---|
| **Peak Performance** Core frequency | | **12.0 GFlop/s** 3.0 GHz | **5.2 GFlop/s** 2.6 GHz | **6.4 GFlop/s** 1.6 GHz |
| **#Registers** | | **8 / 16*** | **16 / 32*** | **128** |
| **L1** | **Size** | 32 kB | 64 kB | *16 KB* |
| | **BW** | 96 GB/s | 41.6 GB/s | *51.2 GB/s* |
| | **Latency** | 2 cycles | 3 cycles | *1 cycle* |
| **L2** | **Size** | **4 MB (2 cores)** | **1 MB** | 256 KB |
| | **BW** | **96 GB/s** | **41.6 GB/s** | 51.2 GB/s |
| | **Latency** | **7 cycles** | **~13 cycles** | 5-6 cycles |
| **L3** | **Size** | | | **6 / 12 MB** |
| | **BW** | | | **51.2 GB/s** |
| | **Latency** | | | **12-13 cycles** |
| **Memory** | **BW** | **10.6 GB/s** | **6.4 GB/s** | **8.5 GB/s** |
| | **Latency** | **~200 ns** | **< 100 ns** | **~200 ns** |

# Memory hierarchies
## Cache structure

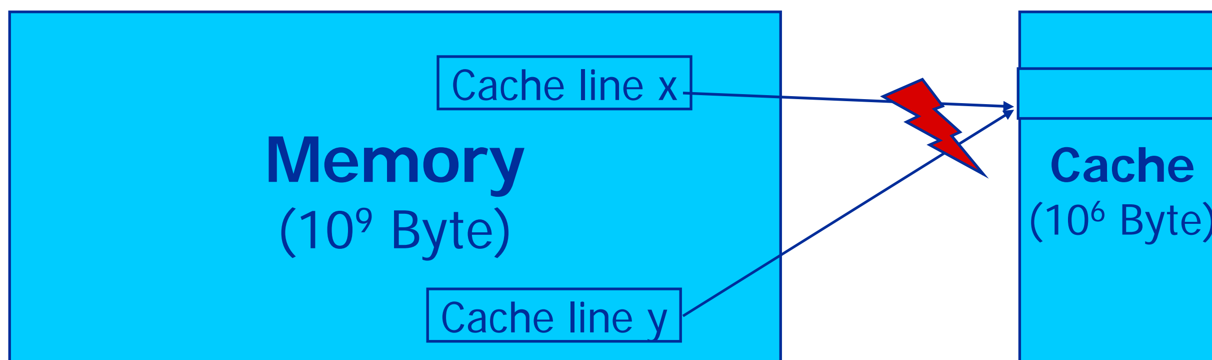
- Caches are organized in cache lines that are fetched/stored as a whole (e.g. 128 bytes=16 double words)
  - If one item is needed, the cache line it belongs to is fetched (miss)
  - Cache line fetch/load has large latency penalty
  - “neighboring“ items can then be used from cache



```
do i=1,n
  s = s + a(i)*a(i)
enddo
```

# Memory hierarchies: Cache structure

- Cache line data is always consecutive
  - Cache use is optimal for contiguous access (stride 1)
  - Non-consecutive access reduces performance (worst case: ≥ cache line size)
  - **Ensure spatial locality by blocking or optimizing data layout**
- Caches (~MB) must be mapped to memory locations (~GB)
  - **Cache multi-associativity enhances utilization**



- *Cache mapping:*
  - Pairing of memory locations with cache line
  - e.g. mapping 1 GB of main memory to 512 KB of cache

- *Directly mapped cache:*
  - Every memory location can be mapped to exactly one cache location
  - If cache size=n, i-th memory location can be stored at cache location mod(i,n)
  - Memory access with stride=cache size will not allow caching of more than one line of data, i.e. effective cache size is one line!
  - No penalty for stride-one access

# Memory hierarchies
## Cache mapping – Directly mapped

| 0 | 1 | 2 | | | ... | N-1 |
|---|---|---|---|---|---|---|
| N | N+1 | N+2 | | | Memory | 2N-1 |
| | | | | | | |
| | | | | | | |

| | | | | | ... Cache | |
|---|---|---|---|---|---|---|

Example: Directly mapped cache. Each memory location can be mapped to one cache location only.

