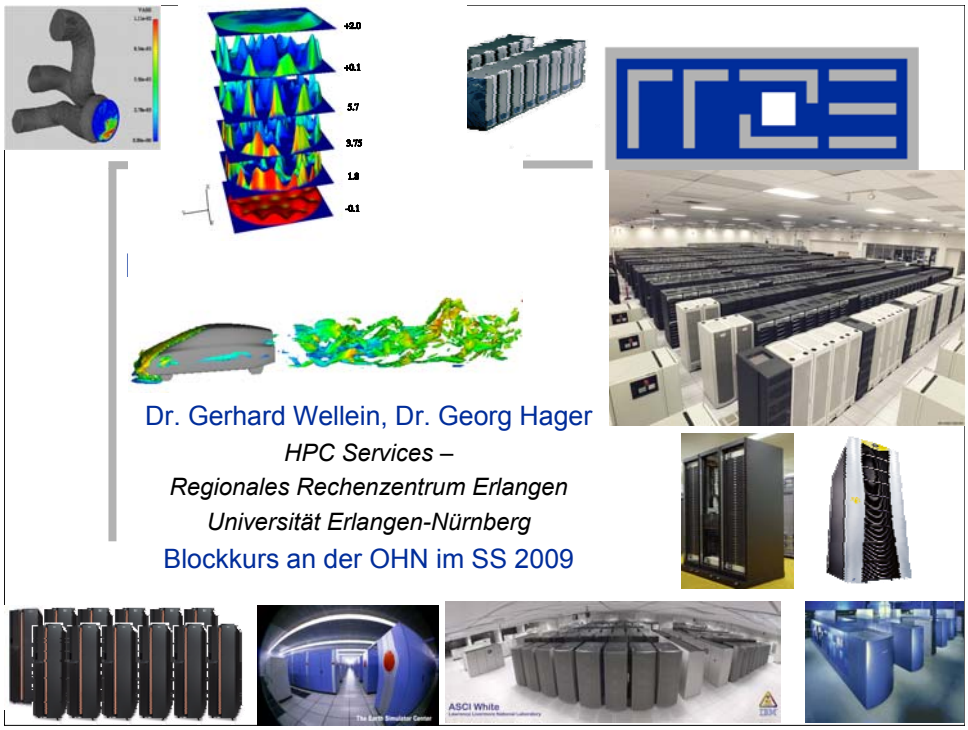


Dr. Gerhard Wellein, Dr. Georg Hager
*HPC Services –*
*Regionales Rechenzentrum Erlangen*
*Universität Erlangen-Nürnberg*
Blockkurs an der OHN im SS 2009

## Format of lecture

- **4 days course: 9.-12.3.**
- **16 units (90 minutes each) in total**
- **2 lectures in the morning**
  - **8:30-10:00**
  - **10:30-12:00**
- **2 tutorials in the afternoon (180 minutes)**
  - **13:30-16:30**
  - **Exercises will be performed at RRZE cluster**
- **13.3.: Visit to RRZE (9:00-11:00)**
- **Exam (18:00-19:30) ? 17.3. / 23.3. ?**

## Topics of lecture

- Day 1: Introduction & Single processor
- Day 2+3: (Shared memory) Parallelism, OpenMP & Multi-Core
- Day 4: Distributed memory parallelism, MPI & Clusters
- Presentations are available on the web:
  *http://www.blogs.uni-erlangen.de/hager/topics/OHN/*
- Exam: 60 minutes, no supporting material allowed



## Survey (Introduction & Single processor)

- Introduction
- Single Processor: Architecture & Programming
- Microprocessors & Pipelining
- Memory hierarchies of modern processors
- Literature

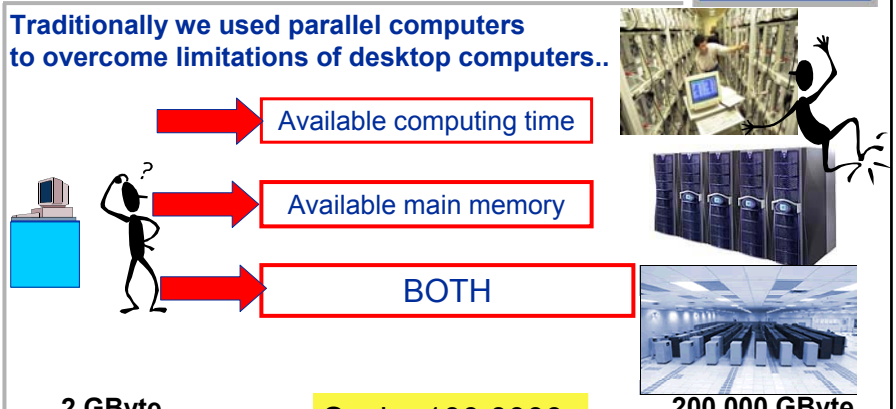
## Introduction

- **Parallel Computer (personal opinion):**
  *Multiple processors or compute nodes tightly connected*
- **Parallel Computing (personal opinion):**
  *Multiple processors solve cooperatively a single numerical problem*
- **(Massively) Parallel computing is the basic paradigm of modern supercomputer architectures and the emerging paradigm for desktop PCs as well!**
- **Distributed resources connected via GRID/Cloud technologies**
  - **are neither a parallel computer nor**
  - **parallel computing can be done on it**



## Introduction

**Traditionally we used parallel computers to overcome limitations of desktop computers..**




**2 GByte**
**10 GFlop/s**
PC / Desktop
(SINGLE CORE)

Scale: 100.0000

**200.000 GByte**
**1.000.000 GFlop/s**
Large scale supercomputers

**.. but with Multi-Core parallel computing is entering the desktop…**

## Last but not least
*TOP500 list*

- **Comprehensive & state-of-the-art survey: TOP500 list**
- **Top 500: Survey of the 500 most powerful supercomputers**
  - http://www.top500.org
  - Solve a large system of linear equations: `A x = b` („LINPACK“)
  - Published twice a year (ISC Heidelberg/Dresden, SC in USA)
  - Established in 1993 (CM5/1024): 60 GFlop/s (Top1)
  - June & Nov 2008 (Roadrunner): 1.105.000 GFlop/s (Top1)
  - Performance increase: 92,5 % p.a. !
- Performance measure: MFlop/s, GFlop/s, TFlop/s, PFlop/s
  - Number of FLOATING POINT operations per second
  - FLOATING POINT operations: double precision (64 bit) Add & Mult ops
  - $10^6$: MFlop/s; $10^9$: GFlop/s; $10^{12}$: TFlop/s; $10^{15}$: PFlop/s



## Last but not least
*Top500 list as of June 2009*

LINPACK [GFlop/s]

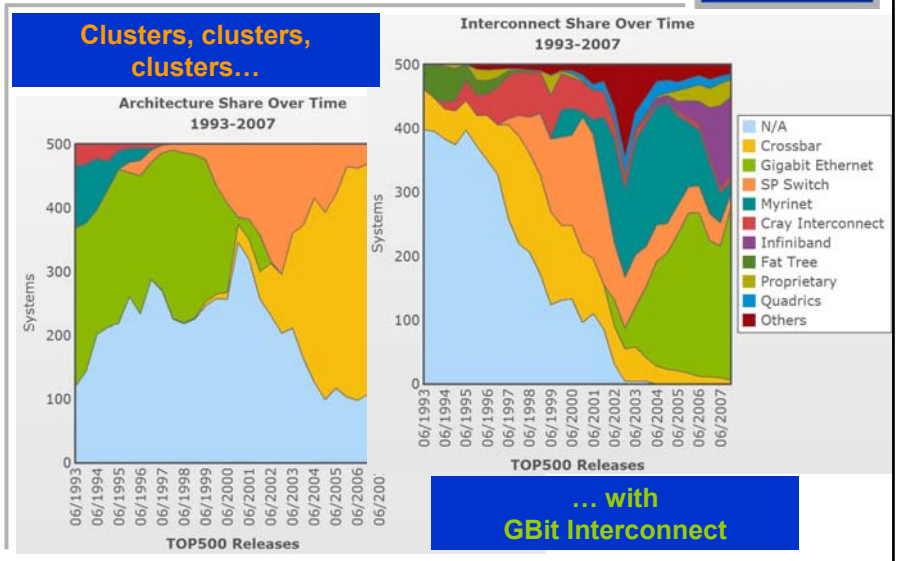
| | Rank | Site | | Computer | Country | Procs. | RMax | RPeak |
|---|---|---|---|---|---|---|---|---|
| 2.5 MW | 1 | LANL | IBM | Hybrid Opteron/ IBM Cell | U.S. | 129.600 | 1.105.000 | 1.456.700 |
| 7.0 MW | 2 | ORNL | CRAY | XT5 / AMD QC | U.S. | 150.152 | 1.059.000 | 1.381.400 |
| 2.1 MW | 3 | NASA / Ames | SGI | ICE: IB Cluster Xeon QC | U.S. | 51.200 | 487.010 | 608.830 |
| | 4 | DOE/NNSA/ LLNL | IBM | Blue Gene/L | U.S. | 212.992 | 478.200 | 596.378 |
| | 5 | Argone | IBM | Blue Gene/P | U.S. | 163.840 | 450.300 | 557.000 |
| | *6* | *Austin / TX* | *SUN* | *IB Cluster+ Quad-Core Opteron* | *U.S.* | *62.976* | *326.000* | *503.000* |
| | 7 | NERSC | CRAY | XT4 / AMD QC | U.S. | 38.642 | 266.300 | 355.510 |
| | 11 | FZ Jülich | IBM | Blue Gene/P | Germany | 65.536 | 180.000 | 222.822 |
| | 44 | LRZ Munich | SGI | Altix4700 | Germany | 9.728 | 56.520 | 62.260 |

**1 MW → 1.75 Mio € p.y.**

RRZE: 876 cores & 7.500 GFlop/s(RMax)

## Last but not least

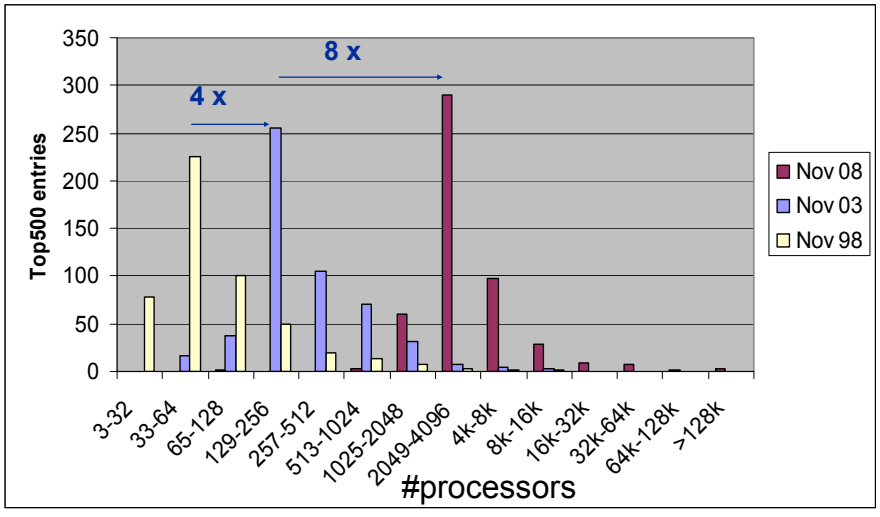
*Top500 list as of November 2007*






## Last but not least

*Top500 is going massively parallel (Nov. 2008)*



## It's a PetaFLOP!

*IBM/LANL break the barrier*


- **June 2008: $10^{15}$ FLOP/s for the first time!**
  **(Nov. 1997: ASI Red (Intel Paragon / P6) breaks the TFlop/s barrier!)**
- **Mankind (7 secs per FLOP per person) → >1 Year to do $10^{15}$ FLOP**
- **It`s not only the first PetaFLOP system, it`s heterogeneous!**

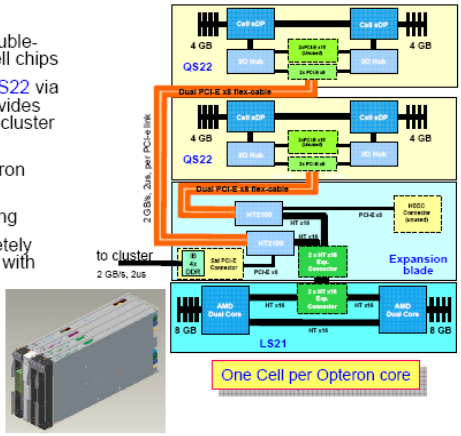


## It's a PetaFLOP!

*IBM/LANL break the barrier*

### A Roadrunner TriBlade node integrates Cell and Opteron blades

- QS22 is a future IBM Cell blade containing two new enhanced double-precision (eDP/PowerXCell™) Cell chips
- Expansion blade connects two QS22 via four PCI-e x8 links to LS21 & provides the node's ConnectX IB 4X DDR cluster attachment
- LS21 is an IBM dual-socket Opteron blade
- 4-wide IBM BladeCenter packaging
- Roadrunner Triblades are completely diskless and run from RAM disks with NFS & Panasas only to the LS21
- Node design points:
  - *One Cell chip per Opteron core*
  - *~400 GF/s double-precision & ~800 GF/s single-precision*
  - *16 GB Cell memory & 16 GB Opteron memory*






