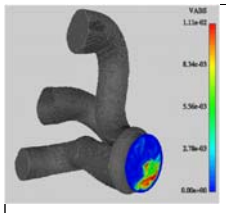
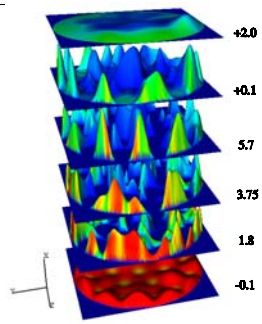
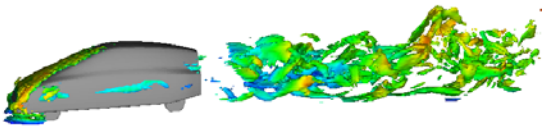
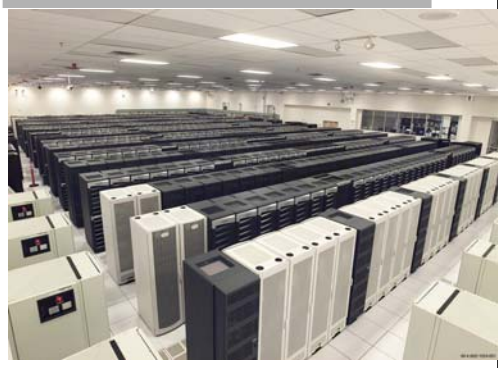



Dr. Gerhard Wellein, Dr. Georg Hager

HPC Services –

Regionales Rechenzentrum Erlangen

Universität Erlangen-Nürnberg

Vorlesung an der FHN im SS 2007




Format of lecture

- **3 days course: 28.2./1.3./2.3.**
- **12 units (90 minutes each) in total**
- **2 lectures in the morning**
  - **8:30-10:00**
  - **10:30-12:00**
- **2 exercises in the afternoon (180 minutes)**
  - **13:30-16:30**
  - **Exercises will be performed at RRZE cluster**
- **5.3.: Exam (8:30-10:00)**
- **5.3.: Visit to RRZE (13:00-14:30)**

Topics of lecture

- 28.2.: Introduction & Single processor
- 1.3.: (Shared memory) Parallelism, OpenMP & Multi-Core
- 2.3.: Distributed memory parallelism, MPI & Clusters
- Presentations are available on the web:
  http://www.home.uni-erlangen.de/hager/topics/FHN/
- Exam: 60 minutes, no supporting material allowed

Survey (28.2.)

Introduction

Single Processor: Architecture & Programming

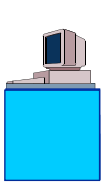
- Microprocessors & Pipelining
- Memory hierarchies of modern processors
- Literature

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**
- **Distributed resources connected via GRID 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..



Available computing time

Available main memory

BOTH

1 GByte
6 GFlop/s
PC / Desktop

Scale: 50000

50.000 GByte
300.000 GFlop/s
Large scale supercomputers

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

Introduction

- **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: 60 GFlop/s (Top1)
  - Latest issue (Nov. 2006): 280.600 GFlop/s (Top1)
  - Performance increase: 92 % 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

Introduction

Top500 list as of November 2006

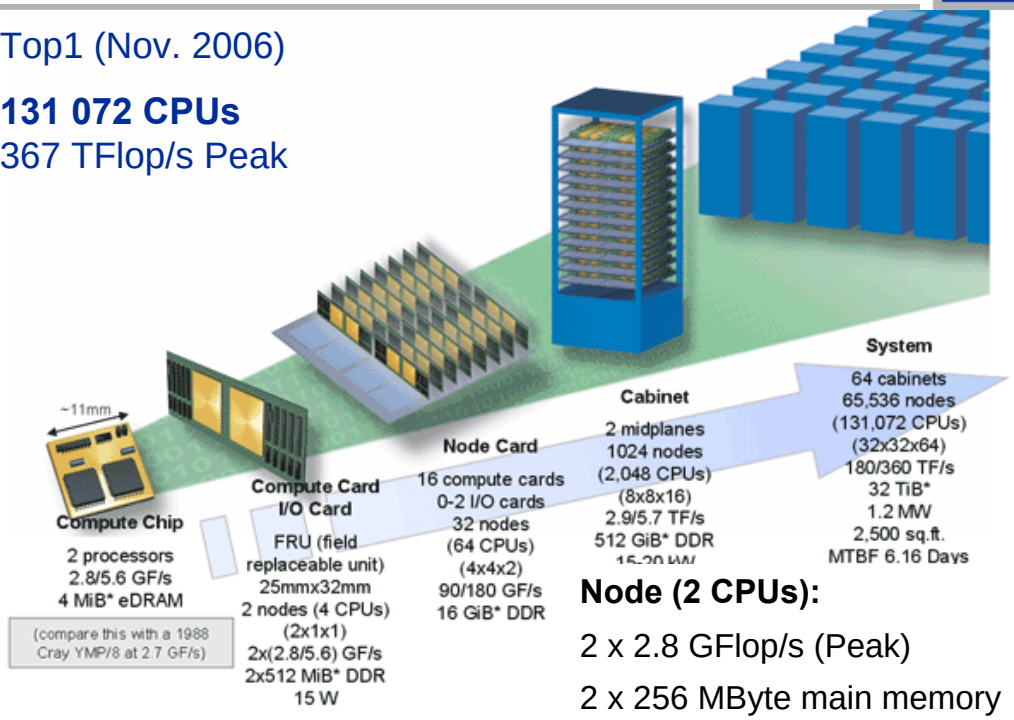
Rank	Site		Computer	Country		Procs.	RMax	RPeak
1	DOE/NNSA/ LLNL	IBM	eServer Blue Gene Solution	U.S.	2005	131072	280600	367000
2	Sandia National Laboratories	Cray Inc.	Red Storm Cray XT3, Opteron 2.4 GHz	U.S.	2006	53088	101400	127411
3	IBM Thomas J. Watson Research Center	IBM	eServer Blue Gene Solution	U.S.	2005	40960	91290	114688
4	DOE/NNSA/ LLNL	IBM	eServer pSeries p5 575 1.9 GHz	U.S.	2006	12208	75760	92781
5	Barcelona Supercomputing Center	IBM	PowerPC 2.3 GHz	Spain	2006	10240	62630	94208
13	Forschungszentrum Juelich (FZJ)	IBM	eServer Blue Gene Solution	Germany	2006	16384	37330	45875
18	LRZ Munich	SGI	Altix4700 Itanium2-1.6 GHz	Germany	2006	4096	24360	26214
124	RRZE Erlangen	HP	Intel Xeon5160 3 Ghz Infiniband	Germany	2006	728	5416	8736

LINPACK [GFlop/s]

Introduction
IBM BlueGene/L

Top1 (Nov. 2006)

131 072 CPUs
367 TFlop/s Peak



Node (2 CPUs):

2 x 2.8 GFlop/s (Peak)

2 x 256 MByte main memory

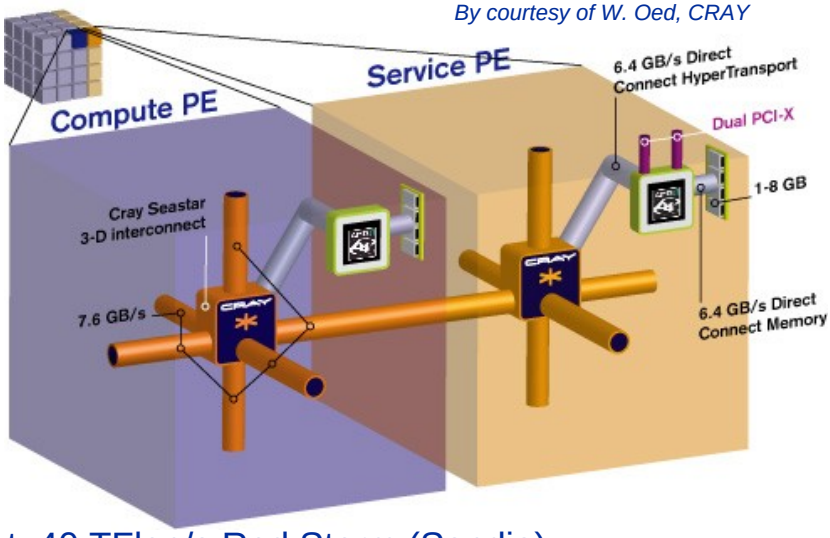
180 MByte/s in each dir. (Interconnect)

Introduction
CRAY XT3

- 3rd generation of CRAY MPP systems (1 node = 1 CPU or DC chip)
- Successor of CRAY T3E
- System is designed to scale to 10.000s CPUs
- Large configs.:
  ORNL: 20 TFlop/s
  PSC: 10 TFlop/s
- NERSC:
  100 TFlop/s in '07

Cray XT3 Scalable Architecture

By courtesy of W. Oed, CRAY



- Original development: 40 TFlop/s Red Storm (Sandia)
- OS: Linux micro kernel

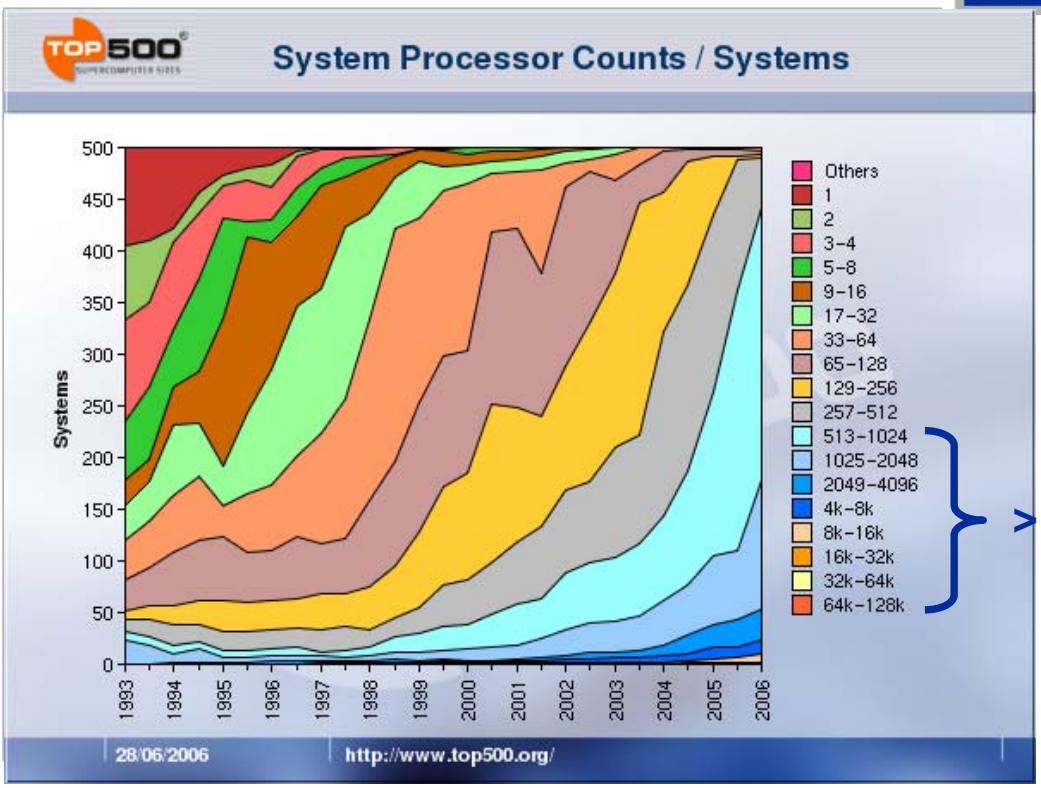
Introduction
RRZE Cluster

- 760 Intel Xeon5160 processor cores
  - Core2Duo architecture
  - 3.0 GHz -> 12 GFlop/s per core
  - 4 cores per compute node
  - Installation: November 2006
- Peak performance: 9120 GFlop/s
- Main memory:
  - 2 GByte per core
  - 1520 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