*E.g. Size of main memory= 1 GByte; Cache Size= 256 KB*
*→ 4096 memory locations are mapped to the same cache location*

# Memory hierarchies
## Cache mapping – Associative caches

- *Set-associative cache:*
  - m-way associative cache of size m x n: each memory location i can be mapped to the m cache locations j*n+mod(i,n), j=0..m-1
  - E.g.: 2-way set associative cache of size 256 KBytes:

| 1 | 2 | 3 | 4 | 5 | ... | 128 KB |
|---|---|---|---|---|---|---|
| 128KB+1 | | | | | | 256 kB |

  - If all m locations are taken, one has to be overwritten on next cache load using different strategies:
    - least recently used (LRU)
    - Random
    - not recently used (NRU)

# Memory hierarchies
## Cache mapping – Associative caches

Example: 2-way associative cache. Each memory location can be mapped to two cache locations:
*E.g. Size of main memory= 1 GByte; Cache Size= 256 KB -> 8192 memory locations are mapped to two cache locations*

# Memory hierarchies
## Cache thrashing

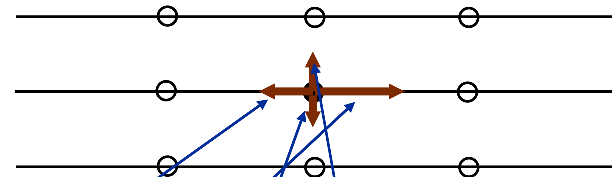
- If many memory locations are used that are mapped to the same m cache slots, cache reuse can be very limited even with m-way associative caches

- Effective cache size is usually less than m x n for real-world applications

- **Warning: Using powers of 2 in the leading array dimensions of multi-dimensional arrays should be avoided!**

# Memory hierarchies
## Cache thrashing - Example

Example: 2D – square lattice
At each lattice point the 4 velocities for each of the 4 directions are stored

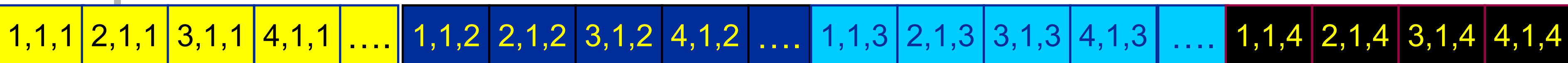


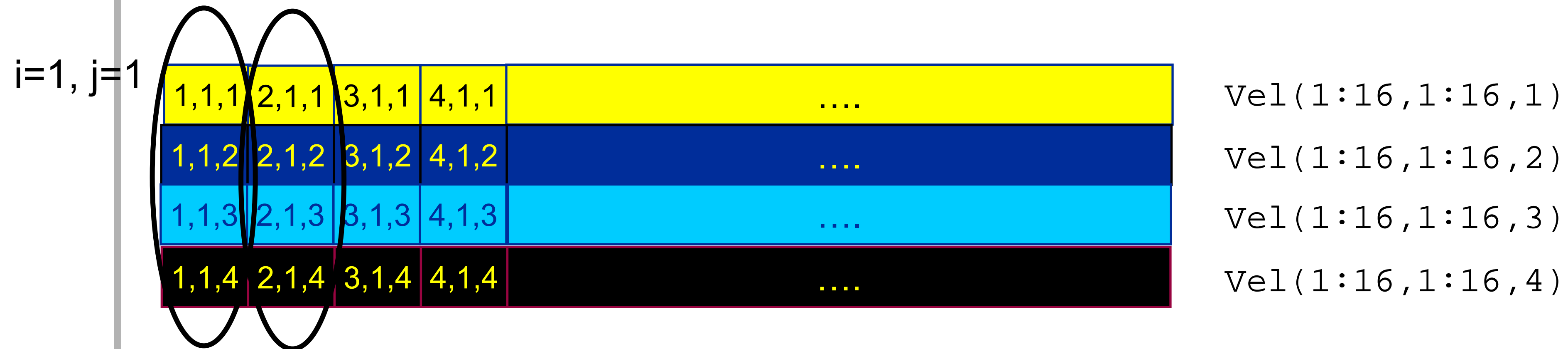
```
N=16
real*8 vel(1:N , 1:N, 4)
......
s=0.d0
do j=1,N
   do i=1,N
      s=s+vel(i,j,1)-vel(i,j,2)+vel(i,j,3)-vel(i,j,4)
   enddo
enddo
```