Los Alamos NATIONAL LABORATORY EST.1943

Operated by the Los Alamos National Security, LLC for the DOE/NNSA

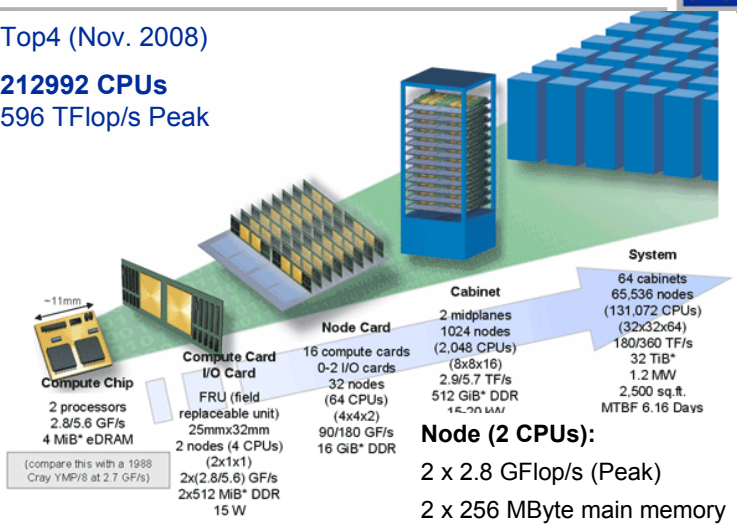
ASC NNSA IBM

http://www.lanl.gov/orgs/hpc/roadrunner/pdfs/Koch%20-%20Roadrunner%20Overview/RR%20Seminar%20-%20System%20Overview.pdf

## Introduction
*IBM BlueGene/L*

Top4 (Nov. 2008)

**212992 CPUs**
596 TFlop/s Peak




**Node (2 CPUs):**

2 x 2.8 GFlop/s (Peak)

2 x 256 MByte main memory

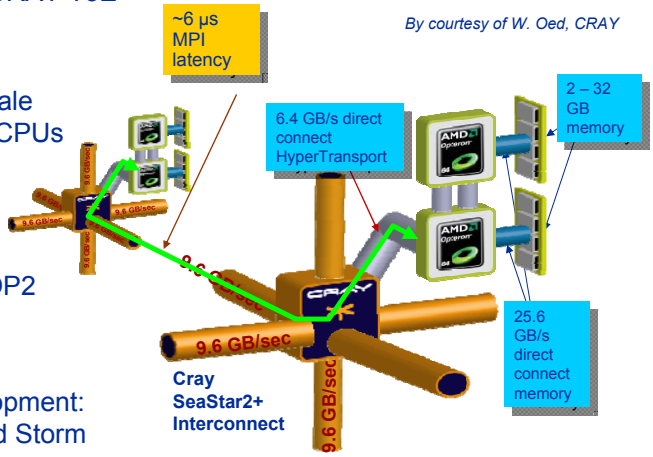
180 MByte/s in each dir. (Interconnect)



## Introduction
*CRAY XT3*

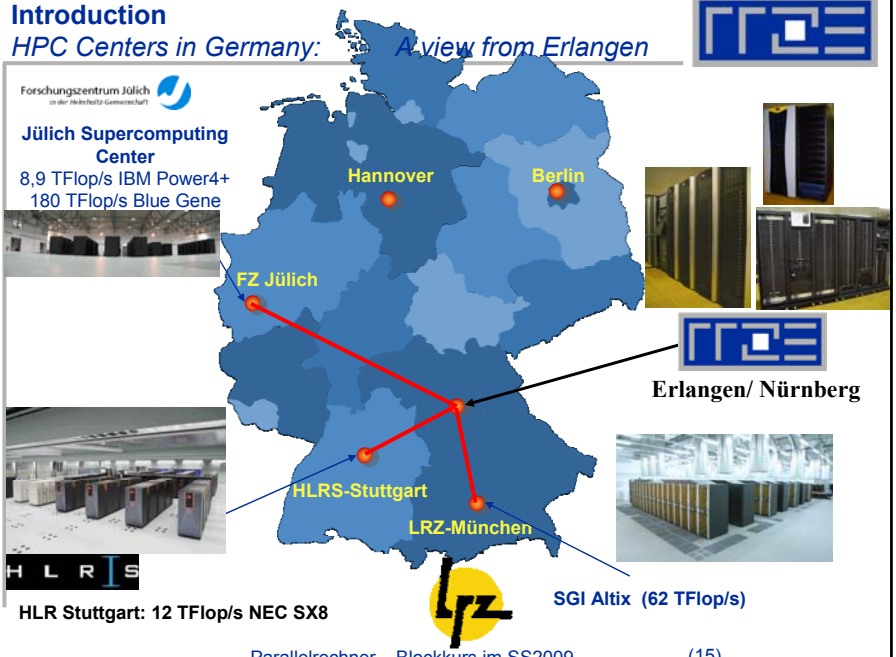
- 5th generation of CRAY MPP systems (1 node = 2 QC chips)
- Successor of CRAY T3E
- System is designed to scale to 1.000.000s CPUs
- Oak Ridge: TOP2
- Original development: 40 TFlop/s Red Storm
- OS: Linux micro kernel

*By courtesy of W. Oed, CRAY*



## Introduction

*HPC Centers in Germany: A view from Erlangen*





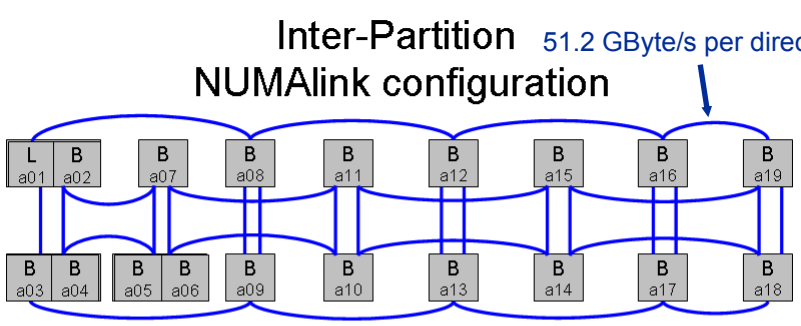

## Introduction

*HLRB II@LRZ Munich SGI Altix 4700 / 9728 cores*


by courtesy of LRZ

## Introduction

*HLRB-II: 2D-Torus between 19 „compute nodes“*




Each grey box:
- 512 core partition (SSI)
- L login, B batch

Each line represents:
- 2 NUMAlink4 planes with 16 cables (total)
- each cable: 2 * 3.2 GB/s
- high density partitions share cables in each line

**Each compute node:**

**512 processors**

**2000 GByte main memory**

by courtesy of LRZ

## Introduction

*System configuration: HLRB-II*

- **Phase 1**
  - 4096 Intel Madison9M cores
    - single core = 4096 cores
    - 6 MByte L3
    - FSB 533: 1.33 Byte/Flop
  - 17 TByte memory
  - 26.2 TF Peak
  - 24.5 TF Linpack
  - 8.2 TF aggreg. weighted performance on LRZ BM
  - 256 core single system images
  - ccNUMA
  - Power: ~ 1 MW

- **Phase 2**
  - 9728 Intel Montecito Cores
    - Dual Core
    - 9 Mbyte L3 per core
    - FSB 533
  - 39 TByte Memory
  - 62.2 TF Peak
  - 53.3 TF Linpack
  - 16 TF aggreg. weighted performance on LRZ BM
  - 512 core single system image
  - ccNUMA
  - Power: ~1.1 MW

**NUMAlink4 Interconnect (2D Torus based on 256/512 core fat trees)**
**globally addressable with low latency**
**High bandwidth/Low Latency (2x6.4 GByte/s link bandwidth, 1-6µs latency)**
**2 x 204 GByte/s between nodes**

by courtesy of LRZ

**Introduction**
*RRZE "Woody-Cluster"*

- 860 Intel Xeon5160 processor cores
  - Core2Duo architecture
  - 3.0 GHz →12 GFlop/s per core
  - 4 cores per compute node
  - Installation: November 2006
- Peak performance: 10400 GFlop/s
- Main memory:
  - 2 GByte per core
  - 1720 GByte in total
- Infiniband network
  - Voltaire DDRx 216 ports
  - 10 GBit/s+ per node & direction
- OS: SuSe Linux: SLES9


- Parallel filesystem: 15 TByte
- NFS filesystem: 15 TByte

**Power consumption > 100 kW**



**Introduction**
*RRZE: Other Compute Resources (2003-2005)*

## *transtec compute cluster*

- 216 2-way compute nodes:
  - 86 nodes: Intel Xeon 2.6 GHz; FSB533
  - 64 nodes: Intel Xeon 3.2 GHz; FSB800
  - **66 nodes: Intel Xeon 2.66 GHZ; Dual-Core**
- 25 4-way compute nodes:
  - AMD Opteron270 (2.0 GHz); **Dual-Core**
- GBit Ethernet network
- Infiniband: 24 nodes (SDR) + 66 nodes (DDR)
- 5,5+13 TByte Disc Space
- Installation: 4/ 2003 ; Upgrades: 12 / 2004, Q4/2005, Q3/2007

## *Compute Servers*

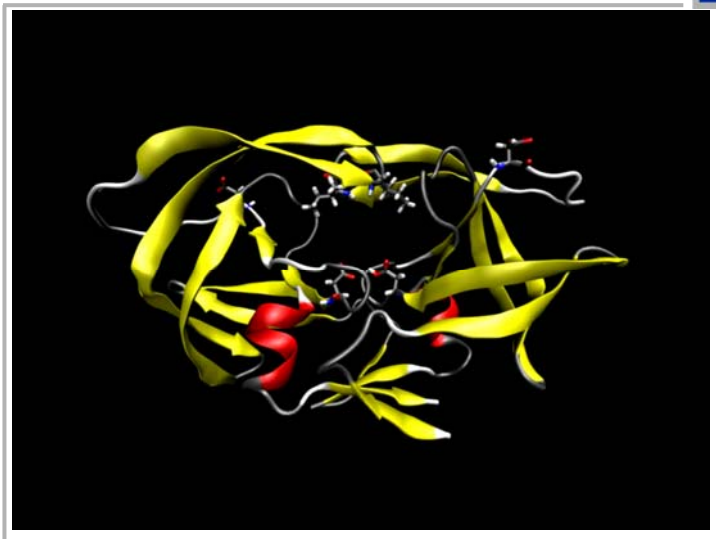
SGI Altix3700

- 32 Itanium2 1.3 GHz
- 128 GByte Memory
- 3 TByte Disc Space
- Inst.: 11 / 2003

SGI Altix330

- 16 Itanium2 1.5 GHz
- 32 GByte Memory
- Inst.: 3 / 2006

## Introduction
*MD Simulation of HIV protease dynamics*

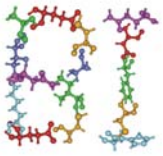


**Real time:**
**10 ns**

**Compute time:**

**18.000 CPU-hrs**

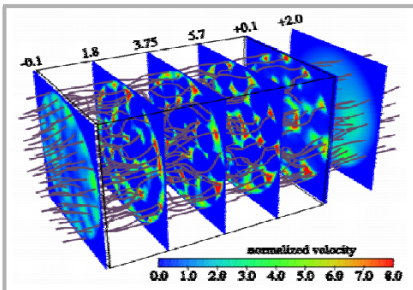
**8 CPUs – 90 days**

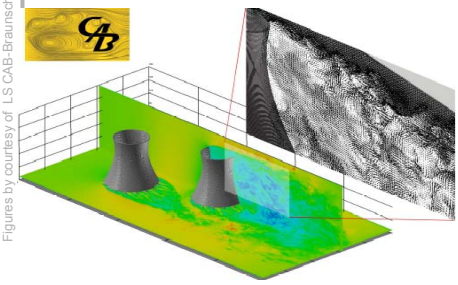


Courtesy: Prof. Sticht, Bio-Informatics, Emil-Fischer Center, FAU

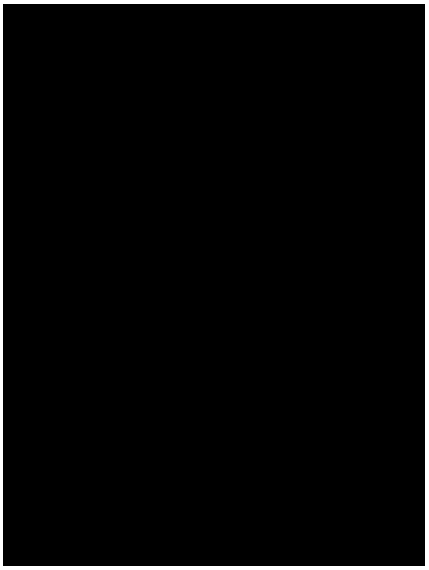


## Introduction
*Lattice Boltzmann flow solvers*

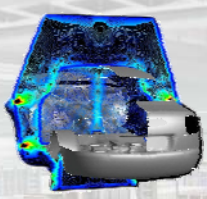


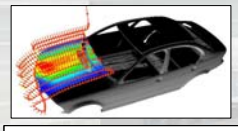



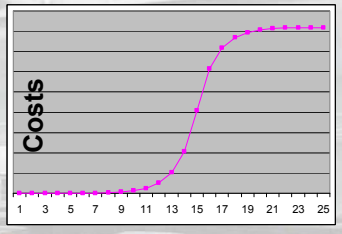
Figures by courtesy of LS CAB-Braunschweig, Thomas Zeiser, N. Thuerey