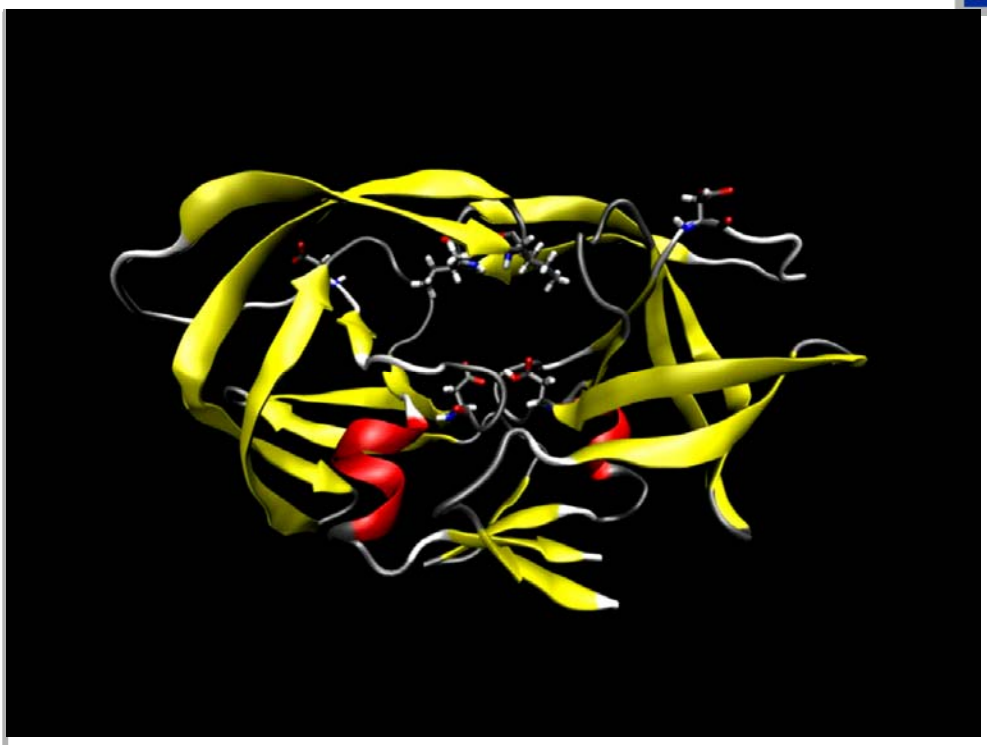
Introduction
Trends in parallel computing



> 80 %

Introduction

MD Simulation of HIV protease dynamics

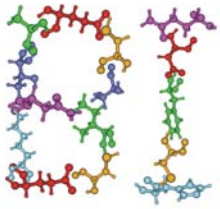


Real time:
10 ns

Compute time:

18.000 CPU-hrs

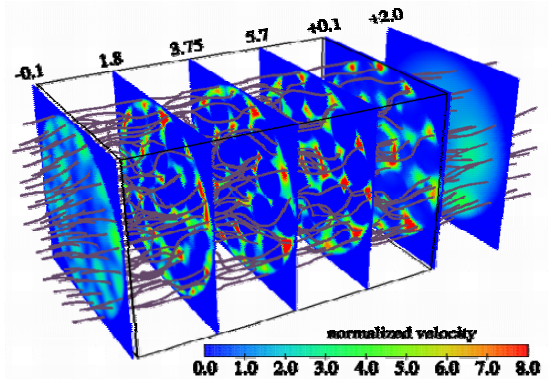
8 CPUs – 90 days

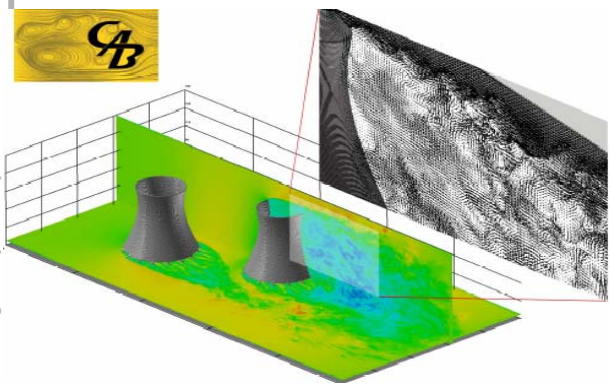


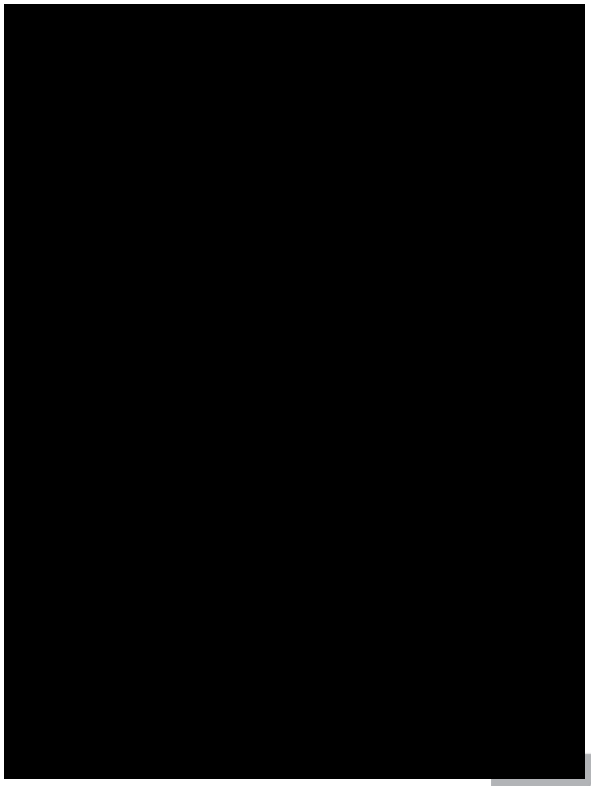
Courtesy: Prof. Sticht, Bio-Informatics, Emil-Fischer Center, FAU

Introduction

Lattice Boltzmann flow solvers

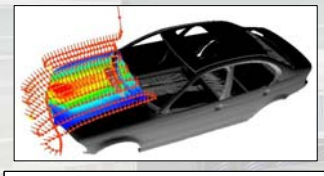


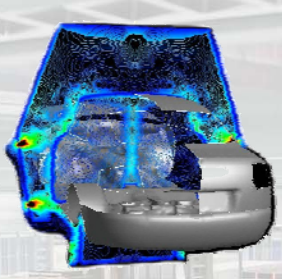


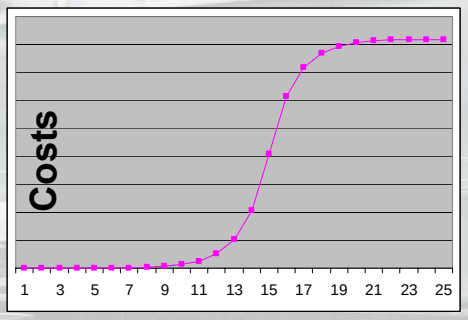


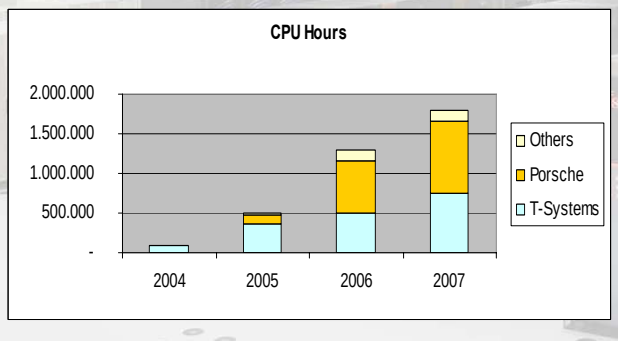
Figures by courtesy of LS CAB-Braunschweig, Thomas Zeiser, N. Thuerey

Introduction
Industrial Usage 2004-2007 (HLRS)





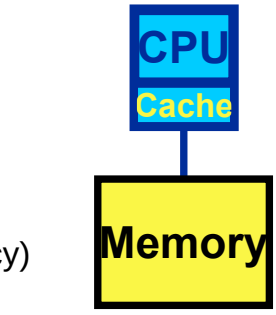


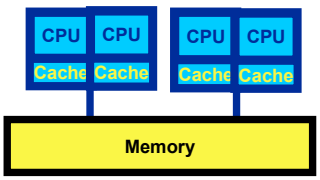


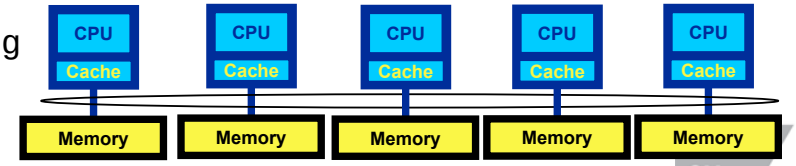
Product cycle

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

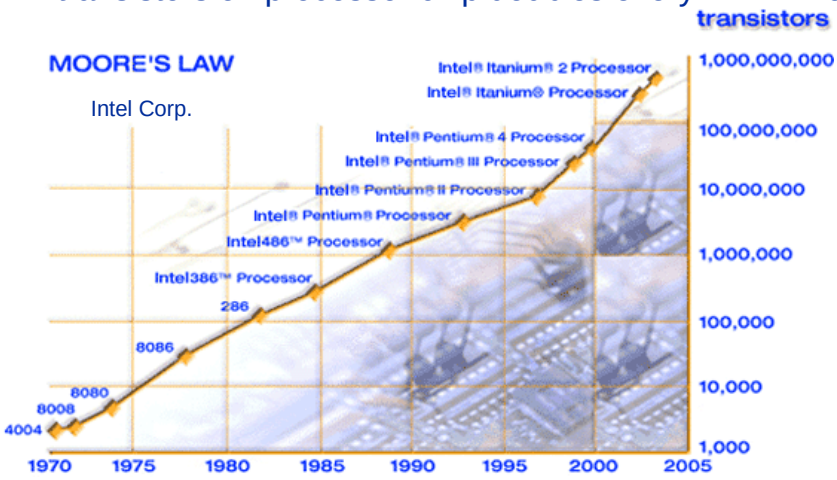


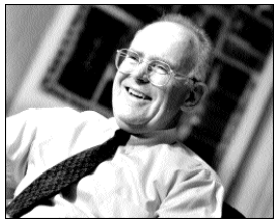




Introduction
Faster computers: Clock speed

- **1965 G. Moore claimed**
  #transistors on processor chip doubles every 12-24 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