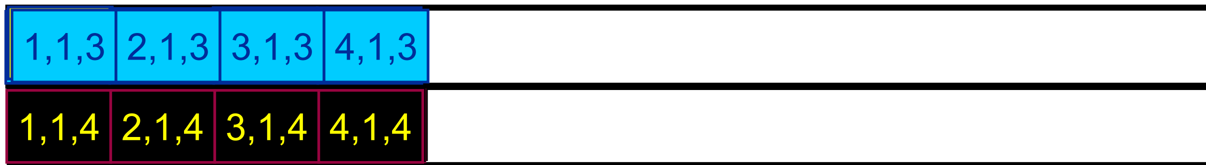
# Memory hierarchies
# Cache thrashing - Example

Memory to cache mapping for `vel(1:16, 1:16, 4)`

Cache: 256 byte (=32 double) / 2-way associative / Cache line size=32 byte




Cache:

2 rows with 16 double each

Each cache line must be loaded 4 times from main memory to cache!

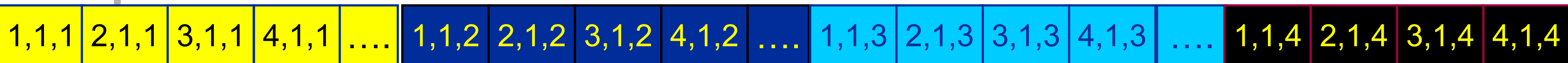


gerhard.wellein@rrze.uni-erlangen.de

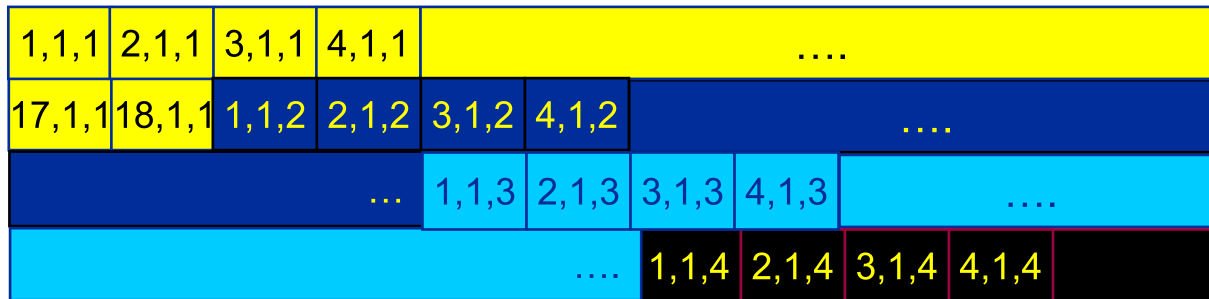
# Memory hierarchies
# Cache thrashing - Example

Memory to cache mapping for `vel(1:18, 1:18, 4)`

Cache: 256 byte (=32 doubles) / 2-way associative / Cache line size=32 byte




i=1, j=1



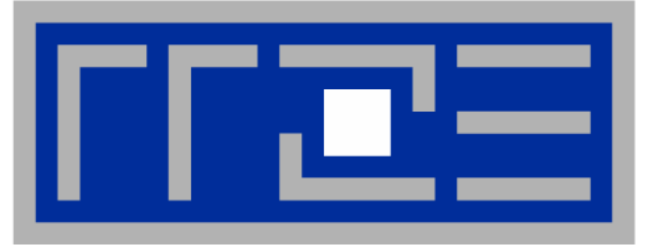
Cache:

2 rows with 16 doubles each

Each cache line needs only be loaded **once** from memory to cache!


gerhard.wellein@rrze.uni-erlangen.de


CX HPC



# Implementation of the

# 3D lattice Boltzmann method

- **Boltzmann Equation**

$$\partial_t f + \xi \cdot \nabla f = -\frac{1}{\lambda}[f - f^{(0)}]$$

$\xi$ ... particle velocity
$f^{(0)}$ ... equilibrium distribution function
$\lambda$ ... relaxation time

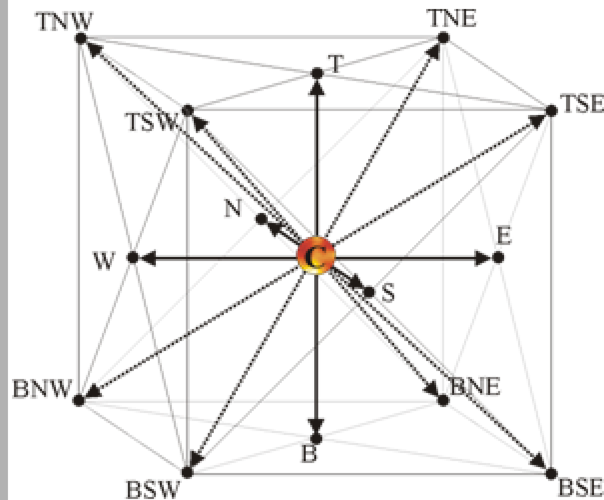
- **Discretization of particle velocity space**
  **(finite set of discrete velocities)**

$$\partial_t f_\alpha + \xi_\alpha \cdot \nabla f_\alpha = -\frac{1}{\lambda}[f_\alpha - f_\alpha^{(eq)}]$$