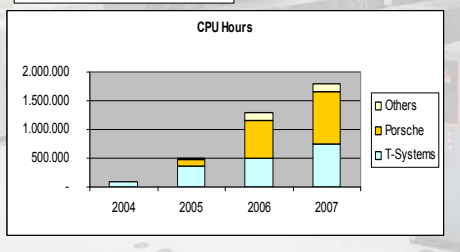
## Introduction
*Industrial Usage 2004-2007 (HLRS)*








Product cycle



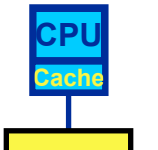


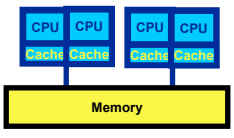


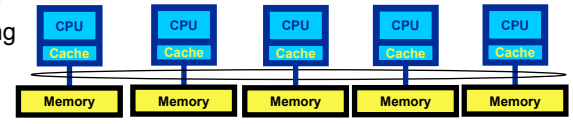
H L R S

## Introduction
*How to build faster computers*

1. **Increase performance / throughput of CPU**
   a) Reduce cycle time, i.e. increase clock speed
   b) Increase throughput, i.e. superscalar
2. **Improve data access time**
   a) Increase cache size
   b) Improve main memory access (bandwidth & latency)
3. **Introduce multi-(core) processing**
   a) Requires shared-memory parallel programming
   b) Shared/separate caches
   c) Possible memory access bottlenecks
4. **Use external parallelism**
   **"Cluster" of computers tightly connected**
   1. Almost unlimited scaling of memory and performance
   2. Distributed-memory parallel programming

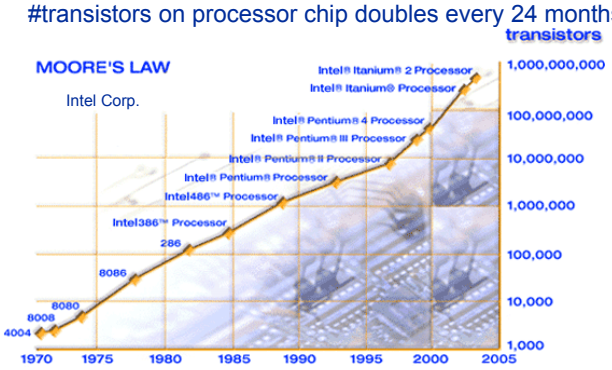


## Introduction
*Faster computers: Complexity and clock speed*

- **1965 G. Moore claimed**
  #transistors on processor chip doubles every 24 months




**This trend is currently changing: see multi-core**

- **Processor speed grew roughly at the same rate**
  My computer: 350 MHz (1998) – 3,000 MHz (2004)
  Growth rate: 43 % p.a. → doubles every 24 months
- **Problem: Power dissipation (see RRZE systems…)**



## Introduction
*Faster computers: Clock speed & Superscalarity*

- **High clock speeds require pipelining of functional units**:
  E.g. it may take 30 cycles to perform a single instruction on a Intel P4
- **Superscalarity – multiple functional units work in parallel:**
  E.g. most processors can perform
  - 4-6 instructions per cycle
  - 2-4 floating point operations per cycle

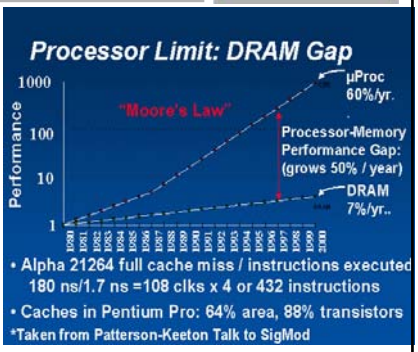
- High complexity of computer architectures
  - CISC → RISC
  - Out-of Order Execution
  - Introduction of new architectures:
    - EPIC/Itanium with fully in-order instruction issue

- **Manual optimization of code is mandatory**
- **Memory bandwidth imposes restrictions for most applications**

## Introduction

*Faster computers: Clock speed vs. DRAM 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
  - Introducing memory hierarchies (caches) – Complex optimization of code




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



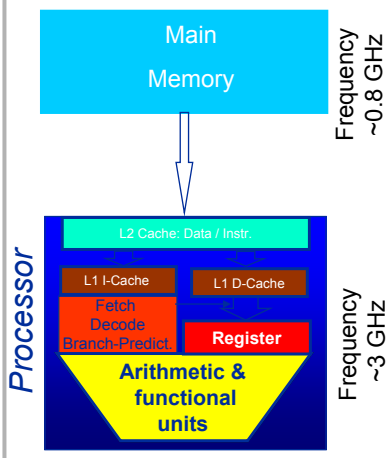
## Microprocessors & Pipelining

## Architecture of modern microprocessors
*History*

- In the beginning: **Complex Instruction Set Computers (CISC)** :
  - Powerful & complex instructions, e.g: A=B*C: 1 instruction
  - Instruction set is close to high-level programming language
  - Variable length of instructions - Save storage!
- Mid 80´s: **Reduced Instruction Set Computer (RISC)** evolved:
  - Fixed instruction length; enables pipelining and high clock frequencies
  - Uses simple instructions, e.g.: A=B*C is split into at least 4 operations (LD B, LD C, MULT A=B*C, ST A)
  - **Nowadays: Superscalar RISC processors**
  - IA32 (Core2, Athlon, Opteron): Compiler still generates CISC instructions; but processor core: RISC like
  - RISC is still implemented in most dual- / quad-core CPUs
- ~2001: **Explicitly Parallel Instruction Computing (EPIC)** introduced
  - Compiler builds large group of instruction to be executed in parallel
  - First processors: **Intel Itanium1/2 using the IA64** instruction set.



## Architecture of modern microprocessors
*Cache based microprocessors (e.g. Intel P4, AMD)*

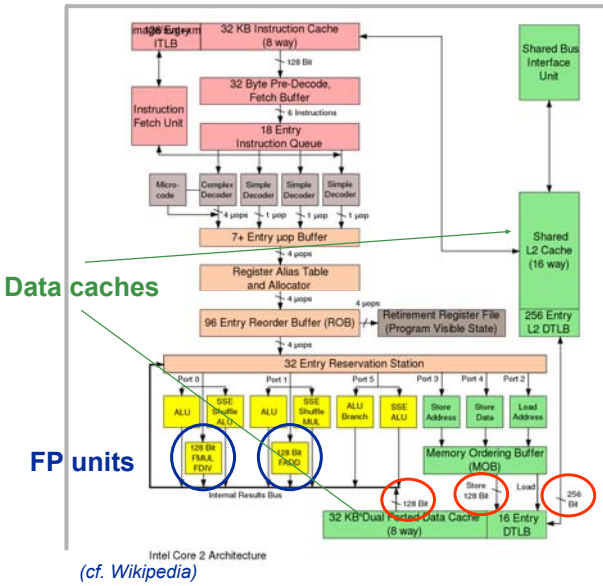



Processor is built from:

- Arithmetic & functional units, e.g. Multiply-unit, Integer-units, MMX, …
- These units can only use operands resident in the registers
- Operands are read (written) by load (store) units from **main memory/caches** to registers
- Caches are fast but small pieces of memory (5-10 times faster than main memory)
- A lot of additional logic: e.g. branch prediction

## Architecture of modern microprocessors

*Architectural block diagram – Intel Core 2*




Intel Core 2 Architecture

*(cf. Wikipedia)*

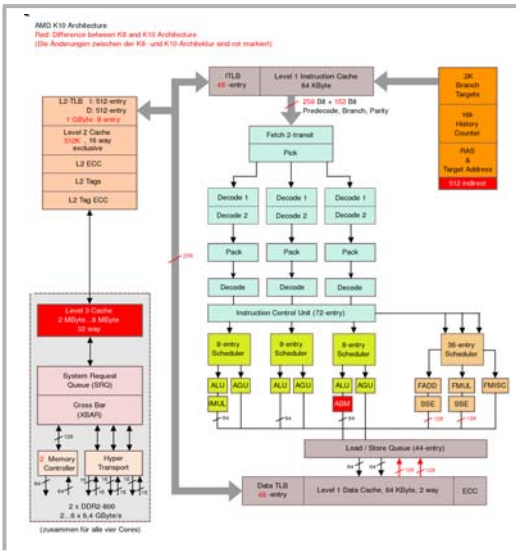
Intel Core 2 architecture

- Successor of Netburst
- Reduced pipeline length
- Improved instruction issue
- Double FP performance
- Instruction level parallelism: 4µops issue/cycle
- Desktop variants (65 nm)
  - Core2Duo → „Conroe“
  - Core2Quad→ „Kentsfield“
  - Top bin: 3 GHz



## Architecture of modern microprocessors

*Architectural block diagram –AMD K10*




*(cf. Wikipedia)*

AMD K10 architecture

- To be used in first native Quad-Core („Barcelona“)
- L3 cache shared by all 4 cores
- Latest implementation (45nm): Shanghai
- L3 up to 6 MB
- Top bin: 3.0 GHz

## Architecture of modern microprocessors

*Pipelining of arithmetic/functional units*

- **Split complex operations (e.g. multiplication) into several simple / fast sub-operations (stages)**
- **Makes short cycle time possible (simpler logic circuits), e.g.:**
  - floating point multiplication takes 5 cycles, but
  - processor can work on 5 different multiplications simultan.
  - one result at each cycle after the pipeline is full
- **Drawback:**
  - Pipeline must be filled - startup times (#Operations >> pipeline steps)
  - Efficient use of pipelines requires large number of independent instructions → instruction level parallelism
  - Requires complex instruction scheduling by compiler/hardware – software-pipelining / out-of-order
- **Vector processors use large numbers of parallel pipelines!**



## Pipelining

*5-stage Multiplication-Pipeline: A(i)=B(i)*C(i) ; i=1,...,N*

| Cycle: | 1 | 2 | 3 | 4 | 5 | 6 | ... | N+4 |
|---|---|---|---|---|---|---|---|---|
| **Operation** | | | | | | | | |
| Separate Mant. / Exp. | B(1) C(1) | B(2) C(2) | B(3) C(3) | B(4) C(4) | B(5) C(5) | B(6) C(6) | ... | |
| Mult. Mantissa | | B(1) C(1) | B(2) C(2) | B(3) C(3) | B(4) C(4) | B(5) C(5) | ... | |
| Add. Exponents | | | B(1) C(1) | B(2) C(2) | B(3) C(3) | B(4) C(4) | ... | |
| Normal. Result | | | | A(1) | A(2) | B(3) C(3) | ... | |
| Insert Sign | | | | | A(1) | A(2) | ... | A(N) |

First result is available after 5 cycles (=latency of pipeline)!