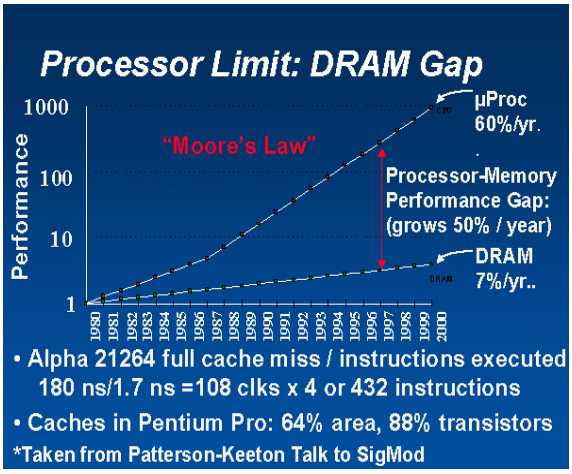
- High complexity of computer architectures
  - CISC -> RISC
  - Out-of Order Execution
  - Introduction of new architectures: EPIC/Itanium with fully in-order instruction issue

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

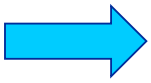
Introduction

Faster computers: Clock speed vs. DRAM gap

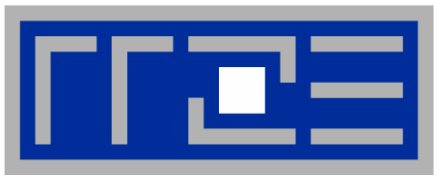
- **Memory (DRAM) Gap**
  - **Memory bandwidth grows only at a speed of 7% a year**
  - Memory latency remains constant / increases in terms of processor speed
  - Loading a single data item from main memory can cost 100s of cycles on a 3 GHz CPU



- Introducing memory hierarchies (caches) – Complex optimization of code



Optimization of main memory access is mandatory for most applications



Microprocessors & Pipelining

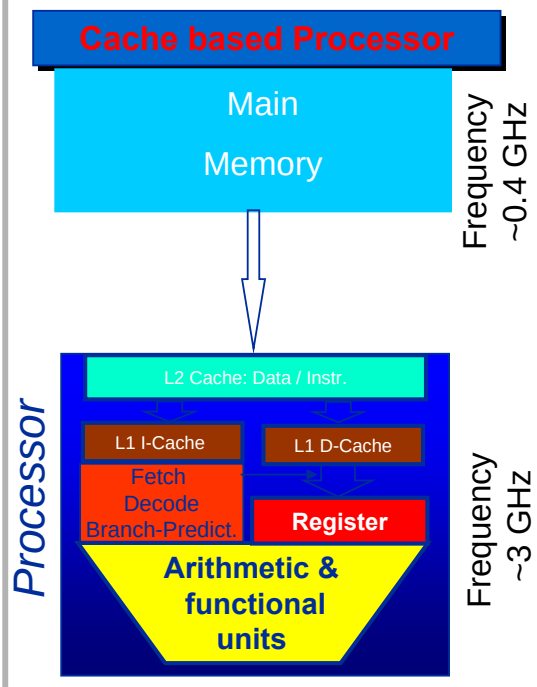
Architecture of modern microprocessors
History

- 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 (P4, Athlon, Opteron): Compiler still generates CISC instructions; but processor core: RISC like
  - RISC is still implemented in most dual- / quad-core CPUs

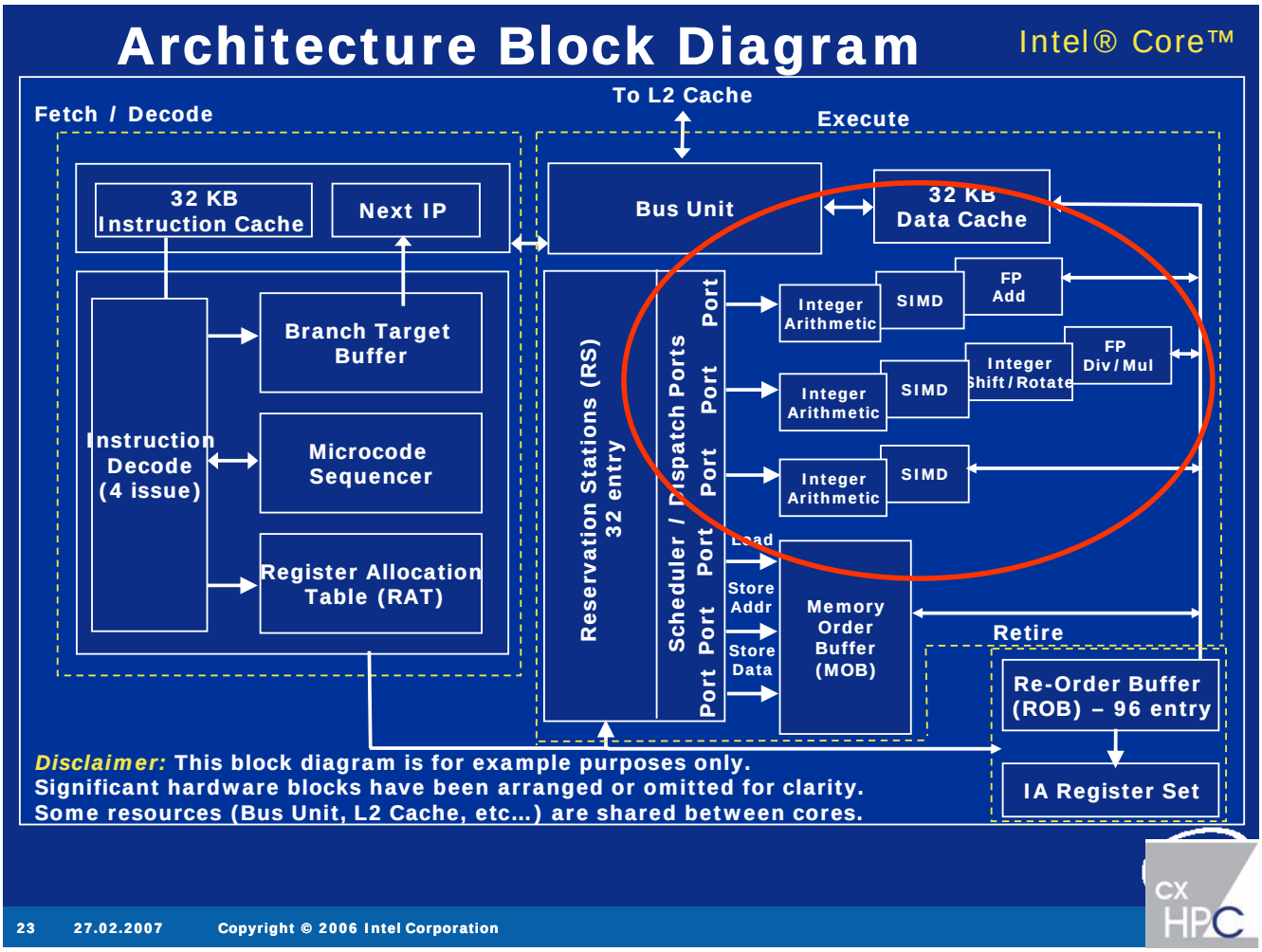
- ~2001: **Explicitly Parallel Instruction Computing (EPIC)** introduced
  - Compiler builds large group of instruction to be executed in parallel
  - First processors: **Intel Itanium1/2 using the IA64** instruction set.

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



Processor is built up by:

- 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



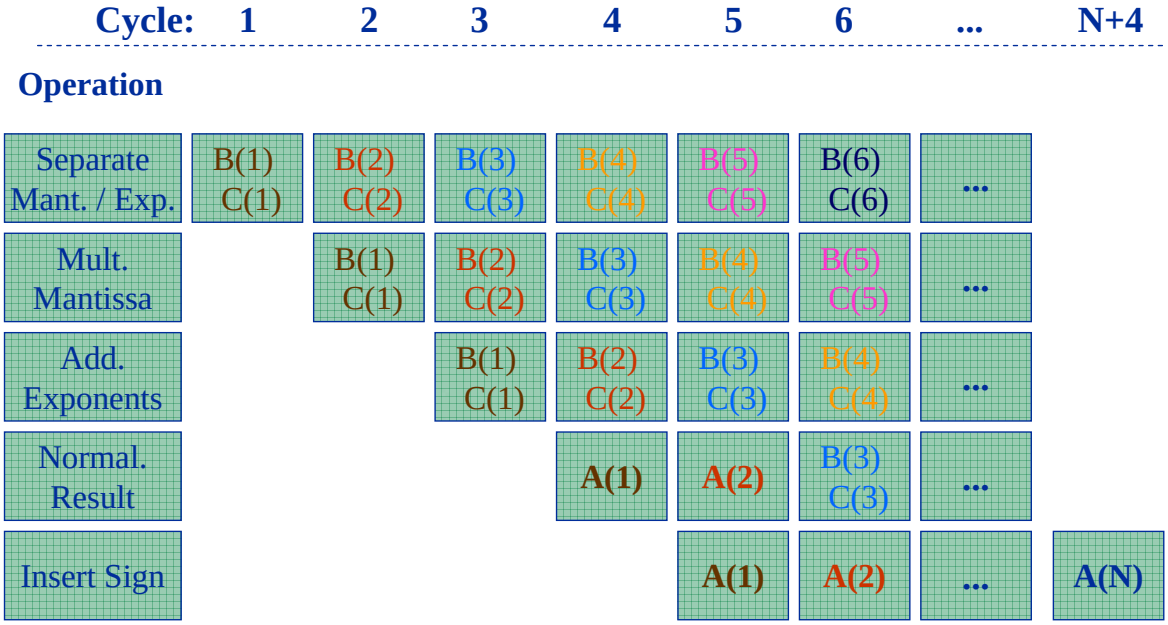
Architecture of modern microprocessors

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 this method excessively**

Pipelining

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

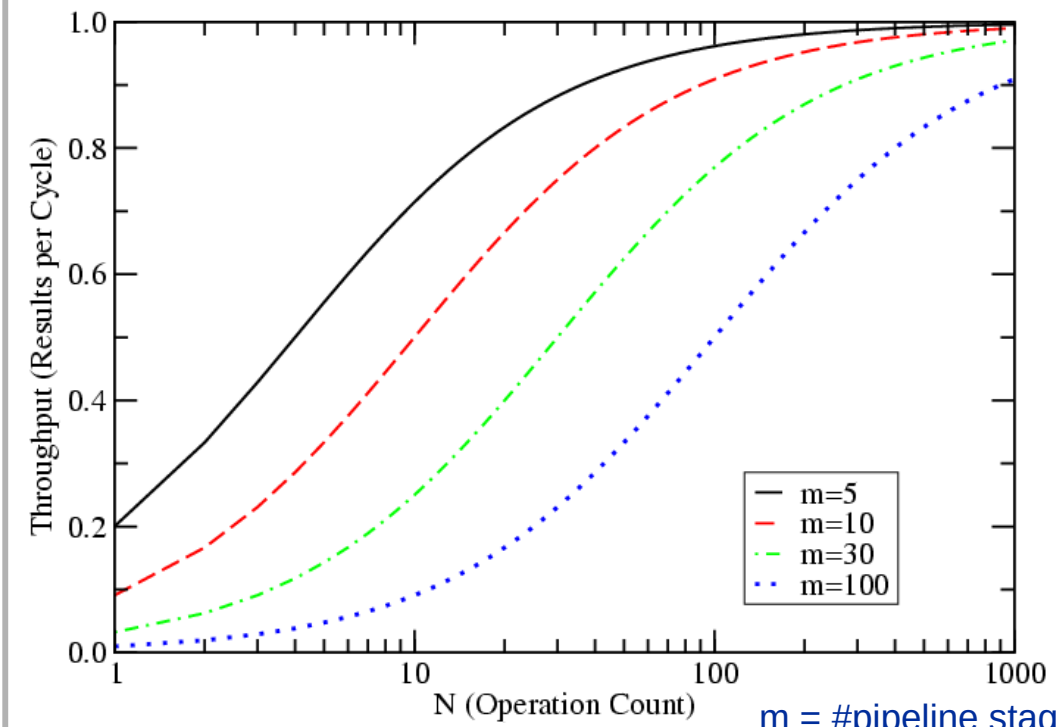
- Number of independent operations (N_C) required to achive T_p results per cycle:

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

  $T_p = 0.5$ ⇒ $N_C = m-1$

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)

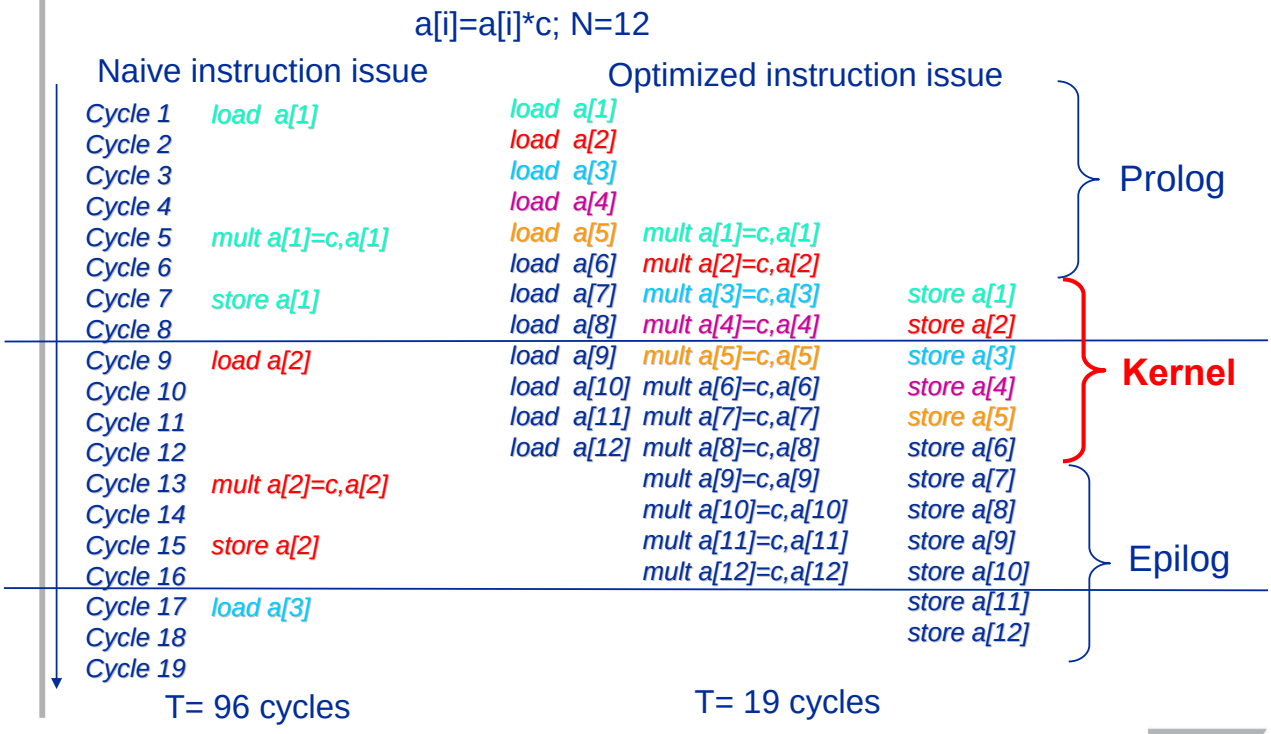
Simple Pseudo Code:
```
loop:   load a[i]
        mult a[i] = c, a[i]
        store a[i]
        branch.loop
```

→

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

Pipelining

Software pipelining



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:

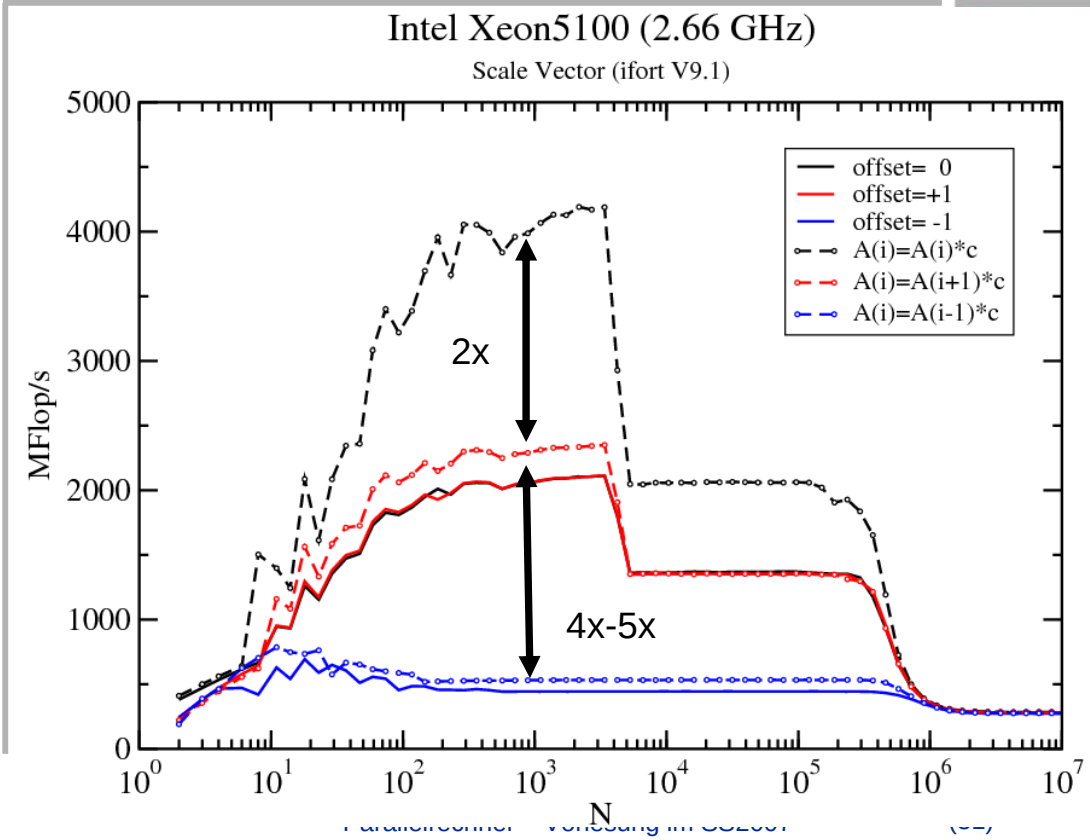
```
do i=1,N-1
   a(i) = a(i+1) * c
end do
```

General version (offset as input parameter):