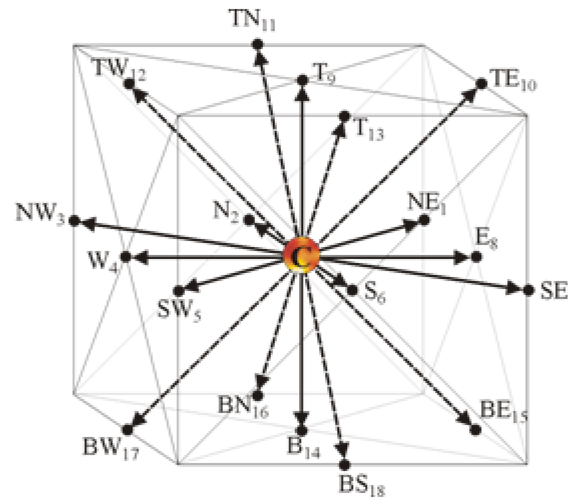
$$f_\alpha(\vec{x},t) = f(\vec{x},\xi_\alpha,\mathrm{t})$$
$$f_\alpha^{(eq)}(\vec{x},t) = f^{(0)}(\vec{x},\xi_\alpha,\mathrm{t})$$

**$\xi_\alpha$ – determined by discretization scheme**

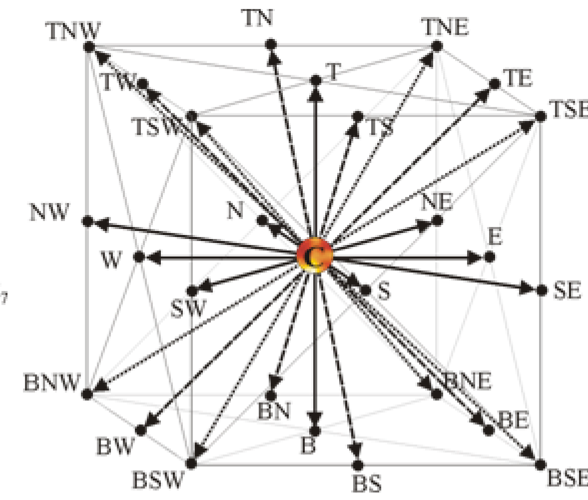
- Different discretization schemes in 3D
  - Numerical accuracy and stability
  - Computational speed and simplicity




D3Q15




D3Q19




D3Q27

**We choose D3Q19 because of good balance between stability and computational efficiency**

# Lattice Boltzmann method
## Stream and collide steps

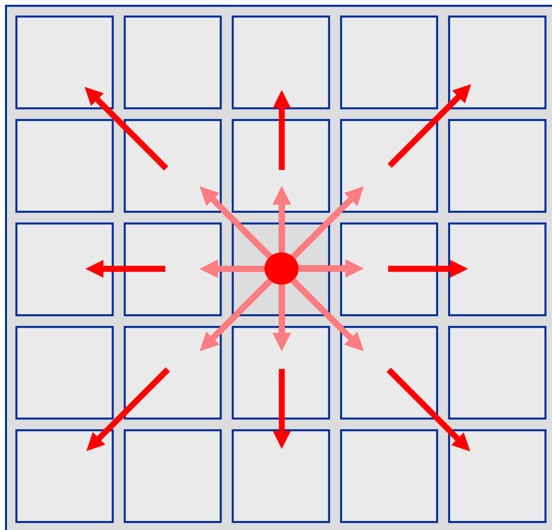
- **Discretization in space $\vec{x}$ and time $t$:**