## Pipelining
*Speed-Up and Throughput*

- In general (m-stage pipe /pipeline depth: m)

  Speed-Up:

  $T_{seq} / T_{pipe} = (m*N) / (N+m-1) \sim m$ for large N (>>m)

  Throughput (=Results per Cycle):

  $N / T_{pipe}(N) = N / (N+m-1) = 1 / [\ 1+(m-1)/N\ ] \sim 1$ for large N

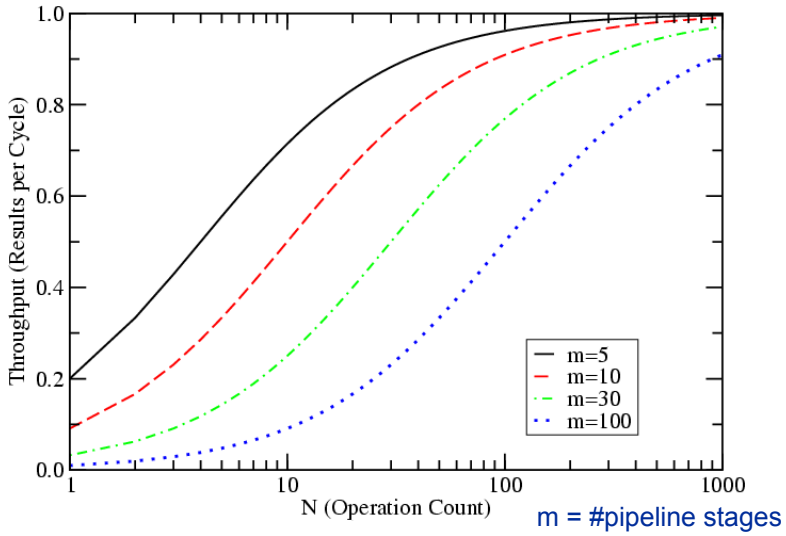
- Number of independent operations ($N_C$) required to achive $T_p$ results per cycle:

  $T_p = 1 / [\ 1+(m-1)/N_C\ ]$ ⇒ $N_C = T_p\ (m-1) / (1- T_p)$

  $T_p = 0.5$ ⇒ $N_C = m-1$



## Pipelining
*Throughput as function of pipeline stages*




m = #pipeline stages

## Pipelining

*Software pipelining*

- Example:

*Fortran Code:*
```
do i=1,N
  a(i) = a(i) * c
end do
```

Assumption:

Instructions block execution if operands are not available

```
load a[i]
mult a[i] = c,a[i]
store a[i]
branch.loop
```

Load operand to register (4 cycles) ← Latencies
Multiply a(i) with c (2 cycles); a[i],c in registers
Write back result from register to mem./cache (2 cycles)
Increase loopcounter as long i less equal N (0 cycles)

*Simple Pseudo Code:*
```
loop:   load a[i]
        mult a[i] = c, a[i]
        store a[i]
        branch.loop
```

→

*Optimized Pseudo Code:*
```
loop:   load a[i+6]
        mult a[i+2] = c, a[i+2]
        store a[i]
        branch.loop
```



## Pipelining

*Software pipelining*

a[i]=a[i]*c; N=12

| | Naive instruction issue | Optimized instruction issue | | | |
|---|---|---|---|---|---|
| Cycle 1 | load a[1] | load a[1] | | | Prolog |
| Cycle 2 | | load a[2] | | | |
| Cycle 3 | | load a[3] | | | |
| Cycle 4 | | load a[4] | | | |
| Cycle 5 | mult a[1]=c,a[1] | load a[5] | mult a[1]=c,a[1] | | |
| Cycle 6 | | load a[6] | mult a[2]=c,a[2] | | |
| Cycle 7 | store a[1] | load a[7] | mult a[3]=c,a[3] | store a[1] | |
| Cycle 8 | | load a[8] | mult a[4]=c,a[4] | store a[2] | |
| Cycle 9 | load a[2] | load a[9] | mult a[5]=c,a[5] | store a[3] | Kernel |
| Cycle 10 | | load a[10] | mult a[6]=c,a[6] | store a[4] | |
| Cycle 11 | | load a[11] | mult a[7]=c,a[7] | store a[5] | |
| Cycle 12 | | load a[12] | mult a[8]=c,a[8] | store a[6] | |
| Cycle 13 | mult a[2]=c,a[2] | | mult a[9]=c,a[9] | store a[7] | Epilog |
| Cycle 14 | | | mult a[10]=c,a[10] | store a[8] | |
| Cycle 15 | store a[2] | | mult a[11]=c,a[11] | store a[9] | |
| Cycle 16 | | | mult a[12]=c,a[12] | store a[10] | |
| Cycle 17 | load a[3] | | | store a[11] | |
| Cycle 18 | | | | store a[12] | |
| Cycle 19 | | | | | |

T= 96 cycles

T= 19 cycles

## Pipelining
*Efficient use*

- Software pipelining can be done by the compiler, but efficient reordering of the instructions requires deep insight into application (data dependencies) and processor (latencies of functional units)
- (Potential) dependencies within loop body may prevent efficient software pipelining, e.g.:

*No dependency:*

```
do i=1,N
  a(i) = a(i) * c
end do
```

***Dependency:***

```
do i=2,N
  a(i) = a(i-1) * c
end do
```

*Pseudo-Dependency:*

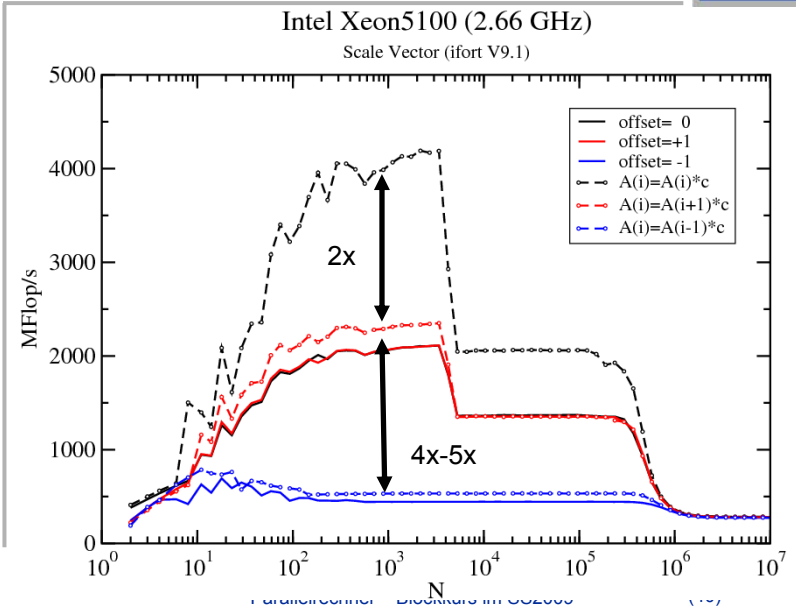
```
do i=1,N-1
  a(i) = a(i+1) * c
end do
```

*General version (offset as input parameter):*

```
do i=max(1-offset,1),min(N-offset,N)
  a(i) = a(i-offset) * c
end do
```

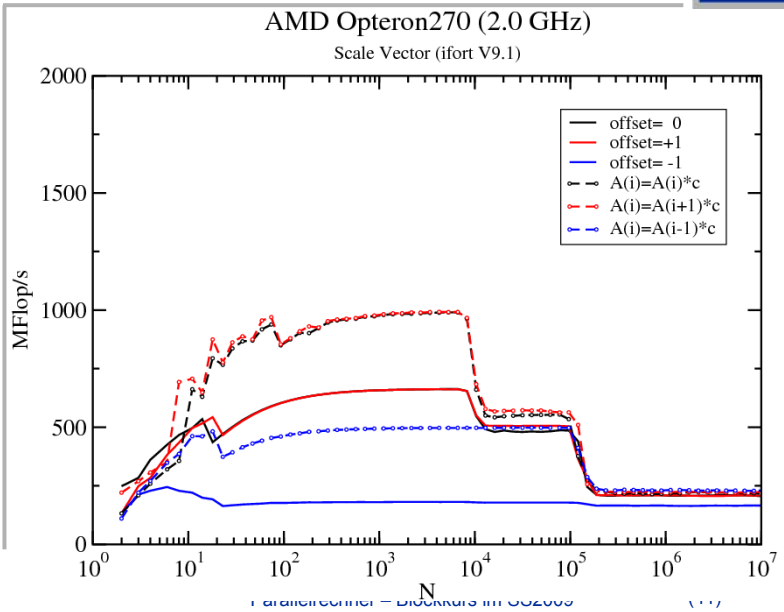


## Pipelining
*Data dependencies*



## Pipelining

*Data dependencies*




## Pipelining

*Data dependencies*

a[i]=a[i-1]*c; N=12

| | Naive instruction issue | Optimized instruction issue | | |
|---|---|---|---|---|
| Cycle 1 | load a[1] | load a[1] | | |
| Cycle 2 | | | | |
| Cycle 3 | | | | Prolog |
| Cycle 4 | | | | |
| Cycle 5 | mult a[2]=c,a[1] | mult a[2]=c,a[1] | | |
| Cycle 6 | | | | |
| Cycle 7 | store a[2] | mult a[3]=c,a[2] | store a[2] | |
| Cycle 8 | | | | |
| Cycle 9 | load a[2] | mult a[4]=c,a[3] | store a[3] | |
| Cycle 10 | | | | |
| Cycle 11 | | mult a[5]=c,a[4] | store a[4] | |
| Cycle 12 | | | | |
| Cycle 13 | mult a[3]=c,a[2] | mult a[6]=c,a[5] | store a[5] | Kernel |
| Cycle 14 | | | | |
| Cycle 15 | store a[3] | mult a[7]=c,a[6] | store a[6] | |
| Cycle 16 | | | | |
| Cycle 17 | load a[3] | mult a[8]=c,a[7] | store a[7] | |
| Cycle 18 | | | | |
| Cycle 19 | | mult a[9]=c,a[8] | store a[8] | |
| | T= 96 cycles | T= 26 cycles | | |

Length of MULT pipeline determines throughput

## Pipelining
*Further potential problems*

- Typical number of pipeline stages: 2-5 for the hardware pipelines on modern CPUs (e.g. Intel Core architecture: 5 cycles for FP MULT)
- Modern microprocessors do not provide pipelines for *div* / *sqrt* or *exp* / *sin* !

  Example: Cycles per Floating Operation (8-Byte) for Xeon/Netburst

| Operation | y=a+y (y=a*y) | y=a/y | y=dsqrt(y) | y=sin(y) |
|---|---|---|---|---|
| Latency | 4* | 70* | 70* | ~160-180 |
| Throughput | 2* | 70* | 70* | 130 |
| Cycles/Operation | 1* | 35* | 35* | 130 |

- Reduce number of complex operations if necessary.
- Replace function call with a table lookup if the function is frequently computed for a few different arguments only.

* Using SIMD instructions (SSE2)



## Pipelining
*Further potential problems*

- Data dependencies: Compiler can not resolve aliasing conflicts!

```
void subscale( A , B )
….
for (i=0;…) A(i) = B(i-1)*c
```

  In C/ C++ the pointers of A and B may point to the same memory location → see above
  - Tell compiler if your are never using aliasing ( -fno-alias for Intel Compiler)
- Subroutine/function calls within a loop

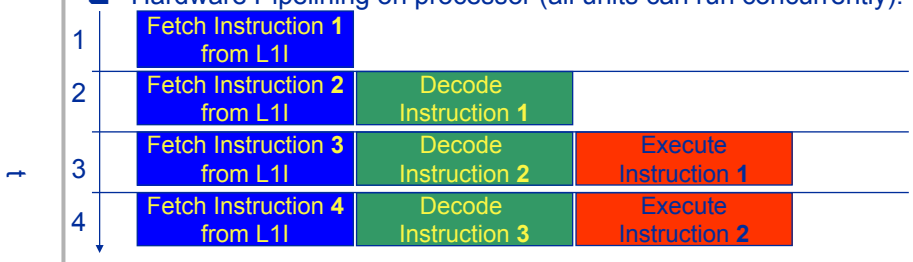
```
do i=1, N
    call elementprod(A(i), B(i), partsum)
    sum=sum+partsum
enddo
…
function elementprod( a, b, sum)
…
sum=a*b
```

  Inline short subroutine/functions!

## Pipelining
*Instruction pipeline*

- Besides the arithmetic and functional unit, the instruction execution itself is pipelined also, e.g.: one instruction performs at least 3 steps:


- Hardware Pipelining on processor (all units can run concurrently):

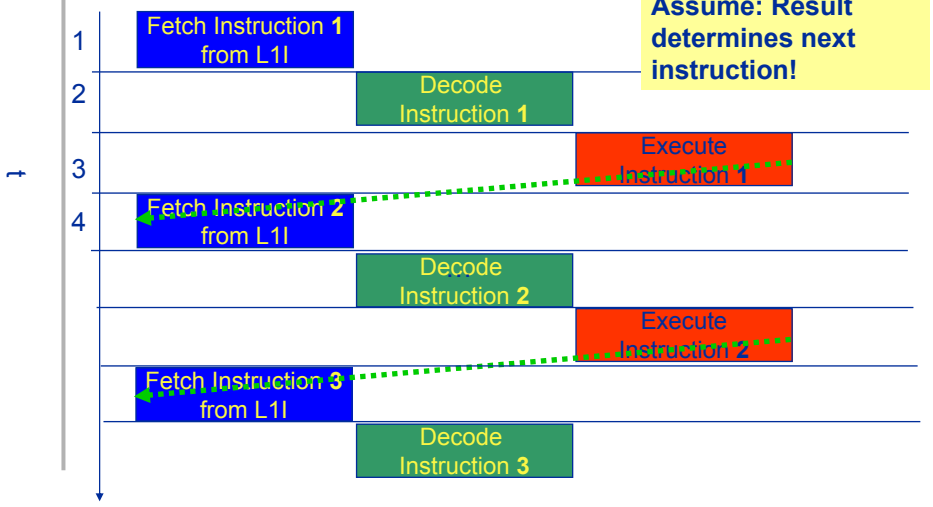



- Branches can stall this pipeline! (Speculative Execution, Predication)
- Each Unit is pipelined itself (cf. Execute=Multiply Pipeline)



## Pipelining
*Instruction pipeline*

- Problem: Unpredictable branches to other instructions



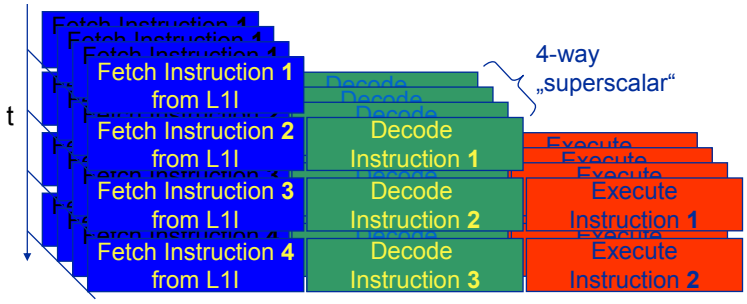
## Pipelining
*Superscalar Processors*

- Superscalar Processors can run multiple Instruction Pipelines at the same time!
- Parallel hardware components / pipelines are available to
  - fetch / decode / issues multiple instructions per cycle (typically 3 – 6 per cycle)
  - load (store) multiple operands (results) from (to) cache per cycle (typically 2-4 8-byte words per cycle)
  - perform multiple integer / address calculations per cycle (e.g. 6 integer units on Itanium2)
  - perform multiple floating point operations per cycle (typically 2 or 4 floating point operations per cycle)
- On superscalar RISC processors out-of order execution hardware is available to optimize the usage of the parallel hardware



## Pipelining
*Superscalar Processors*

- Multiple units enable use of Instrucion Level Parallelism (ILP):




- Issuing m concurrent instructions per cycle: m-way superscalar
- Modern processors are 3- to 6-way superscalar & can perform 2 or 4 floating point operations per cycles

## Pipelining
*Superscalar Processor*

- **Example: Calculate norm of a vector on a CPU with 2 MultAdd (MADD) units**
- **Naive version:**

```
t=0
do i=1,n
   t=t+a(i)*a(i)
end do
```

2 FP Mult/Add units cannot be busy at the same time because of dependency in summation variable t


- **2nd MADD has to wait for the first to be completed, although in principle two independent MADD could be done**



## Pipelining
*Superscalar Processor*

- Optimized version:

```
t1=0
t2=0
do I=1,N,2
   t1=t1+a(i)*a(i)
   t2=t2+a(i+1)*a(i+1)
end do
t=t1+t2
```

Most compilers can do those optimizations automatically (if you allow them to do so)!