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

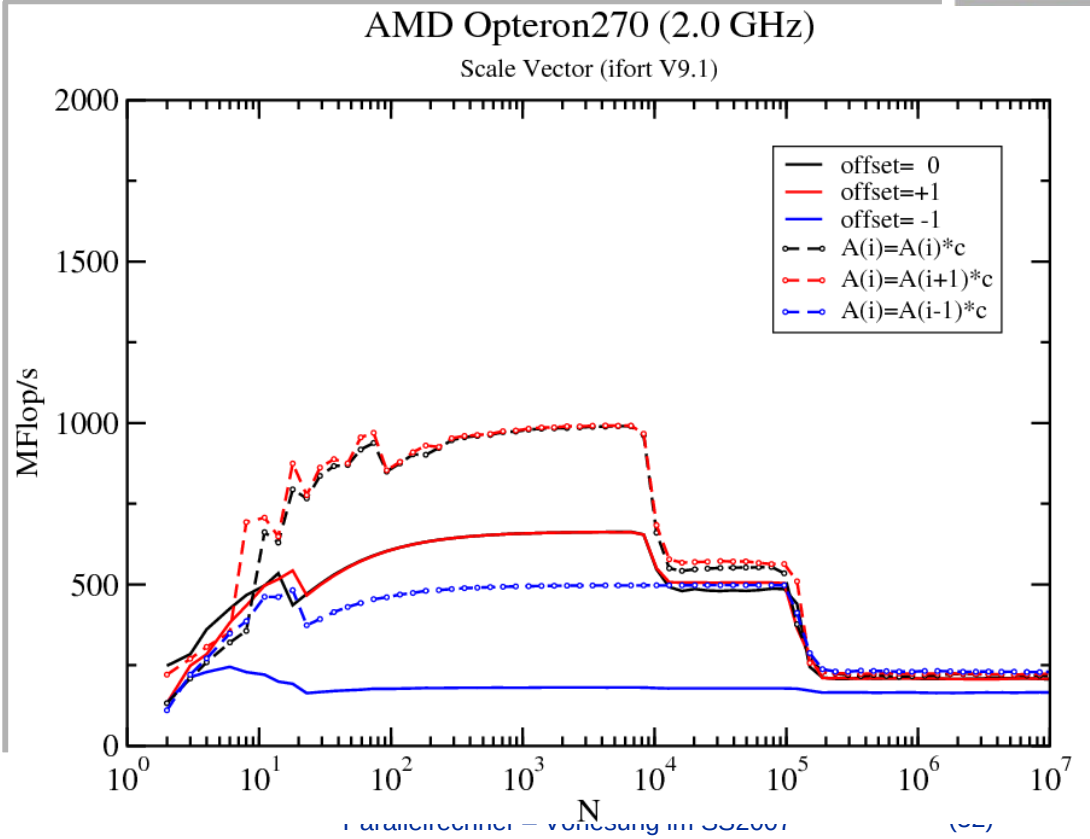
Pipelining

Data dependencies

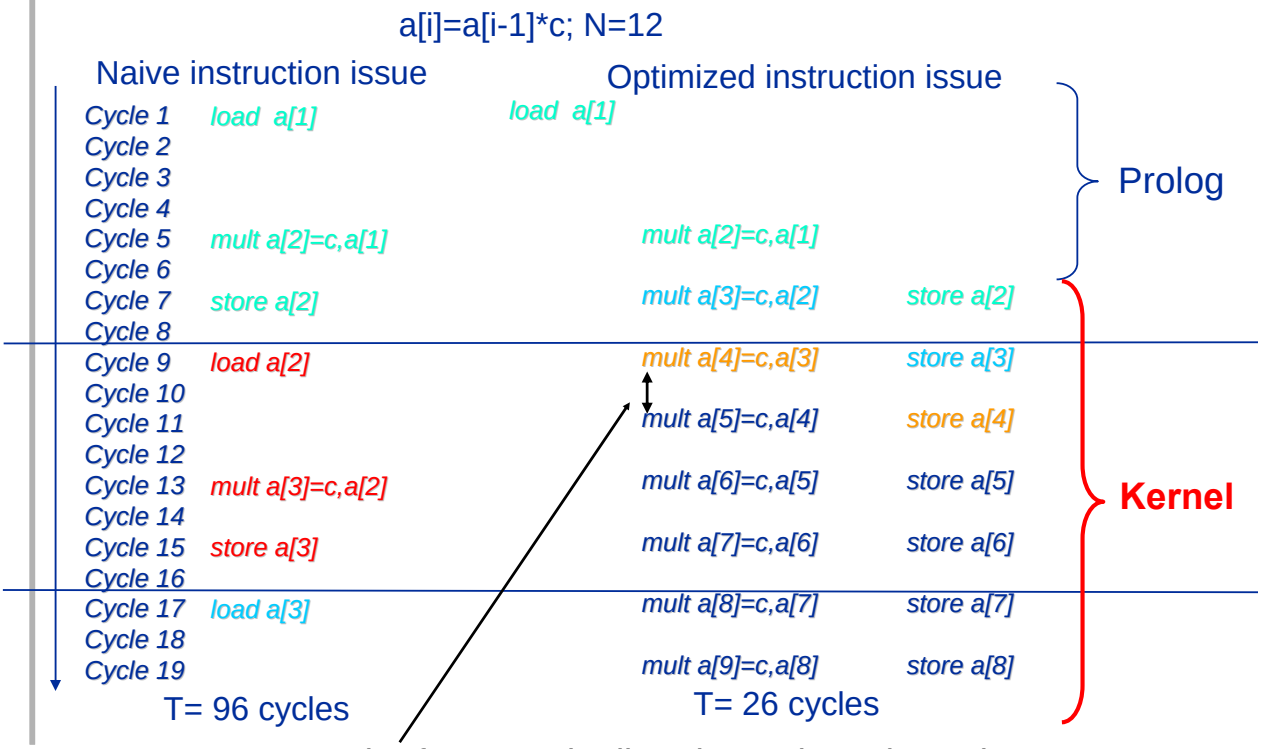


Pipelining

Data dependencies



Pipelining
Data dependencies



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

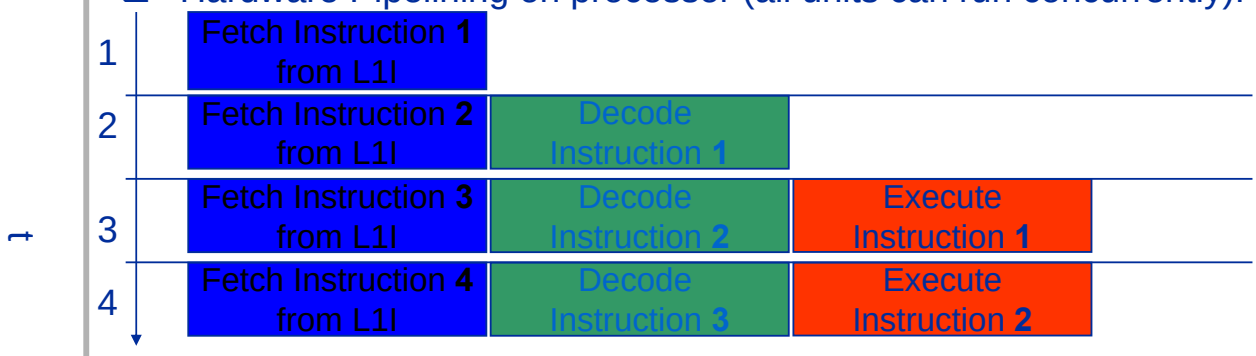
```
do i=1, N
    call elementprod(A(i), B(i), partsum)
    sum=sum+partsum
enddo
...
function elementprod( a, b, sum)
...
sum=a*b
```

  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):



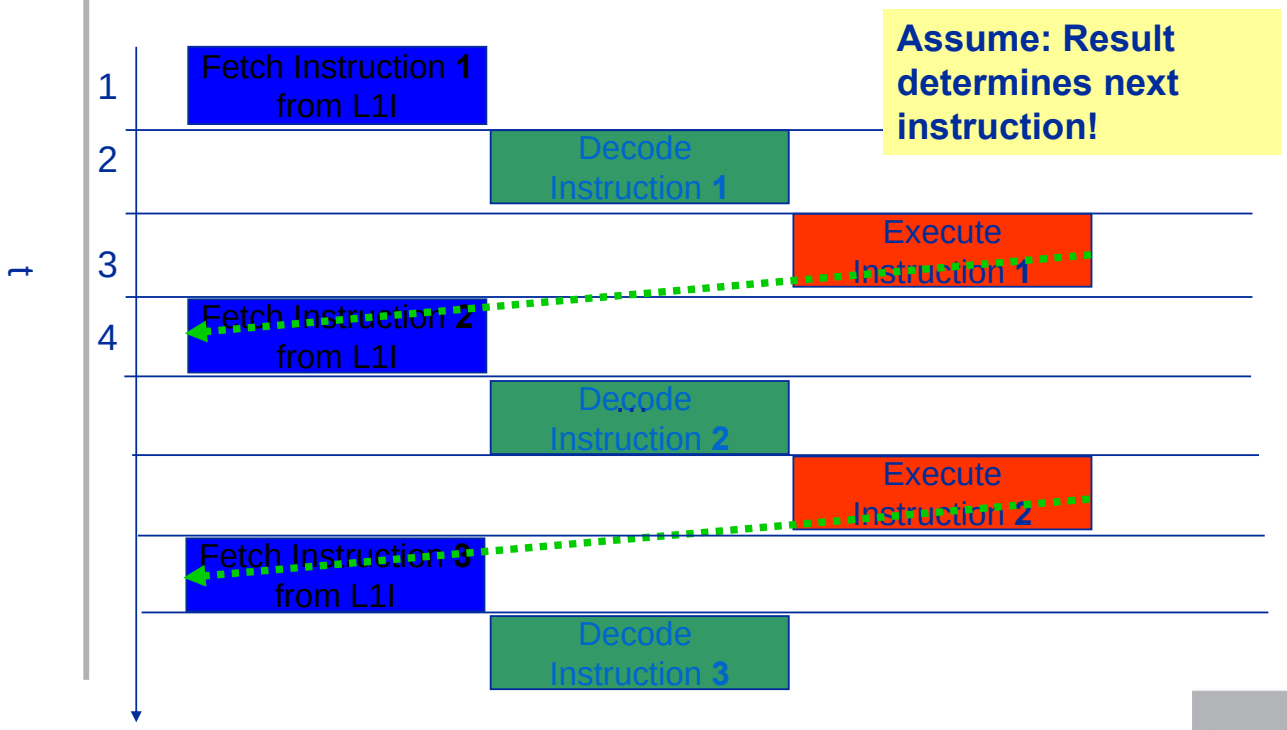
…

❑ 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

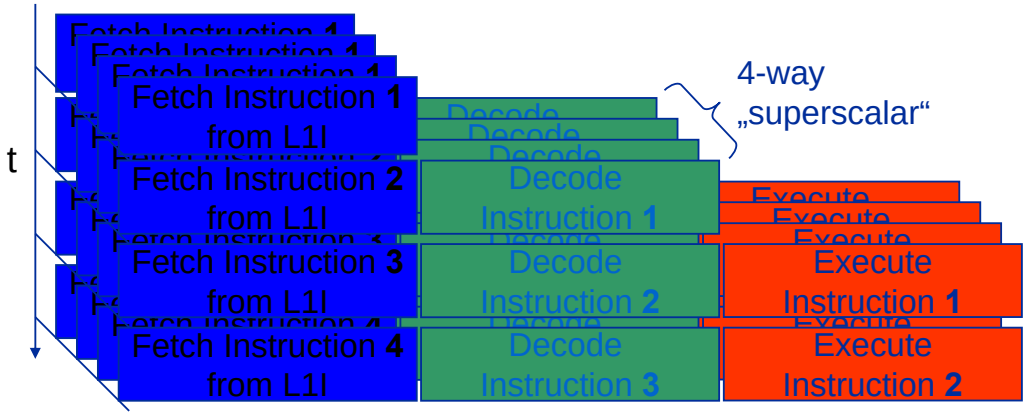


Pipelining
Superscalar Processors

- 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

„Load-after-Store dependency“

- **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

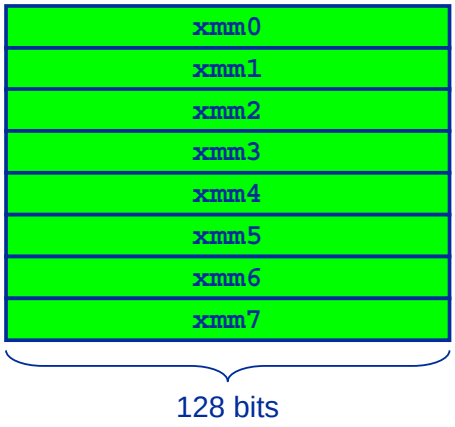
	Intel P4/Netburst	Intel Core
FP units	**1 MULT & 1 ADD pipeline**	
Width of operands	**128 Bit**	
FP ops/unit	**2 DP or 4 SP**	
Throughput	**2 cycles**	**1 cycle**
Max. FP ops/cycle	**2 DP or 4 SP**	**4 DP or 8 SP**
Latency of FP units (FPMULTD)	**7 cycles**	**5 cycles**