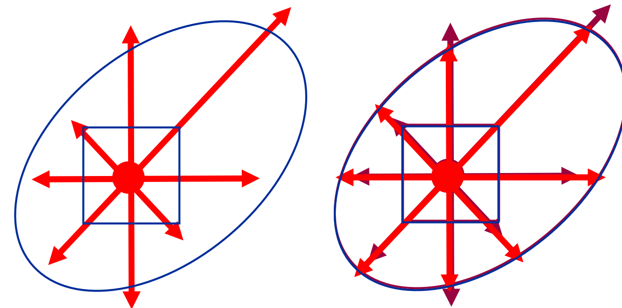
Relaxation of distribution function: Mainly floating point operations

collision step: $\tilde{f}_\alpha(x_i,t) = f_\alpha(x_i,t) - \omega\,[f_\alpha(x_i,t) - f_\alpha^{(eq)}(x_i,t)]$

streaming step: $f_\alpha(x_i + \vec{e}_\alpha \delta t, t + \delta t) = \tilde{f}_\alpha(x_i,t)$

Propagation of distribution function: Moving data only





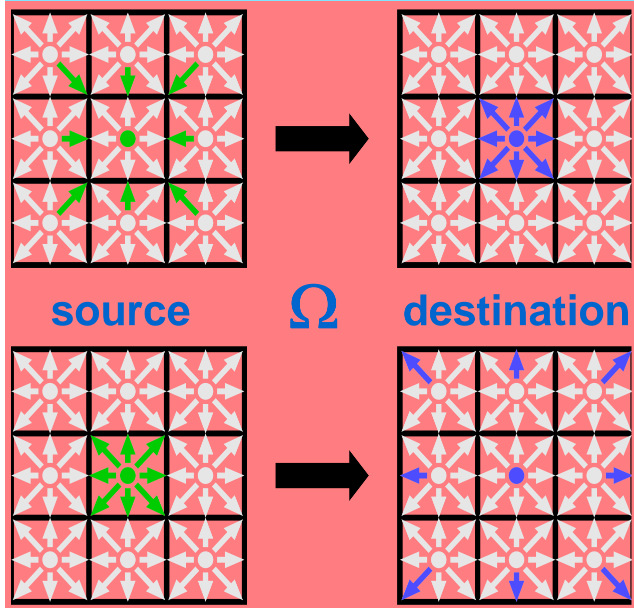

gerhard.wellein@rrze.uni-erlangen.de

# Lattice Boltzmann method
## Stream and collide steps

- **Discretization in space $\vec{x}$ and time $t$:**

collision step: $$\tilde{f}_\alpha(x_i,t) = f_\alpha(x_i,t) - \omega\,[f_\alpha(x_i,t) - f_\alpha^{(eq)}(x_i,t)]$$

streaming step: $$f_\alpha(x_i + \vec{e}_\alpha \delta t, t + \delta t) = \tilde{f}_\alpha(x_i,t)$$




Stream-Collide (Pull-Method)

**Get the distributions from the neighboring cells in the source array and store the relaxated values to one cell in the destination array**

Collide-Stream (Push-Method)

**Take the distributions from one cell in the source array and store the relaxed values to the neighboring cells in the destination array**

**We choose Collide-Stream in what follows**

# Lattice Boltzmann method: Basic implementation strategy

- Use "*full matrix*" representation: **F(0:18,x,y,z,0:1)**
  - **F(α,x,y,z,0)** = $f_\alpha(x,y,z,t)$ **& F(α,x,y,z,1)** = $f_\alpha(x,y,z,t+1)$ **(t odd;** $\vec{x}$**=(x,y,z))**
  - For complex geometries, *vector-like representations* might be interesting as they can block out the solid parts
  - However, the connectivity of the cells has to be stored additionally (therefore, you need at least 1/3 of solid to save memory)

## Basic Optimizations

- Analyze relaxation step: Many operations can be eliminated (common sub-expressions, zero-velocity components):
  # of floating point operations depends on compiler & optimization level!
- **Combine Collide & Stream step in a single loop to minimize data transfer!**

# Lattice Boltzmann method
## Basic implementation strategy

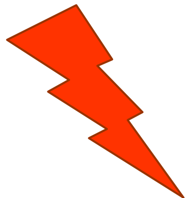
```
double precision F(0:18,0:xMax+1,0:yMax+1,0:zMax+1,0:1)
do z=1,zMax
   do y=1,yMax
      do x=1,xMax
           if( fluidcell(x,y,z) ) then
              LOAD F(0:18,x,y,z,t)
              Relaxation (complex computations)
              SAVE F( 0,x  ,y  ,z  ,t+1)
              SAVE F( 1,x+1,y+1,z  ,t+1)
              SAVE F( 2,x  ,y+1,z  ,t+1)
              SAVE F( 3,x-1,y+1,z  ,t+1)
              …
              SAVE F(18,x  ,y-1,z-1,t+1)
           endif
      enddo
   enddo
enddo
```

LD 1-2 Cachelines (cont. access)

Collide Step

Stream Step

LD & ST
19 Cachelines



If cache line of store operation is not in cache it must be loaded first (RFO)!