*Two independent „instruction streams” can be processed by two separate FP Mult/Add units!*

| | |
|---|---|
| R1 = MADD(R1,A(I)) | R2 = MADD(R2,A(I+1)) |
| R1 = MADD(R1,A(I+2)) | R2 = MADD(R2,A(I+3)) |
| … | |

## Pipelining
*Superscalar PCs*

| | Intel P4/Netburst | Intel Core |
|---|---|---|
| FP units | 1 MULT & 1 ADD pipeline | |
| Width of operands | 128 Bit | |
| FP ops/unit | 2 DP or 4 SP | |
| Throughput | 2 cycles | 1 cycle |
| Max. FP ops/cycle | 2 DP or 4 SP | 4 DP or 8 SP |
| Latency of FP units (FPMULTD) | 7 cycles | 5 cycles |

DP: double precision, i.e. 64 bit operands (double)

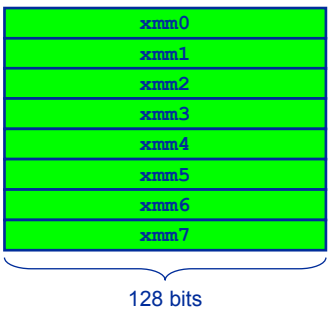
SP: single precision, i.e. 32 bit operands (float)

Throughput: Repeat rate of instruction issue,
e.g. 1 cycle → in each cycle an new operation can be started



## Pipelining
*Superscalar PCs – SSE*

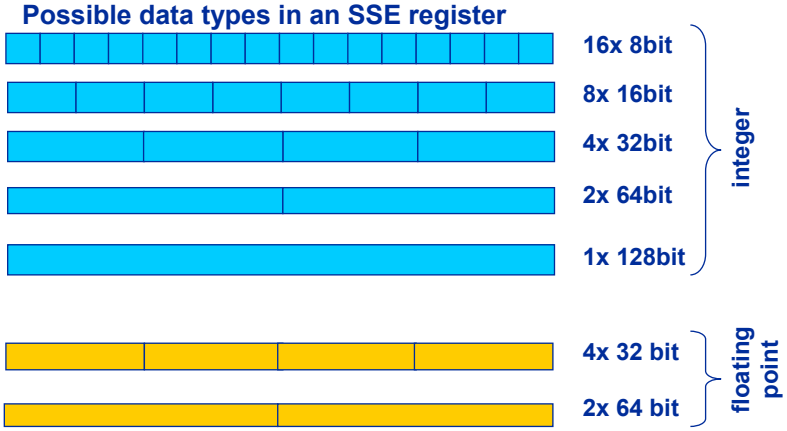
- **Streaming SIMD Extensions (SSE) instructions must be used to operate on the 128 bit registers**
  - **Register Model:**




- **Each register can be partitioned into several integer or FP data types**
  - **8 to 128-bit integers**
  - **single (SSE) or double precision (SSE2) floating point**
- **SIMD instructions can operate on the lowest or all partitions („Packed SSE“) of a register at once**

## Pipelining
*Superscalar PCs – SSE*

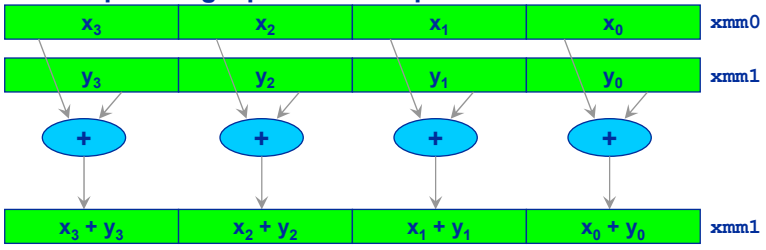
- **Possible data types in an SSE register**






## Pipelining
*Superscalar PCs – SSE*

- **Example: Single precision FP packed vector addition**




- **Four single precision FP additions with one single instruction**
- **Packed SSE → Code vectorizatzion is a must**
- **Vectorization only possible if data are independent**
- **Automatic vectorization by compiler (appropriate compiler flag needs to be set) or forced by programmer (via directive)**
  - **"`LOOP WAS VECTORIZED`" messages** ✓

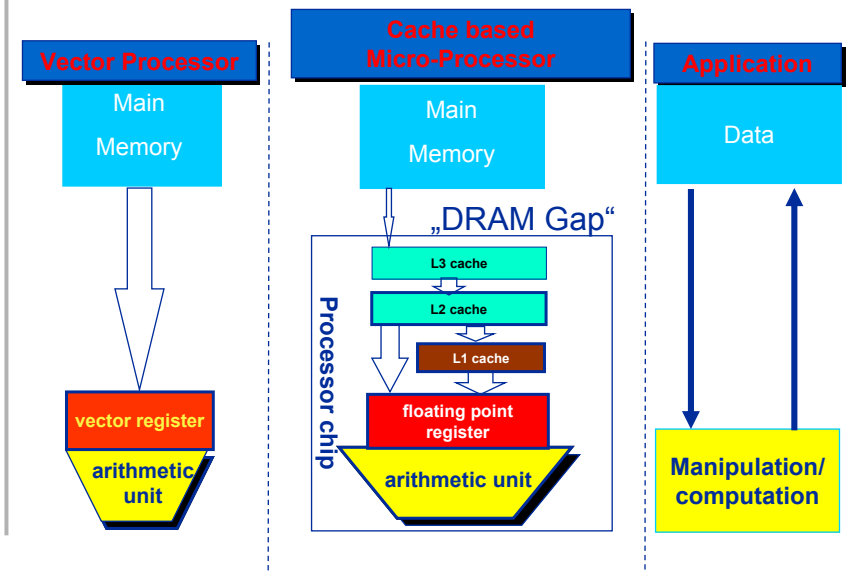
## Pipelining
*Efficient Use of Pipelining*

- **Efficient use of pipelining/ILP requires intelligent compilers**
  - Rearrangement of instructions to hide latencies
  - „Software pipelining“
  - Remove interdependencies that block parallel execution
- **Programmer should**
  - Avoid unpredictable branches (stop & restart of instruction pipeline!)
  - Avoid Data dependencies (if possible)
  - Tell compiler that instructions are independent (e.g. do not use pointer aliasing: `-fno-alias` with intel compiler)
- **Long pipelines are inefficient for very small loops**
  - Pipeline must be filled, i.e. long start-up times (latency!)
- **Summary:**
  - Large number of independent / parallel instruction is mandatory to efficiently use pipelined, superscalar processors.
  - Most of the work can be done by the compiler, however programmer must provide reasonable code



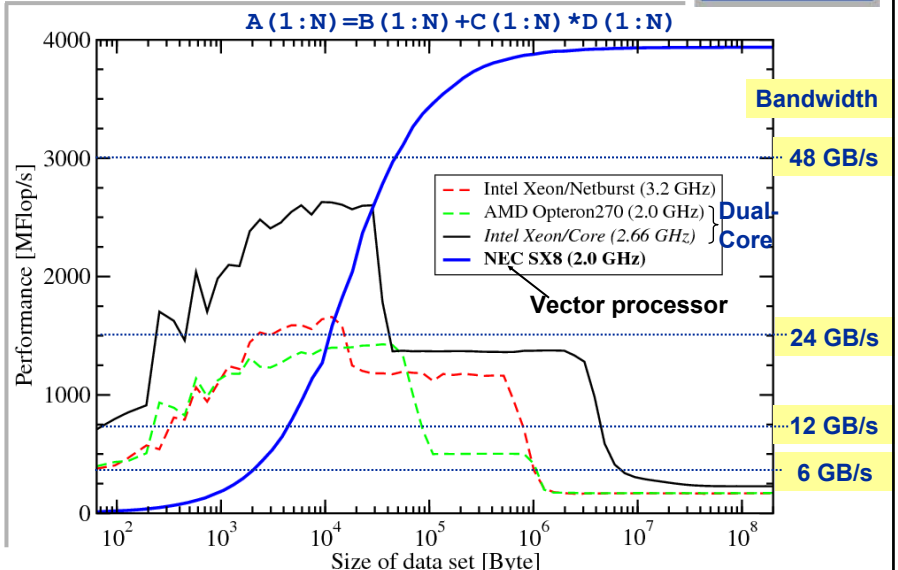
## Memory hierarchies of modern processors

## Memory hierarchies
*Schematic View*






## Memory hierarchies
*Performance Characteristics*



## Memory hierarchies

*Cache-based architectures – "private cluster"*

| *SSE2 | | Intel Xeon/ Netburst | Intel Xeon / Core | AMD Opteron |
|---|---|---|---|---|
| **Peak Performance** Core frequency | | 7.2 GFlop/s 3.6 GHz | **12.0 GFlop/s** 3.0 GHz | **5.6 GFlop/s** 2.8 GHz |
| **#Registers** | | 16 / 32* | **16 / 32*** | **16 / 32*** |
| **L1** | **Size** | 16 kB | 32 kB | 64 kB |
| | **BW** | 115 GB/s | 96 GB/s | 45 GB/s |
| | **Latency** | 12 cycles | 3 cycles | 3 cycles |
| **L2** | **Size** | 2 MB | **4 MB (2cores)** | **1 MB** |
| | **BW** | 115 GB/s | **96 GB/s** | **45 GB/s** |
| | **Latency** | 20 cycles | **13 cycles** | **12 cycles** |
| **Memory** | **BW** | 6.4 GB/s | **10.6 GB/s** | **10.6 GB/s** |
| | **Latency** | ~200 ns | **~200 ns** | **< 100 ns** |

***dual–core CPUs***



## Memory hierarchies

*Processor architectures – "supercomputing centers"*

| *dual - core | | NEC SX8 | IBM Power5* | Intel Itanium2* |
|---|---|---|---|---|
| **Peak Performance** Core frequency | | **16 GFlop/s** 2.0 GHz | **7.2 GFlop/s** 1.9 GHz | **6.4 GFlop/s** 1.6 GHz |
| **#Registers** | | **8*256** | **32** | **128** |
| **L1** | **Size** | | 32 kB | *16 KB* |
| | **BW** | | 72 GB/s | *51.2 GB/s* |
| | **Latency** | | 3 cycles | *1 cycle* |
| **L2** | **Size** | | **1.9 MB (2 cores)** | 256 KB |
| | **BW** | | **72 GB/s** | 51.2 GB/s |
| | **Latency** | | **~13 cycles** | 5-6 cycles |
| **L3** | **Size** | | 36 MB | **6 / 12 MB** |
| | **BW** | | ~10 GB/s | **51.2 GB/s** |
| | **Latency** | | ~80 cycles | **12-13 cycles** |
| **Memory** | **BW** | **64 GB/s** | **~10 GB/s** | **8.5 GB/s** |
| | **Latency** | **vectorization** | **100 ns** | **~200 ns** |

## Memory hierarchies
*Characterization*

Two quantities characterize the quality of each memory hierarchy:

- **Latency ($T_{lat}$):** *Time to set up the memory transfer from source (main memory or caches) to destination (registers).*
- **Bandwidth (BW):** *Maximum amount of data which can be transferred per second between source (main memory or caches) and destination (registers).*