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

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

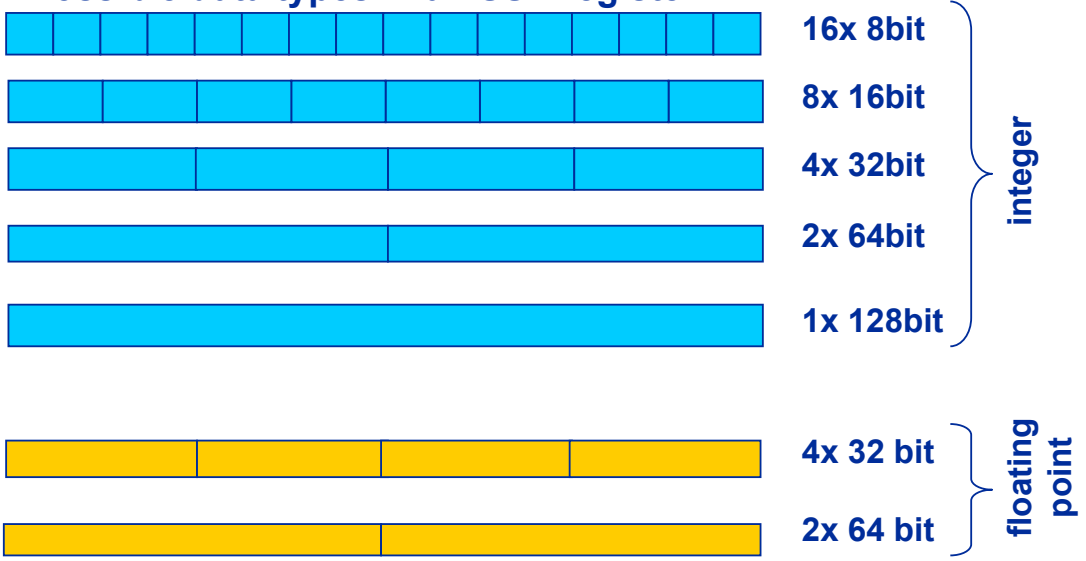
Throughput: Repeat rate of instruction issue,
e.g. 1 cycle -> in each cycle an new operation can be started

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



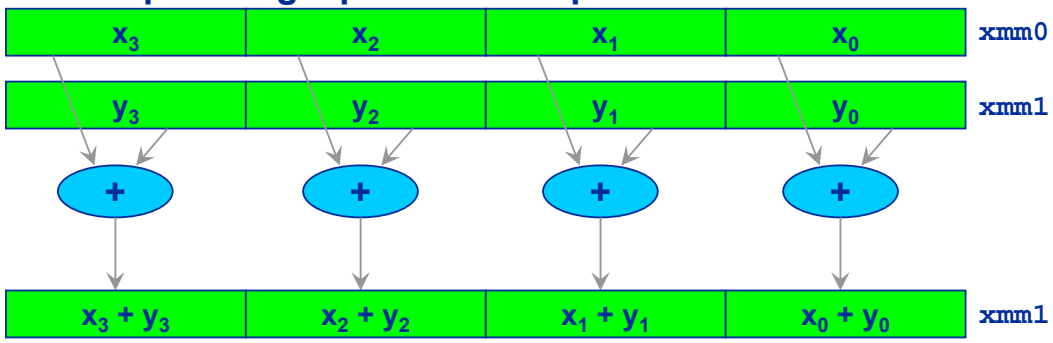
- **Each register can be partitioned into several integer or FP data types**
  - **8 to 128-bit integers**
  - **single (SSE) or double precision (SSE2) floating point**
- **SIMD instructions can operate on the lowest or all partitions („Packed SSE“) of a register at once**

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



Pipelining
Superscalar PCs – SSE

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



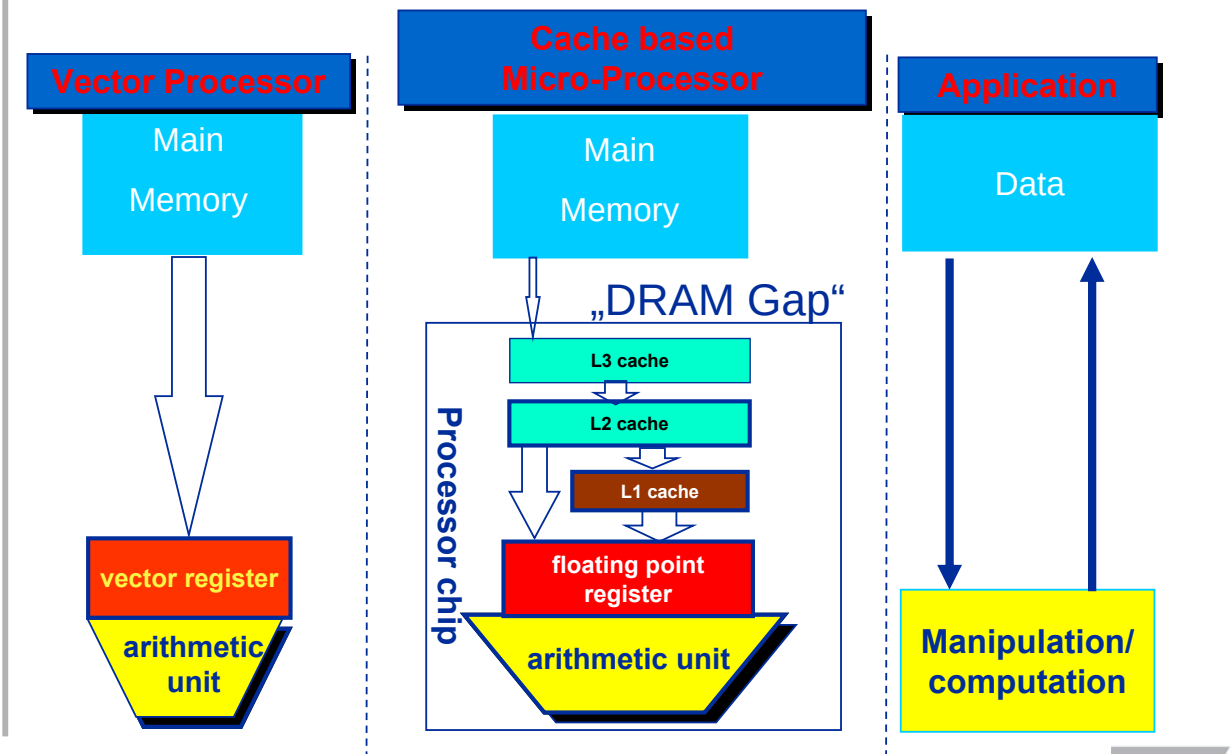
- **Four single precision FP additions with one single instruction**
- **Packed SSE -> Code vectorizatzion is a must**
- **Vectorization only possible if data are independent**
- **Automatic vectorization by compiler (appropriate compiler flag needs to be set) or forced by programmer (via directive)**

Pipelining
Efficient Use of Pipelining

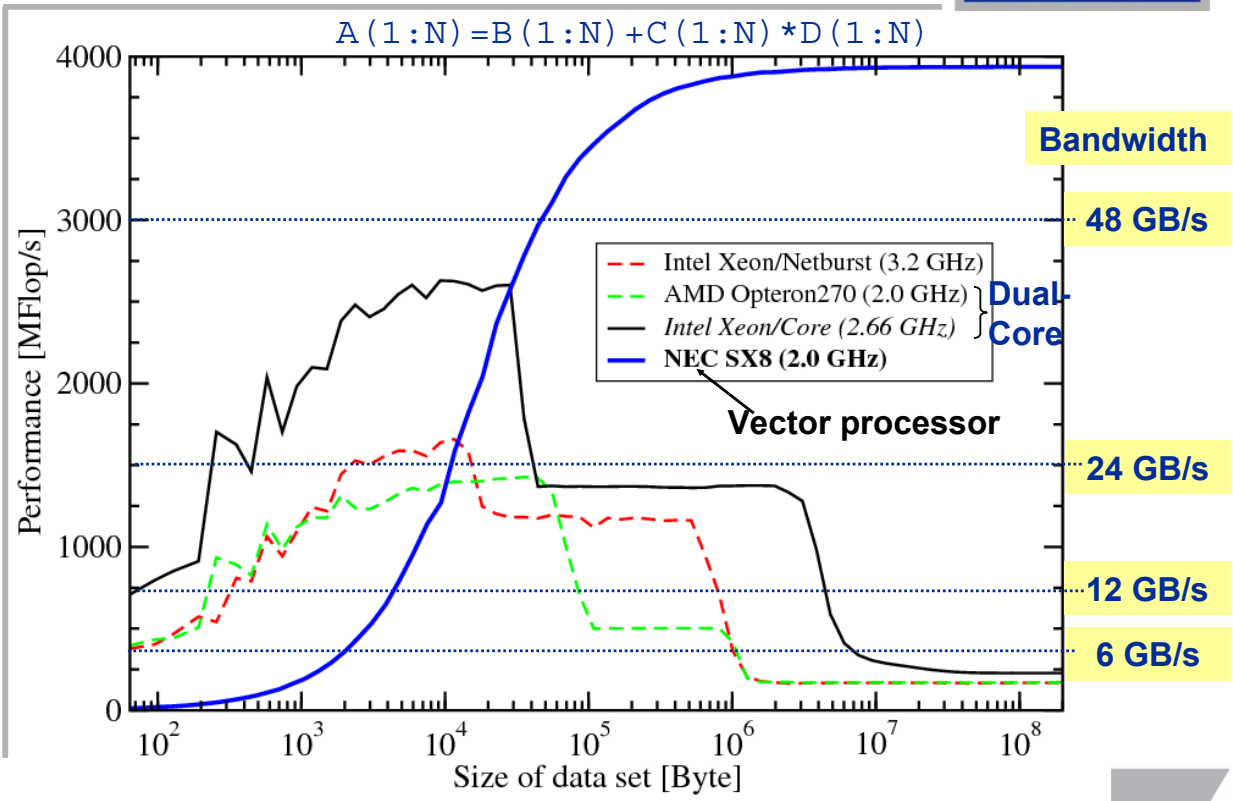
- Efficient use of pipelining/ILP requires intelligent compilers
  - Rearrangement of instructions to hide latencies
  - „Software pipelining“
  - Remove interdependencies that block parallel execution
- Programmer should
  - Avoid unpredictable branches (stop & restart of instruction pipeline!)
  - Avoid Data dependencies (if possible)
  - Tell compiler that instructions are independent
    (e.g. do not use pointer aliasing: -fno-alias with intel compiler)
- Long FP pipeline is inefficient for very small loops
  - Pipeline must be filled, i.e. long start-up times
- 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		8 / 16*	**8 / 16***	**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	**10 cycles**	**8 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).*