#load operations: 19*xMax*yMax*zMax + 19*xMax*yMax*zMax

#store operations: 19*xMax*yMax*zMax



CX HPC

# Lattice Boltzmann method
# Performance (1)

- Standard performance measure for LBM:
  *Million fluid cell updates per second: MFLUP/s*
  (or *Million Lattice cell updates per second: MLUP/s*)

## Performance estimation

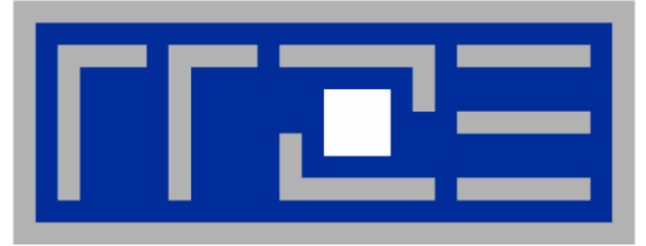
- Assumption:
  200 Floating point operations per fluid cell update
- If data transfer is infinite fast (all data fits into cache):
  - Max. MFLUP/s = PeakPerformance / (200 Flop/cell update)
  - Peak Performance = 4,000 MFLOP/s → Max. 20 MFLUP/s
  - In reality complex kernels do not achieve peak performance and transfer speed is finite even for caches → ~10-15 MFLUP/s

# Lattice Boltzmann method
# Performance (2)

- If data must be transferred from and to main memory in each time step:
  - Assumption: full use of each cache line loaded
  - Data to be transferred for a single fluid cell update: (2+1)*19*8 Byte → 456 Bytes/(fluid cell update)
  - Max. MFLUP/s= MemoryBandwidth / (456 Bytes/(fluid cell update))
  - MemoryBandwidth = 6,000 MByte/s → Max. 13 MFLUP/s
  - In reality even simple kernels only achieve 50%-60% of theoretical main memory bandwidth → ~ 6-8 MFLUP/s
- Crossover between cache and memory bound computations:
  *2 *19 *xMax*$^3$ ** 8Byte ~ L2/L3 cache size* → xMax ~ 14-15 (for 1 MB cache)



CX HPC



# Optimization of data access for

# 3D lattice Boltzmann method

- In our case we have a 5-dimensional array F
  - `x,y,z` the three spatial directions
  - `i` the discrete velocity directions
  - `t` the source/destination pointer (current / next time step)

- In principle, any permutation of these indices can be used…
  …but which is the most efficient?

- In the following, it is assumed that by increasing the first index by one, you go to the physically next location in memory (FORTRAN)

  - `F(i,x,y,z,t)` „collision optimized“ („array of structures”)

  - `F(x,y,z,i,t)` „propagation optimized“ („structure of arrays”)

# Optimization of data access
## Collision optimized layout: F(0:18,x,y,z,t)

Intel Xeon 3.4 GHz (1 MB L2; 5.3 GByte/s) → 5-6 MFLUP/s

Nx = Ny = Nz

Nx * Ny * Nz = const

Intel Xeon 3.4 GHz

CollOpt

Data set fits into cache

Performance breakdowns?

MLUPS

0
4
8
12

10
100
1000

Nx



gerhard.wellein@rrze.uni-erlangen.de

# Optimization of data access
# Collision optimized layout: F(0:18,x,y,z,t)

A single cell is updated by 18 write accesses from 3 successive z-planes

Total amount of data for 3 successive z-planes:
$Mem_z \sim 3 * 2 * 19 * 8 * (xMax+2)*(yMax+2)$ Bytes

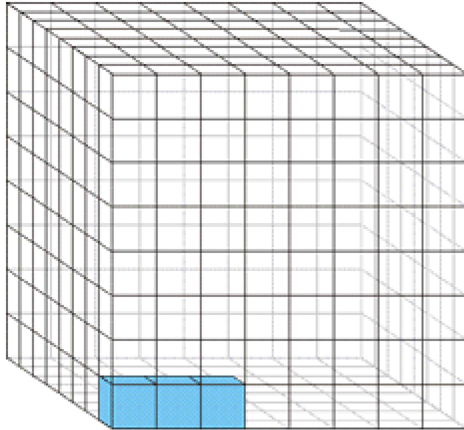
Cache lines must be reloaded if
$Mem_z \sim$ L2/L3 cache