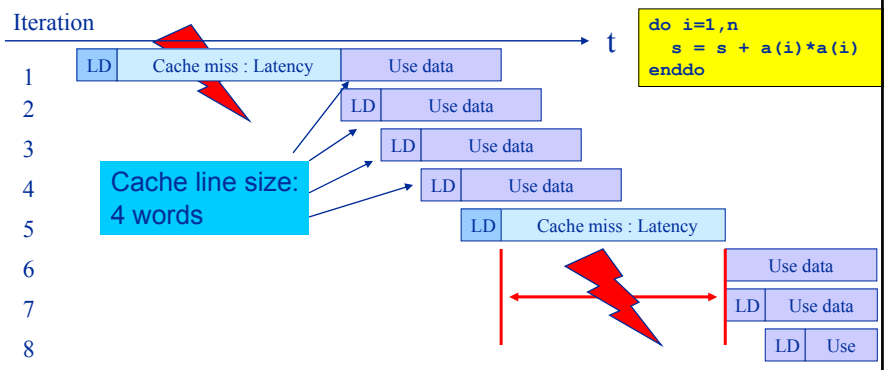
Transfer time: $T = T_{lat} + (\text{amount of data}) / BW$

- For microprocessor holds $T \tilde{=} T_{lat}$
  (e.g.: $T_{lat}$=100 ns; BW=4 GByte/s; amount of data=8 byte-> T=102 ns)
- Caches are organized in cache lines that are fetched/stored as a whole (e.g. 128 byte = 16 double words)



## Memory hierarchies
*Cache structure*

- If one item is loaded from main memory (cache miss), the whole cache line it belongs is loaded to the caches
- Cache lines are contiguous in main memory, i.e. “neighboring“ items can then be used from cache




$T_{lat}$=100 ns; BW=4 GByte/s; amount of data=128 byte-> T=132 ns

## Memory Hierarchies
*Cache Structure*

- Cache line data is always consecutive
  - Cache use is optimal for contiguous access (stride 1)
  - Non-consecutive reduces performance
  - Access with wrong stride (e.g. with cache line size) can lead to disastrous performance breakdown
- Long cache lines reduces the latency problem for contiguous memory access. Otherwise: latency problem becomes worse.
- Calculations get cache bandwidth inside the cache line, but main memory latency still limits performance

- Cache lines must somehow be mapped to memory locations
  - **Cache multi-associativity enhances utilization**
  - **Try to avoid cache thrashing**



## Memory Hierarchies
*Cache Line Prefetch to hide latencies*

- *Prefetch (PFT) instructions:*
  - Transfer of consecutive data (one cache line) from memory to cache
  - Followed by LD to registers
  - Useful for executing loops with consecutive memory access
  - Compiler has to ensure correct placement of PFT instructions
    - Knowledge about memory latencies required
    - Loop timing must be known to compiler
  - Due to large latencies, outstanding pre-fetches must be sustained

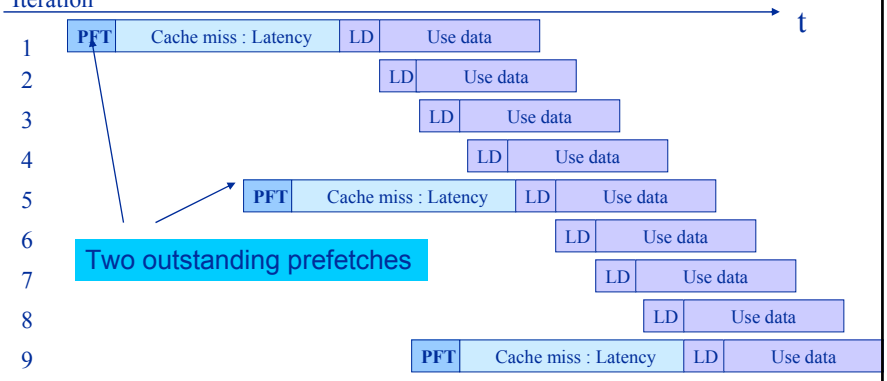
## Memory Hierarchies

*Cache Line Prefetch to hide latencies*

Prefetching allows to overlap of data transfer and calculation!

Hardware assisted prefetching for long contiguous data accesses

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




Intel Itanium2/EPIC: Software pipelining using PFT operations

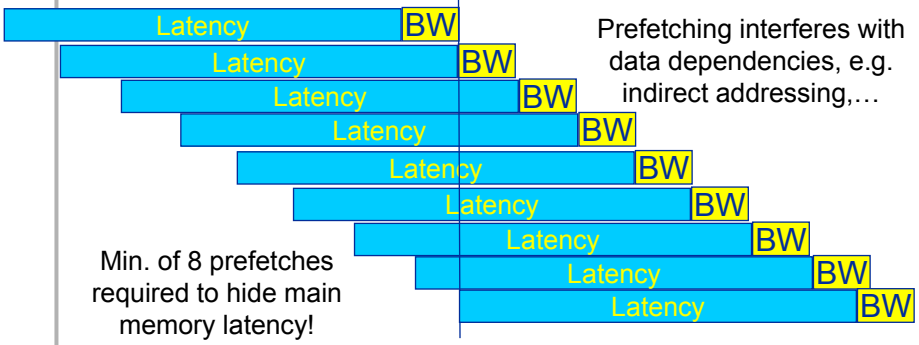


## Memory Hierarchies

*Cache Line Prefetch to hide latencies*

Hide memory latency on Itanium2 systems:

1. Latency approx.140 ns
2. Time to transfer one cache-line: 128 Byte/6.4 GByte/s = 20 ns
3. Total time to transfer one cache line: 160 ns



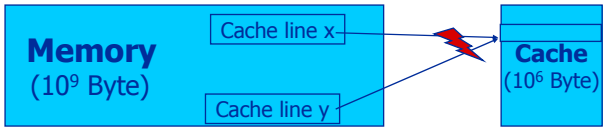

Prefetching interferes with data dependencies, e.g. indirect addressing,…

Min. of 8 prefetches required to hide main memory latency!

Long loops: min. 8*16

## Memory Hierarchies
*Cache Mapping*

- **Cache Mapping**
  - Pairing of memory locations with cache line
  - e.g. mapping 1 GB of main memory to 1 MB of cache




- **Static Mapping**
  - Directly Mapped caches vs. m-way set associative caches

- **Replacement strategies**
  If all potential cache locations are full, one line has to be overwritten ("invalidated") on next cache load using different strategies:

  Least Recently Used (LRU) random vs. Not Recently Used (NRU)

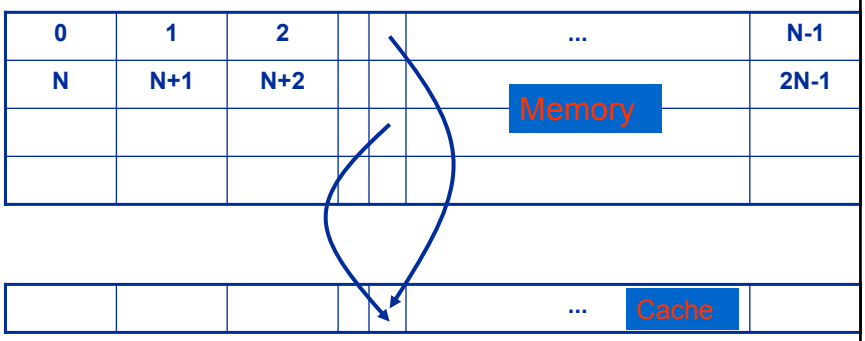
  May incur additional data transfer (→ cache thrashing) !



## Memory Hierarchies
*Cache Mapping – Directly mapped*

- **Directly mapped cache:**
  - Every memory location can only be mapped to exactly one cache location
  - If cache size=n, i-th memory location can be stored at cache location mod(i,n)
  - Easy to implementation & fast lookup
  - No penalty for stride-one access
  - 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!

## Memory Hierarchies
*Cache Mapping – Directly Mapped*




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*



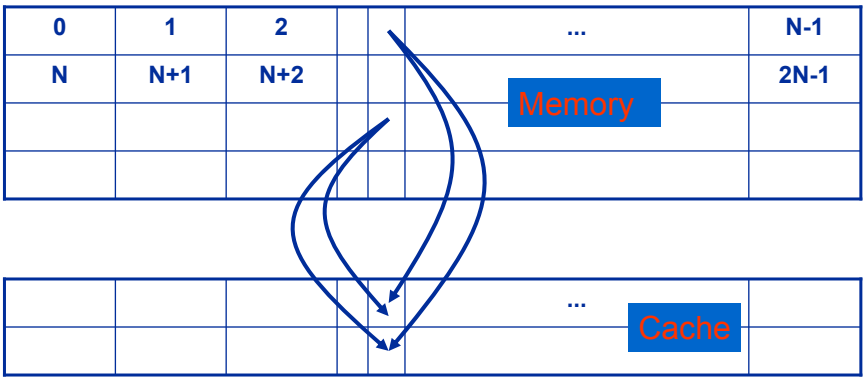
## Memory Hierarchies
*Cache Mapping – Associative Caches*

- **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 |

- **Ideal world**: Fully associative cache where every memory location is mapped to any cache line
  - Thrashing nearly impossible
  - The higher the associativity, the larger the overhead, e.g. latencies increase; cache complexity limits clock speed!

## 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
*Pitfalls & Problems*

- 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

  ↓

  Warning: Using powers of 2 in the leading array dimensions of multi-dimensional arrays should be avoided! (Cache Thrashing)

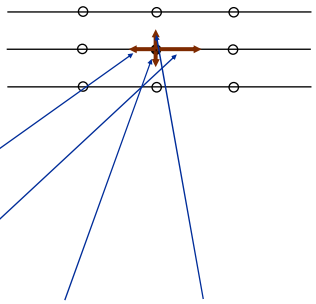
- If the cache / m associativity slots are full and new data comes in from main memory, data in cache (cache line) must be invalidated or written back to main memory

  ↓

  Ensure spatial and temporal data locality for data access! (Blocking)

## Memory hierarchies
*Cache thrashing - Example*

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

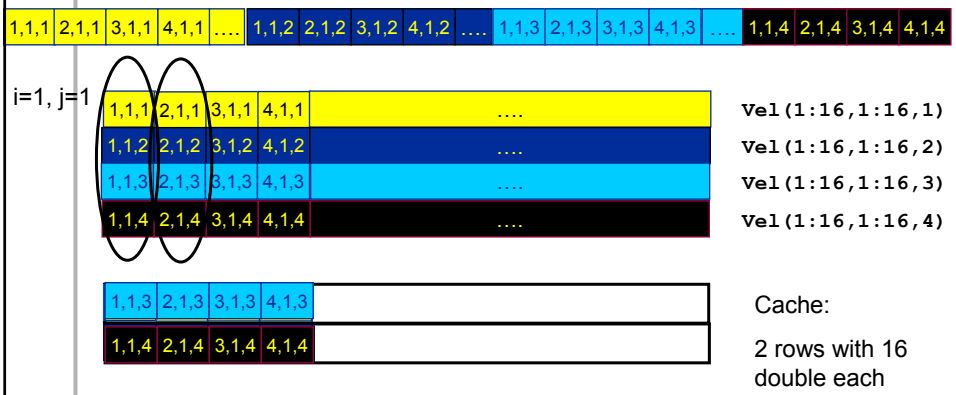


```
N=16
real*8 vel(1:N , 1:N, 4)
……
s=0.d0
do j=1,N
   do i=1,N
      s=s+vel(i,j,1)-vel(i,j,2)+vel(i,j,3)-vel(i,j,4)
   enddo
enddo
```



## 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




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

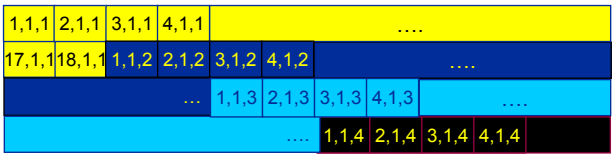
## Memory hierarchies

*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!