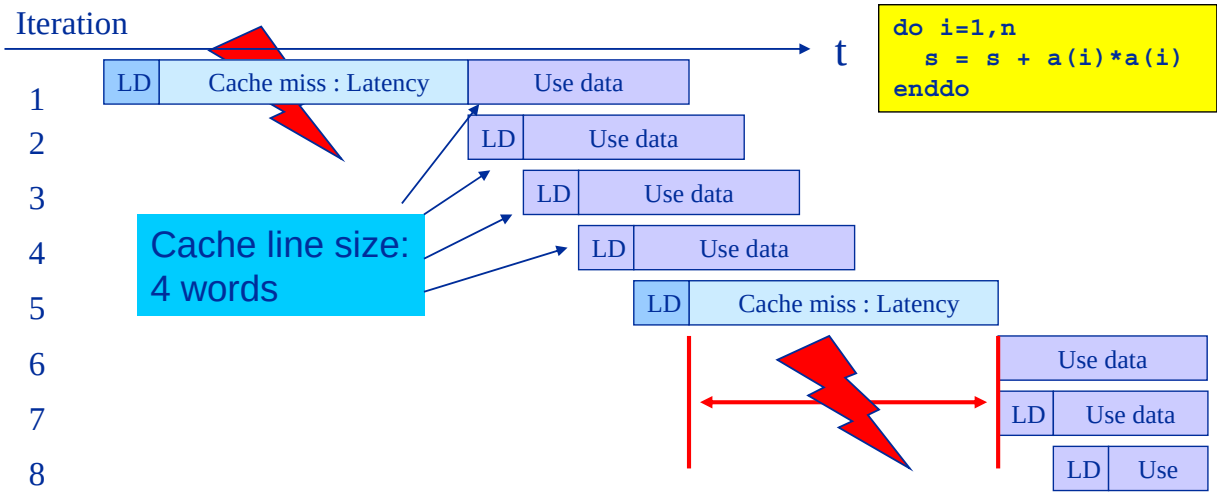
$$\text{Transfer time: } T = T_{lat} + \text{(amount of data)} / BW$$

- 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

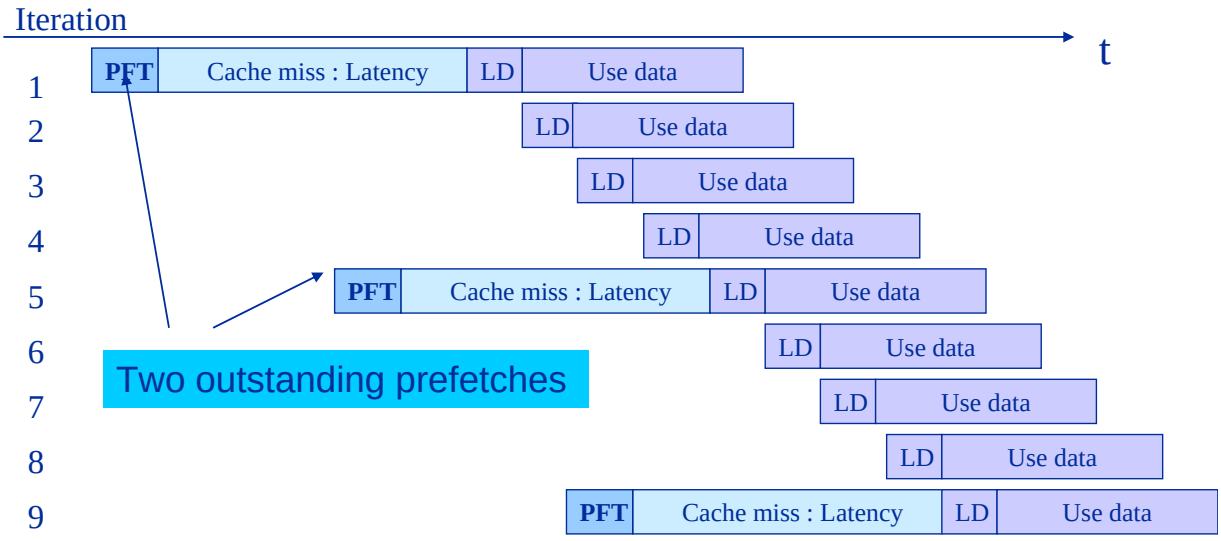
- *Prefetch (PFT) instructions:*
  - Transfer of consecutive data (one cache line) from memory to cache
  - Followed by LD to registers
  - Useful for executing loops with consecutive memory access
  - Compiler has to ensure correct placement of PFT instructions
    - Knowledge about memory latencies 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!

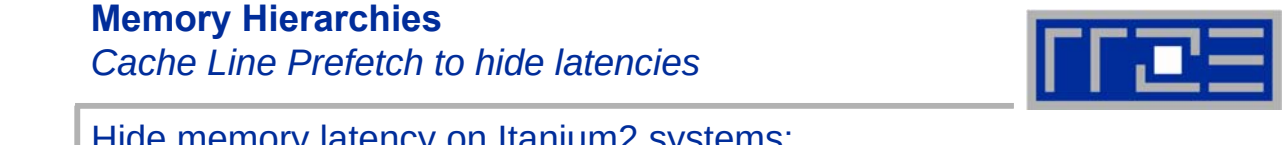
Hardware assisted prefetching for long contiguous data accesses

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



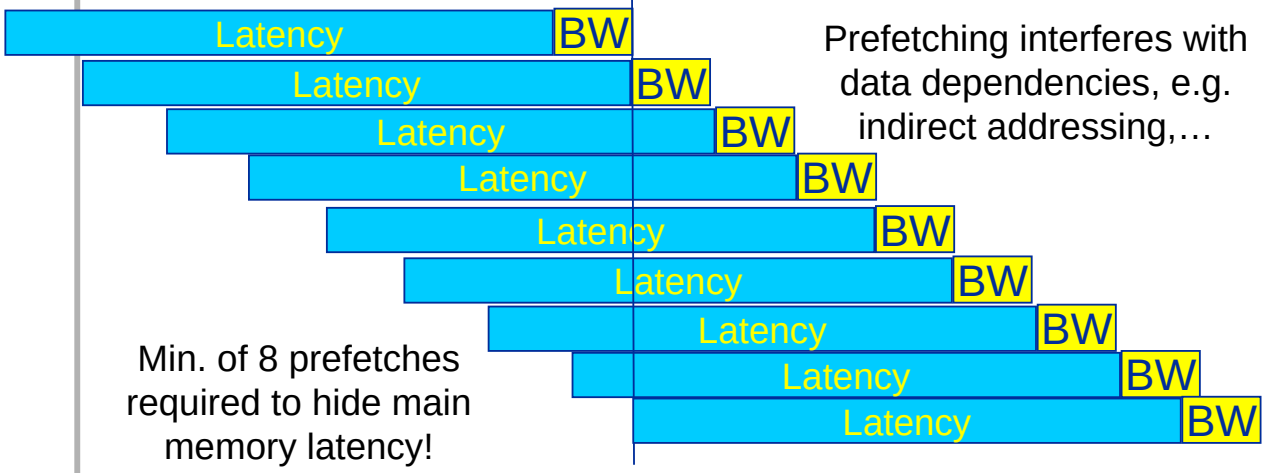
Intel Itanium2/EPIC: Software pipelining using PFT operations

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



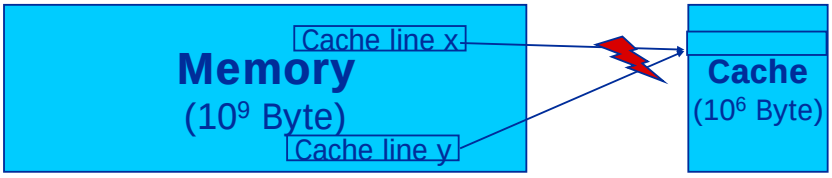
Prefetching interferes with data dependencies, e.g. indirect addressing,…

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 cache has to be overwritten ("invalidated") on next cache load using different strategies:

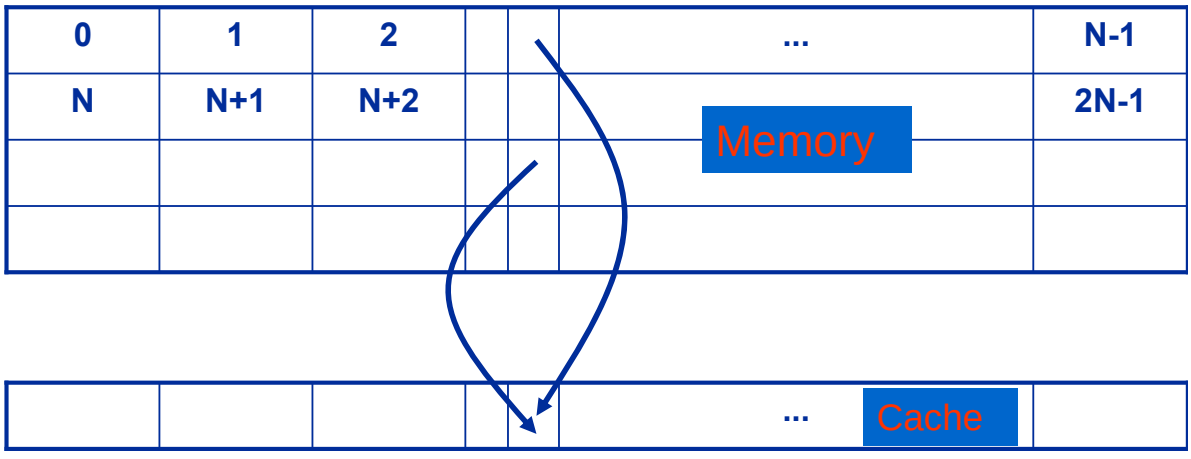
  least recently used (LRU) – random - not recently used

  May introduce 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



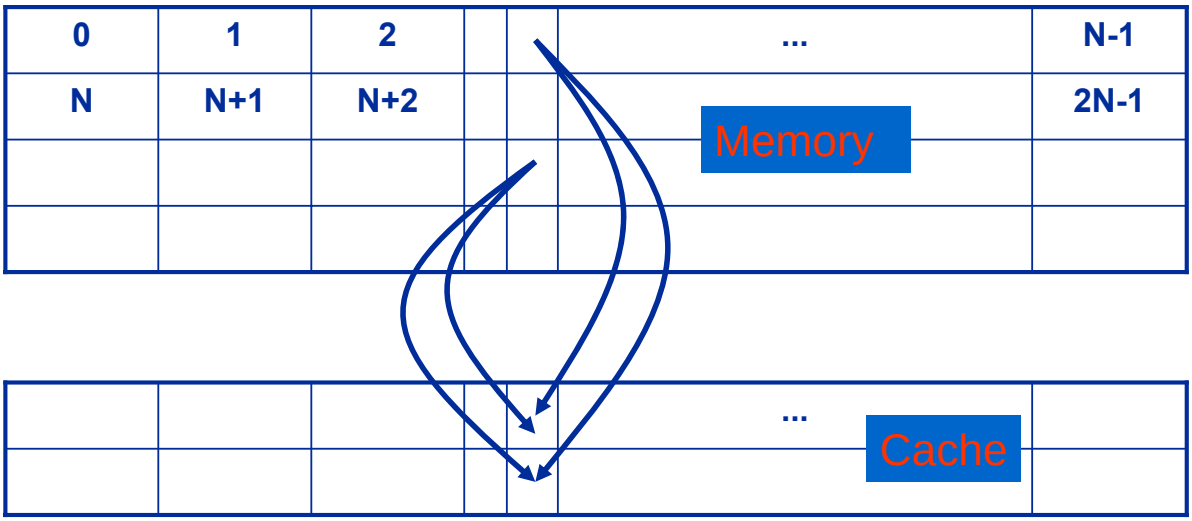
Example: Directly mapped cache. Each memory location can be mapped to one cache location only.