xMax=yMax ~ 33
(1 MByte cache)

z

# Optimization of data access
# Spatial blocking

- Increase spatial locality by spatial blocking



```
real(8) F(0:18,0:xMax+1,…,0:1)
do zz=1,zMax,blcksize
 do yy=1,yMax,blcksize
  do xx=1,xMax,blcksize

   do z=zz,min(zMax,zz+blcksize-1)
    do y=yy,min(yMax,yy+blcksize-1)
     do x=xx,min(xMax,xx+blcksize-1)
          if( fluidcell(x,y,z) ) then
          …………
          endif
     enddo
    enddo
   enddo

  enddo
 enddo
enddo
```

Correct choice of **blcksize**?

- 2*19***blcksize**$^3$ Byte < L2/L3
  → **blcksize** ~8-10
- 2*19*3***blcksize**$^2$ Byte < L2/L3
  → **blcksize** ~25-30

# Optimization of data access
# Collision optimized layout: F(0:18,x,y,z,t)

Intel Xeon 3.4 GHz (1MB L2; 5.3 GByte/s) → 5-6 MFLUP/s

# Optimization of data access
# Propagation optimized layout : F(x,y,z,0:18,t)

```
double precision F(0:xMax+1,0:yMax+1,0:zMax+1,0:18,0:1)
do z=1,zMax
   do y=1,yMax
      do x=1,xMax
          if( fluidcell(x,y,z) ) then
             LOAD F(x,y,z, 0:18,t)
             Relaxation (complex computations)
             SAVE F(x  ,y  ,z  , 0,t+1)
             SAVE F(x+1,y+1,z  , 1,t+1)
             SAVE F(x  ,y+1,z  , 2,t+1)
             SAVE F(x-1,y+1,z  , 3,t+1)
             …
             SAVE F(x  ,y-1,z-1,18,t+1)
          endif
      enddo
   enddo
enddo
```

2*19 caches lines are touched for a single cell update – however they are accessed contiguously

High spatial data locality if 38 cache lines stay in the cache!

(38 * 128 Byte ~ 5 kByte << L2/L3 caches)

## Optimization of data access
## Impact of data layout: F(0:18,x,y,z,t) vs. F(x,y,z,0:18,t)

Intel Xeon 3.4 GHz (1MB L2; 5.3 GByte/s) → 5-6 MFLUP/s

Correct choice of data layout (PropOpt) improves performance by 2x-3x without the need for blocking

# Optimization of data access
## Propagation optimized layout: Cache thrashing

- Avoid powers of 2 in the leading dimensions of multidimensional array by array padding!

Use

`F(0:128,0:128,…)`

instead of

`F(0:127,0:127,…)`

Intel Xeon 3.4 GHz

CollOpt
PropOpt

MLUPS

Nx+2

xMax=yMax=zMax=Nx

# Optimization of data access
# Splitting innermost loops is essential!

⓿
```
! temporary arrays to hold pre-calculated values
real(8) :: ux(xMax), uy(xMax), uz(xMax), dens(xMax)
```

```
do k=1,zMax; do j=1,yMax
```

❶
```
    do i=1,xMax
       dens(i)=sum( F(i,j,k,:,t) )
       ux(i)=F(i,j,k,1,t)-F(i,j,k,5,t)+ …
       uy(i)= … ; uz(i)= …
    end do
```

❷
```
    ! update (relax and advect) first half of directions
    do i=1,xMax
       ! calculate common coefficients
       …
       sum15_1 = 3/4*coefA*(ux(i)+uy(i))
       sum15_2 = coefA*(ux(i)+uy(i))*(ux(i)+uy(i)) + usqnA
       F(i+1,j+1,k,1,t+1) = F(i,j,k,1,t)*coefA+sum15_1+sum15_2
       F(i-1,j-1,k,5,t+1) = F(i,j,k,5,t)*coefA-sum15_1+sum15_2
       …
    end do
```

❸
```
    ! update (relax and advect) second half of directions
    do i=1,xMax; similar calculations; end do
end do; end do
```

# Optimization of data access
# Splitting innermost loops is essential!

**Woodcrest:**
Intel Xeon 5150

2.67 GHz
4 MB L2 (2 cores)

FSB1333
(10.6 GB/s)