**Characterization of Memory Hierarchies:**

**Modeling,**

**measuring and**

**understanding**

**the performance limitations**

## Characterization of Memory Hierarchies:
*Kernel benchmark – Vector-Triad*

- Kernel benchmarks:
  - Characterize computer architecture (effective performance of processor & memory hierarchies)
  - Results are easy to be interpreted and to be compared
  - Provide upper performance bounds for specific applications
  - E.g.: *stream, cachebench*

Vector-Triad

```
REAL*8 (SIZE): A,B,C,D
DO ITER=1,NITER
  DO i=1,N
    A(i) = B(i) + C(i) * D(i)
  ENDDO
  <OBSCURE>
ENDDO
```

**Balance**

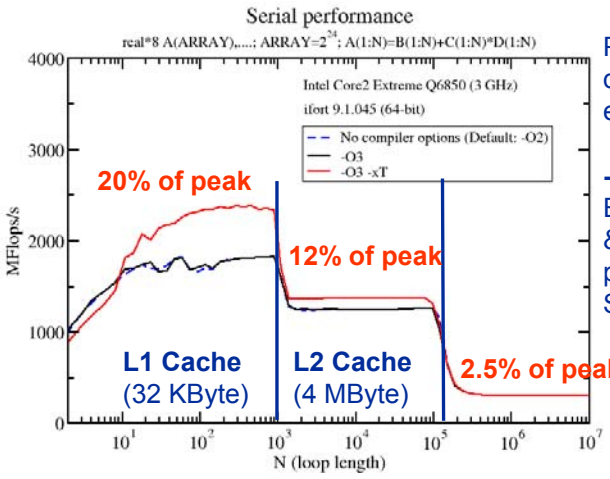
•Computation:
**2 Flops / i-Iteration**
• Bandwidth:
**(3 LD & 1 ST) / i-Iteration**

• *Balance = 2 Word / Flop*



## Characterization of Memory Hierarchies:
*Vector-Triad on Intel Core2 Q6850: 1 core results*

Peak Performance for 1 core of Intel Core2 Q6850 (double arguments):
3 GHz * (2 Flops (DP-Add) + 2 Flops (DP-Mult)) = **12 GFlops/s**




Performance decreases if data set exceeds cache size

**-xT** :
Enables vectorization & improves in-cache performance: Packed SSE instructions

**WHY?!**

## Characterization of Memory Hierarchies:
*Vector-Triad on Intel Core2 Q6850: 1 core results*

```
A(1:N) = B(1:N) + C(1:N) * D(1:N)
```

- **Potential limiting factors:**
  - **FP units: 1 ADD + 1 MULTIPLY**
  - **Data transfer: L1 cache – Registers**
    - Available per cycle (for consecutive data):
      128 Bit Load from L1 + 128 Bit Store to L1
    - Code requires per iteration (2 Flops):
      3 Loads (64 Bit) from L1 and 1 Store (64 Bit) to L1
    - Consider data transfer for four successive iterations:
      (assume no cost for FP computation)

LD C(i-2:i-1)
LD D(i-2:i-1)
LD B(i-2:i-1) ST A(i-4:i-3)

LD C(i :i+1)
LD D(i :i+1)
LD B(i :i+1) ST A(i-2:i-1)

*3 cycles to transfer data for 2 iterations*

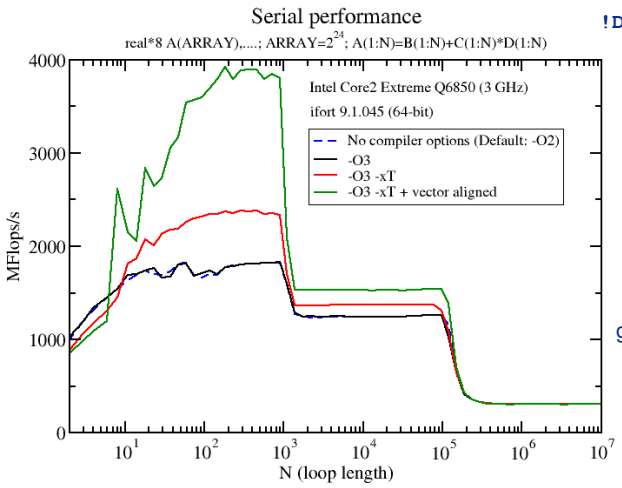
*Maximum performance for Vector-Triad:*
*4 Flop/3 cycles*

***4 GFlops/s at 3 GHz***



## Characterization of Memory Hierarchies:
*Vector-Triad on Intel Core2 Q6850: 1 core results*

Maximum performance of Vector-Triad in L1 cache: **4 GFlops/s**




Put directive before the loop:

```
!DEC$ VECTOR ALIGNED
```

Compiler generates „perfect“ assembly code:

Aligned SSE LD/ST instructions

However:

Programmer has to guarantee that arrays are aligned to 16-Byte boundaries!!!

## Characterization of Memory Hierarchies:
*Vector-Triad on Intel Core2 Q6850: 1 core results*

```
A(1:N) = B(1:N) + C(1:N) * D(1:N)
```

- **Potential limiting factors:**
  - **Data transfer: L2 cache – L1 cache - Registers**
    - L2 – L1 bandwidth: 4 double words (256 Bits) per cycle
    - Data transfer between L1 and L2 is on cache line basis (8 double words), i.e. 2 cycles arer required to transfer 1 cache line
    - L1 cache: write back cache (Core2 architectural design feature) If A(i) is not resident in L1 („L1 store miss“) it must first be loaded from L2 to L1 (complete cache line of A(i)): „Read for Ownership“ (RFO)
    - Data transfers for a single iteration: L1 / L2
      - 1 L1→L2 transfer: A(i)
      - 4 L2→L1 transfers: A(i), B(i), C(i), D(i)



## Characterization of Memory Hierarchies:
*Vector-Triad on Intel Core2 Q6850: 1 core results*

```
A(1:N) = B(1:N) + C(1:N) * D(1:N)
```

- **Potential limiting factors:**
  - **Data transfer: L2 cache – L1 cache – Registers (continued)**
    - Assuming that L1 – Register and L2 – L1 transfer can not occur concurrently 10 more cycles are required for 8 iterations (1.25 cycles/iteration):
      L2→L1: C(i:i+7) [2 cycles]
      L2→L1: D(i:i+7) [2 cycles]
      L2→L1: B(i:i+7) [2 cycles]
      L2→L1: A(i:i+7) [2 cycles]
      L1→L2: A(i:i+7) [2 cycles]
    - Total transfer time for 2 iterations:
      L2→L1: 2.5 cycles
      L1→Registers: 3.0 cycles (cf. L1-Register slide)

      2 iterations/ 5.5 cycles → 4 Flops/5.5 cycles

      → **Max. performance for data in L2 cache: 2.18 GFlops/s**

## Characterization of Memory Hierarchies:
*Vector-Triad on Intel Core2 Q6850: 1 core results*

```
A(1:N) = B(1:N) + C(1:N) * D(1:N)
```

- **Potential limiting factors:**
  - **Data transfer: Memory – L2 cache:**
  - **Memory access through FSB1333@64Bit, i.e. Memory-L2 bandwidth: 1.33 GHz * 8 Byte = 10.67 GByte/s**
  - **L2 cache is write back cache -> RFO for A(i) from memory required**
  - **For a single iteration (4+1) double words = 40 Bytes have to be transferred → 20 Byte/Flop**

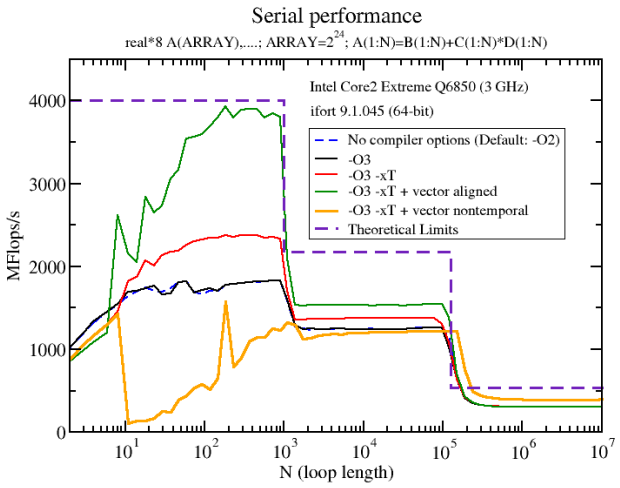
    **→ Max. performance for data in main memory:**

    **(10.67 GByte/s) / (20 Byte/Flop) = 0.534 GFlops/s**



## Characterization of Memory Hierarchies:
*Vector-Triad on Intel Core2 Q6850: 1 core results*

- Insert `!DEC$ VECTOR NONTEMPORAL` → A is directly written to main memory, BYPASSING the caches – No RFO load operation



# Optimization of data access

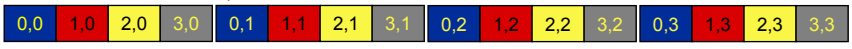


## Data layout optimizations
*Basics*

- 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!

- Stride one access in inner loops to exploit cache lines!

```
real*8 A(SIZE,SIZE)
real*8 B(SIZE,SIZE)
N=SIZE
do i=1,N
   do j=1,N
      A(j,i)=A(j,i)*B(j,i)
   enddo
enddo
```

```
real*8 A(SIZE,SIZE)
real*8 B(SIZE,SIZE)
N=SIZE
do j=1,N
   do i=1,N
      A(j,i)=A(j,i)*B(j,i)
   enddo
enddo
```

## Data layout optimizations
*Example: Dense MVM*

- **Dense matrix vector multiplication (MVM) is a frequently used kernel operation**

  **y = y + A * x**

- **DMVM involves data access (assuming square matrix):**
  - Matrix: real*8 A(N,N) → N*N double words (8 Bytes)
  - Vector1: real*8 x(SIZE) → N double words (8 Bytes)
  - Vector2: real*8 y(SIZE) → N double words (8 Bytes)
- **Amount of data involved: ($N^2$ + 2 * N) Words (=double words)**
- **#Floating point ops: 2 * N * N Flop**
  (1 ADD & 1 MULT for each matrix entry)
- **Balance between data transfer and Flop=** (($N^2$+2N) W) / (2 $N^2$ Flop)
  = (0,5+ 1/N) W/Flop **~ 0.5 W/Flop (for large N)**



## Data layout optimizations
*Dense MVM*

- If the matrices are big, data transfer should be minimized!
- Calculate number of memory references

```
do i=1,N
 do j=1,N
  y(i)=y(i)+A(j,i)*x(j)
 enddo
enddo
```

N + N + N*N + N*N

= 2* N + 2 * $N^2$

```
do j=1,N
 do i=1,N
  y(i)=y(i)+A(j,i)*x(j)
 enddo
enddo
```

N*N + N*N + N*N + N

= N + 3 * $N^2$

Benefit:

- Lower data transfer for large N
- Contiguous access to A

Implementation still away from minimal data amount: **2*N + $N^2$** !

## Minimize Memory References
*Dense MVM*

- **Start with stride 1 access**

```
do i=1,N
  tmp=0.d0
  do j=1,N
     tmp = tmp + a(j,i) * x(j)
  end do
  y(i) = y(i) + tmp
end do
```

Innermost loop: two loads and two flops performed:

**Balance=1 Word / Flop**

Vector x is still loaded N times!

**Use outer loop unrolling or blocking to reduce the number of references to vector x!**



## Minimize Memory References
*Dense MVM: Outer loop unrolling*

**Outer loop unrolling**

```
do i=1,N,2
  t1=0
  t2=0
  do j=1,N
    t1=t1+a(j,i)   *x(j)
    t2=t2+a(j,i+1) *x(j)
  end do
  y(i)  =t1
  y(i+1)=t2
end do
```

Outer loop unrolled twice

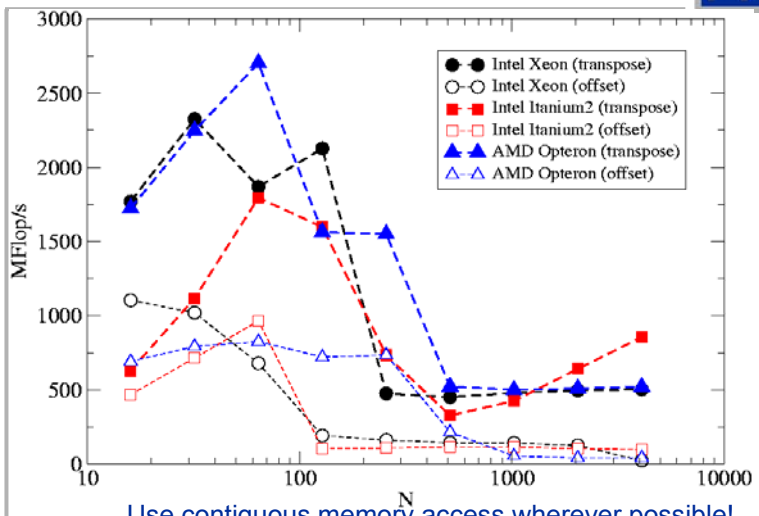
Innermost loop:
three loads and four flops: **0.75 W/Flop**
**(1.5 $N^2$ data transfers)**

How about unrolling by 4? **0.625 W/Flop**
**(1.25 $N^2$ data transfers)**

**Watch register spill!**

## Minimize Memory References

*Dense MVM: Plain programming*

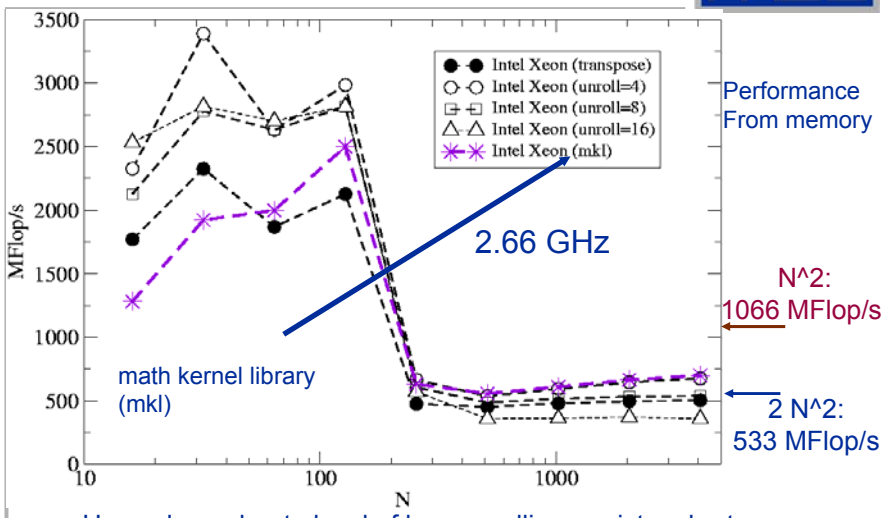



Use contiguous memory access wherever possible!
(Compilers did not interchange loops!)



## Minimize Memory References

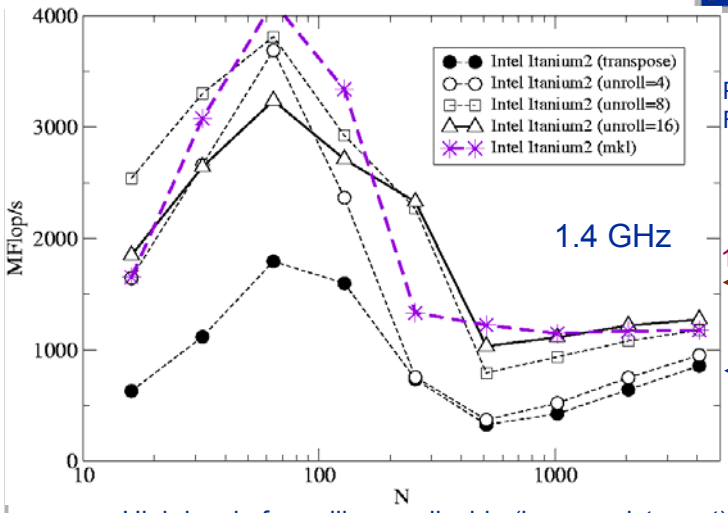
*Dense MVM: Outer loop unrolling – Intel Xeon (Netburst)*




Use only moderate level of loop unrolling: register shortage

## Minimize Memory References

*Dense MVM: Outer loop unrolling – Intel Itanium2*




Performance From memory

N^2: 1600 MFlop/s

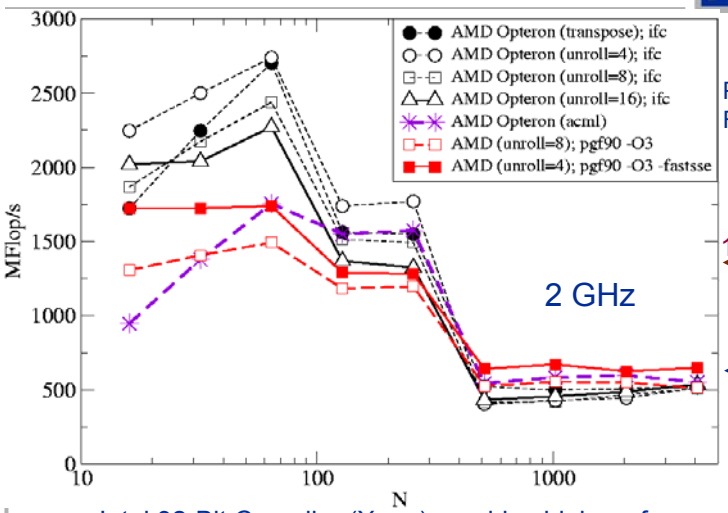
2 N^2: 800 MFlop/s

High level of unrolling applicable (large register set)!
Intel Library (mkl) does very well! Use Intel compilers!



## Minimize Memory References

*Dense MVM: Outer loop unrolling – AMD Opteron*




Performance From memory

N^2: 1320 MFlop/s

2 N^2: 660 MFlop/s

Intel 32-Bit Compiler (Xeon) provides high performance!
Only moderate unrolling (pgf90 (64-Bit) - room for improvement)

## Minimize Memory References
*Dense MVM: Blocking*

Blocking: Split up inner loop in small chunks (pref. Cache Line Size) and perform all computations.

```
do i=1,N
  do j=1,N
    y(i)=y(i)+a(j,i)*x(j)
  end do
end do
```

Whole vector x is loaded from memory or cache to register_ N times!

```
bs=CLS, nb=N/bs
do k=1,nb
  do i=1,N
    do j=(k-1)*bs+1,k*bs
      y(i)=y(i)+a(j,i)*x(j)
    end do
  end do
end do
```

What is the problem here?

Vector x is loaded only once and (re-)used cacheline by cacheline.

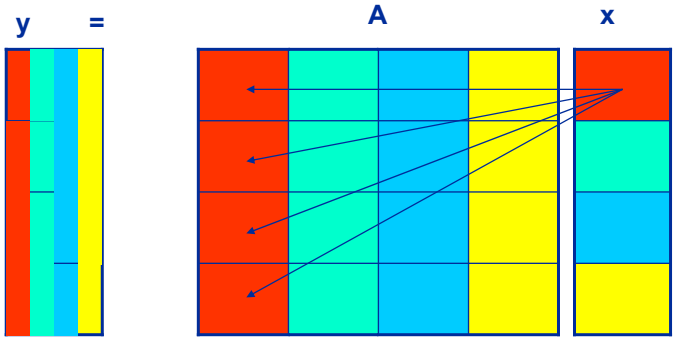
**CLS**: Cache Line Size

Blocking size (**bs**) should be **CLS** or multiple of it. Upper limit is imposed by cache size



## Minimize Memory References
*Dense MVM: Blocking*

### Blocking Matrix-Vector Multiply




Cache line size

Save loads (x) from memory at the cost of additional store operations (y):

- Vector x is loaded only once instead of **N** times
- Vector y is loaded **nb** times instead of only once

## Minimize Memory References
*Blocking – Exercise*

Optimize Matrix transpose !

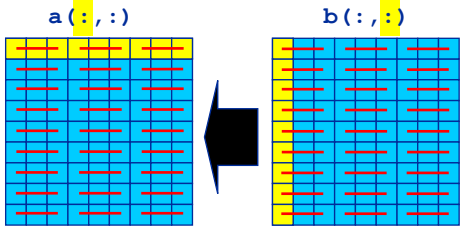
A(i,j) = B(j,i)



## Minimize Memory References
*Exercise: Dense matrix transpose*

- **Simple example for data access problems in cache-based systems**
- **Naïve code:**

```
do i=1,N
 do j=1,N
  a(j,i) = b(i,j)
 enddo
enddo
```



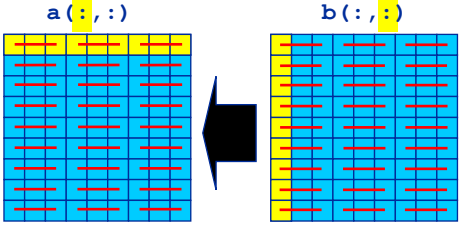

- **Problem: Stride-1 access for `a` implies stride-N access for `b`**
  - Access to `a` is perpendicular to cache lines ( —)
  - Possibly bad cache efficiency (spatial locality)
- **Remedy: Outer loop unrolling and blocking**

## Minimize Memory References

*Exercise: Dense matrix transpose*

- **Data transfer analysis**

```
do i=1,N
 do j=1,N
  a(j,i) = b(i,j)
 enddo
enddo
```

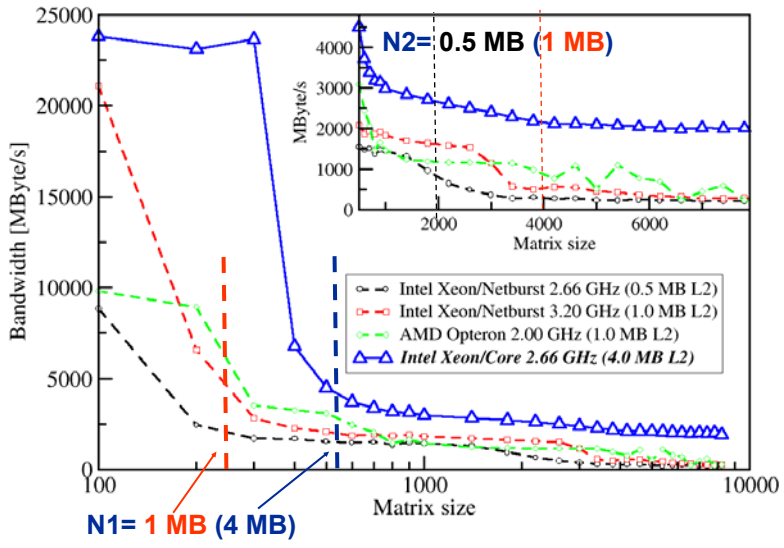



- **Assume a L2 cache of size L2SIZE and a CLS of 16 double (128 Byte)**
- **All data fits in L2 cache:**
  `2 * N1² * 8 Byte < L2SIZE`
- **All cache lines of B stay in L2 cache until they are fully used:**
  `2 * N2 * 16 * 8 Byte = 256 *N2 Byte < L2SIZE`



## Minimizing Memory References

*Dense matrix transpose: Vanilla version*



## Minimizing Memory References

*Dense matrix transpose: Unrolling and blocking*

```
do i=1,N
 do j=1,N
  a(j,i) = b(i,j)
 enddo
enddo
```

unroll/jam

```
do i=1,N,U
 do j=1,N
  a(j,i)     = b(i,j)
  a(j,i+1)   = b(i+1,j)
  ...
  a(j,i+U-1) = b(i+U-1,j)
 enddo
enddo
```

block

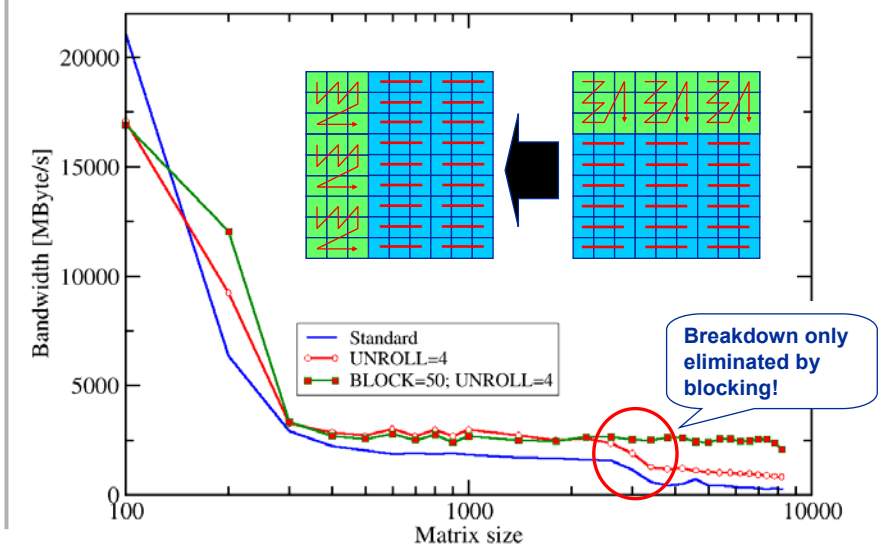
```
do ii=1,N,B
 istart=ii; iend=ii+B-1
 do jj=1,N,B
  jstart=jj; jend=jj+B-1
  do i=istart,iend,U
   do j=jstart,jend
    a(j,i)     = b(i,j)
    a(j,i+1)   = b(i+1,j)
    ...
    a(j,i+U-1) = b(i+U-1,j)
enddo;enddo;enddo;enddo
```

Blocking and unrolling factors (B,U) can be determined experimentally; be guided by cache sizes and line lengths



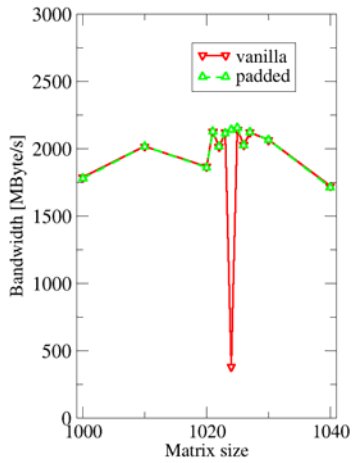
## Minimizing Memory References

*Dense matrix transpose: Block/unroll - Intel Xeon 3.2 GHz*



## Minimizing Memory References
*Dense matrix transpose: Cache thrashing*

- **A closer look (e.g. on Xeon/Netburst) reveals interesting performance characteristics:**




- **Matrix sizes of powers of 2 seem to be extremely unfortunate**
  - Reason: Cache thrashing!
- **Remedy: Improve effective cache size by padding the array dimensions!**
  - `a(1024,1024)` → `a(1025,1025)`
    `b(1024,1024)` → `b(1025,1025)`
  - Eliminates the thrashing completely
- **Rule of thumb: If there is a choice, use dimensions of the form 16•(2k+1)**



## Literature


- G. Hager and G. Wellein
  *Concepts of High Performance Computing*
  `http://www.blogs.uni-erlangen.de/hager/topics/OHN/`
- K. Dowd, C. Severance
  *High Performance Computing*
  O' Reilly, 2nd Edition (ISBN 156592312X)
- S. Goedecker, A. Hoisie
  *Performance Optimization of Numerically Intensive Codes*
  Society for Industrial & Applied Mathematics,U.S. (ISBN 0898714842)
- J.L. Hennessy, D.A. Patterson
  *Computer Architecture – A Quantitative Approach*
  Morgan Kaufmann Publishers , Elsevier 4th Edition (ISBN 0123704901 )