**Opteron:**
AMD Opteron 270

2.0 GHz
1 MB L2 (per core)

6.4 GB/s

Nx = Ny = Nz
Nx * 100 * 100

MLUPS

Nx

- Woodcrest (PropOpt; Split)
- Woodcrest (PropOpt; NoSplit)
- Opteron (PropOpt; Split)
- Opteron (PropOpt; NoSplit)
- Itanium2 (PropOpt; Split)
- Itanium2 (PropOpt; NoSplit)

**Itanium2:**
Intel Itanium2/Madison9M

1.6 GHz / 6 MB L3 / 8.5 GB/s


gerhard.wellein@rrze.uni-erlangen.de

# Summary – Efficient LBM implementation

- Efficient code implementation requires insight into memory hierarchy of modern processors
- Data layout analysis and/or spatial blocking is mandatory to optimize data transfer between main memory and processor
- Optimizing single processor performance and parallelization are tightly connected to the use of multi-core processors
- Not covered in this tutorial
  - Spatial blocking techniques for complex geometries
  - Temporal blocking techniques
  - Parallelization

# Literature

## Performance of LBM:

T. Pohl, N. Thürey, F. Deserno, U. Rüde, P. Lammers, G. Wellein, and T. Zeiser,
In: IEEE/ACM: *Proceedings of the IEEE/ACM SC2004 Conference* (2004), pp. 1-13.
*Performance Evaluation of Parallel Large-Scale Lattice Boltzmann Applications on Three Supercomputing Architectures*

T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, and U. Rüde, Parallel Processing Letters, Vol. 13, No. 4 (2003), pp. 549-560.
*Optimization and Profiling of the Cache Performance of Parallel Lattice Boltzmann Codes*

G. Wellein, T. Zeiser, G. Hager, and S. Donath, Computers & Fluids, Vol. 35 (2006), pp. 910-919.
*On the Single Processor Performance of Simple Lattice Boltzmann Kernels*

M. Schulz, M. Krafczyk, J. Tölke, E. Rank, in: M. Breuer, F. Durst, C. Zenger (Eds.), High Performance Scientific and Engineering Computing, Springer, Berlin (2001), pp. 115–122.
*Parallelization strategies and efficiency of CFD computations in complex geometries using lattice Boltzmann methods on high performance computers*

# Literature

## Performance of LBM:

J. Habich, Bachelor Thesis, Erlangen 2006.

*Improving computational efficiency of Lattice Boltzmann methods on complex geometries*

*S. Donath,* Bachelor Thesis, Erlangen 2004.

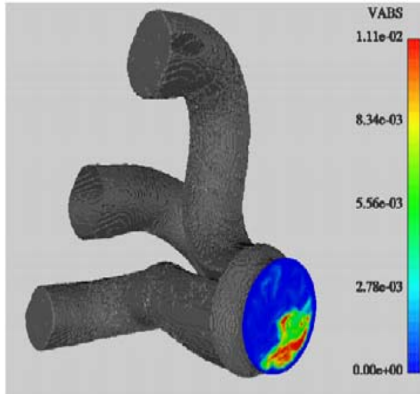
*On Optimized Implementations of the Lattice Boltzmann Method on Contemporary Architectures*

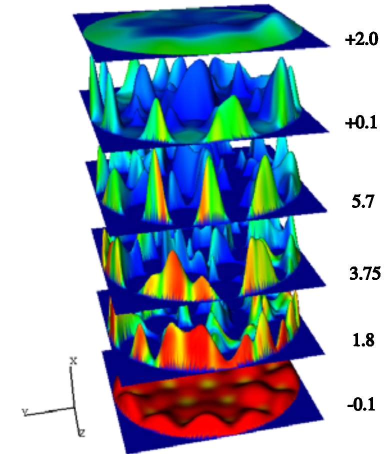
K. Iglberger, Bachelor Thesis, Erlangen 2003.
*Performance Analysis and Optimization of the Lattice Boltzmann Method in 3D*

**Find these and more interesting Bachelor and Master Theses at:**

http://www10.informatik.uni-erlangen.de/en/Publications/Theses/

Conference: July 25-28, 2006
Courses: July 24, 2006






# Thank you!

Bayerisches Staatsministerium für Wissenschaft, Forschung und Kunst

http://www.rrze.uni-erlangen.de/hpc/


Acknowledgement:

R. Vogelsang, R. Wolff (SGI)
W. Oed (CRAY)
Th. Schoenemeyer (NEC)

Acknowledgement:

M. Brehm et al. (LRZ)
U. Küster, P. Lammers (HLRS)
H. Cornelius, H. Bast, A. Semin (Intel)