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

- **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!



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

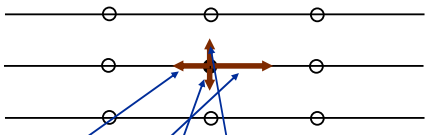
Warning: Using powers of 2 in the leading array dimensions of multi-dimensional arrays should be avoided! (Cache Thrashing)

- 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

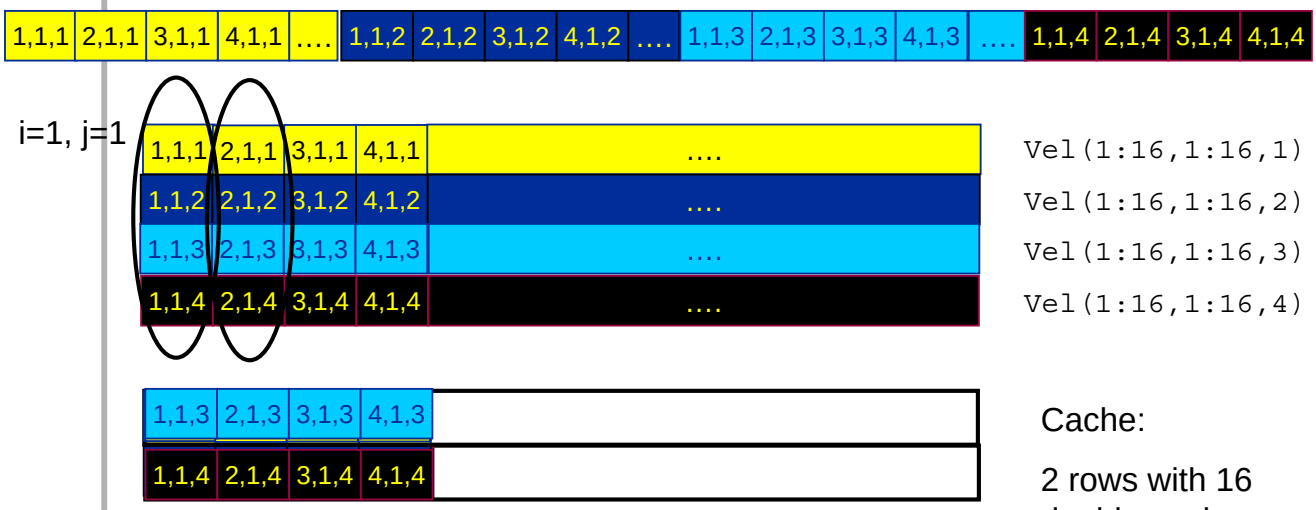
Example: 2D – square lattice
At each lattice point the 4 velocities for each of the 4 directions are stored



```
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

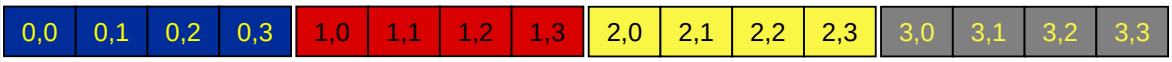
Each cache line needs only be loaded **once** from memory to cache!

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

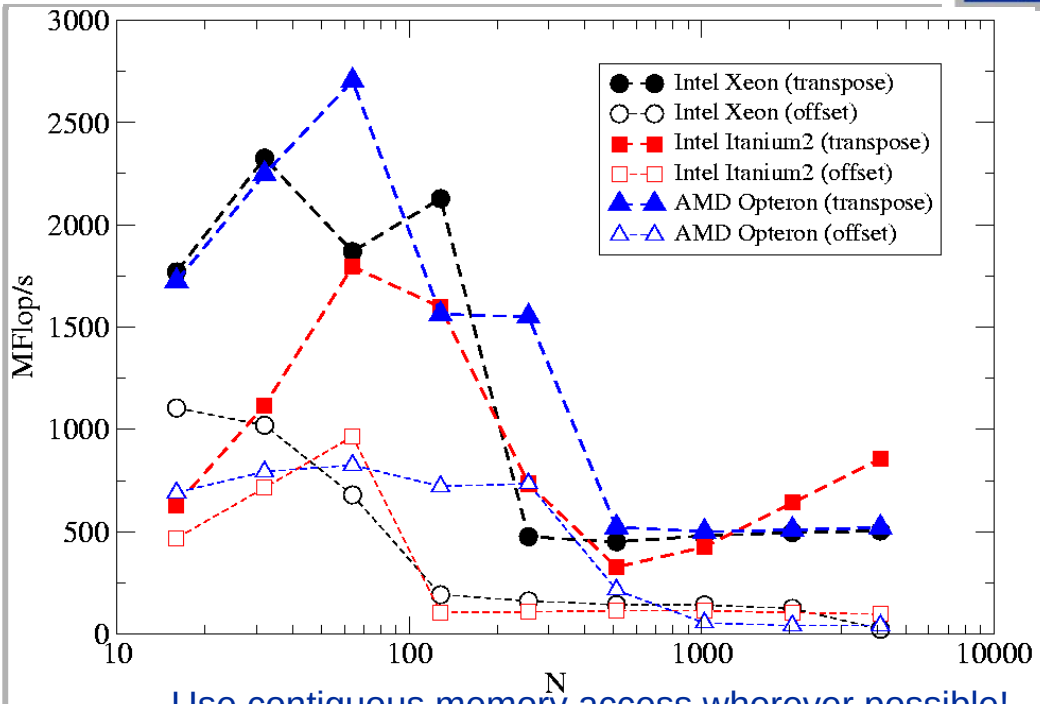
Innermost loop:
three loads and four flops: **0.75 W/Flop**
(1.5 N^2 data transfers)

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

Watch register spill!

Minimize Memory References

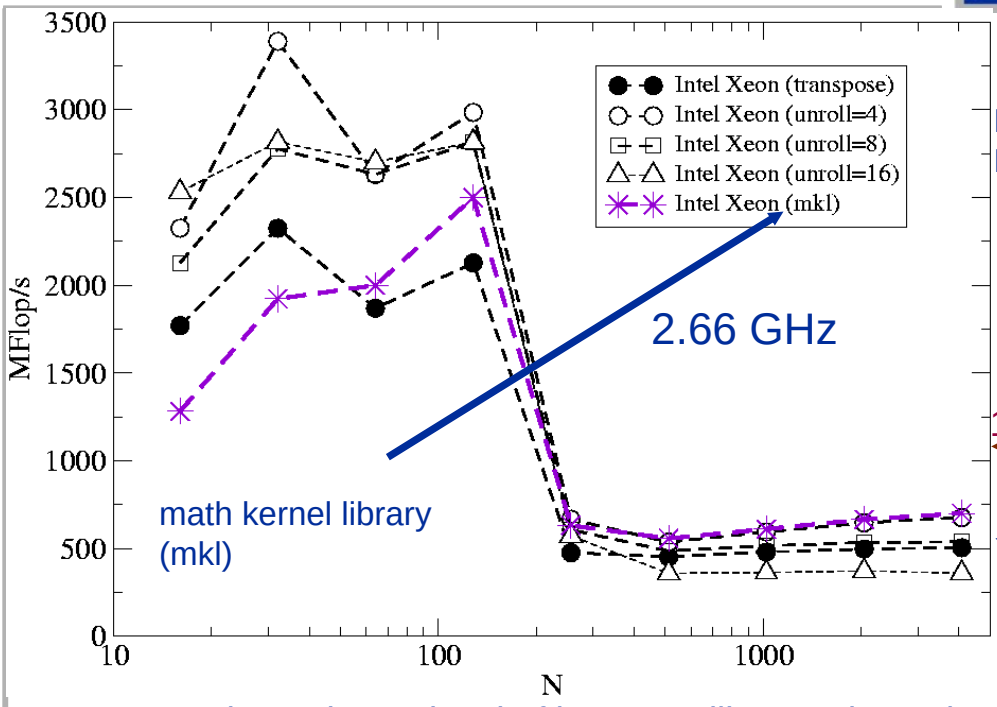
Dense MVM: Plain programming



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

Minimize Memory References

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



Performance From memory

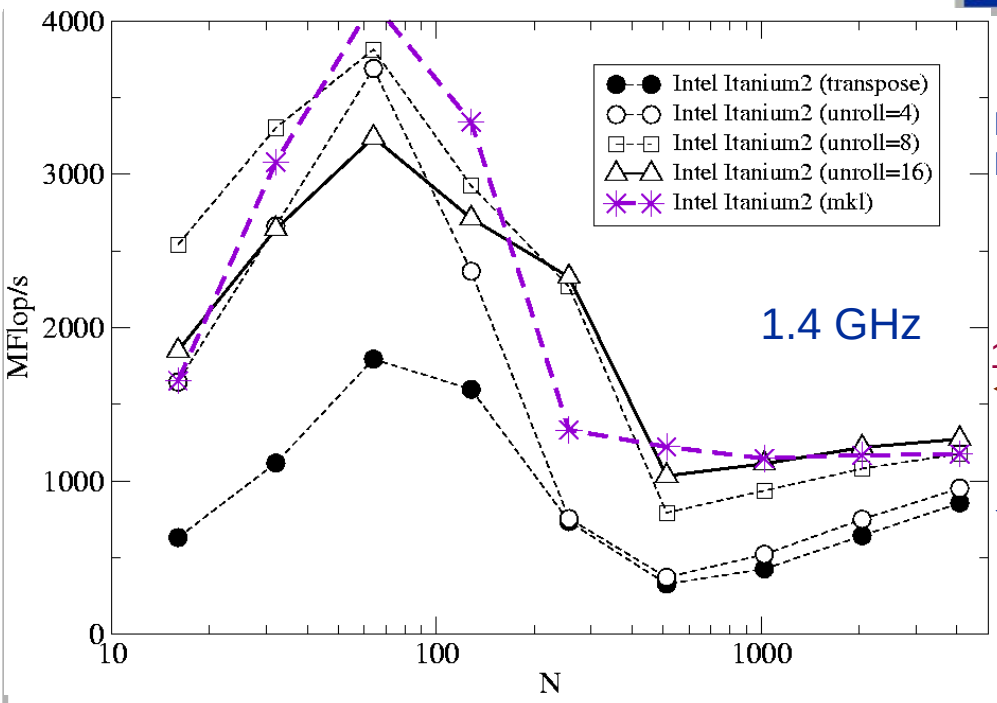
N^2: 1066 MFlop/s

2 N^2: 533 MFlop/s

Use only moderate level of loop unrolling: register shortage

Minimize Memory References

Dense MVM: Outer loop unrolling – Intel Itanium2



Performance From memory

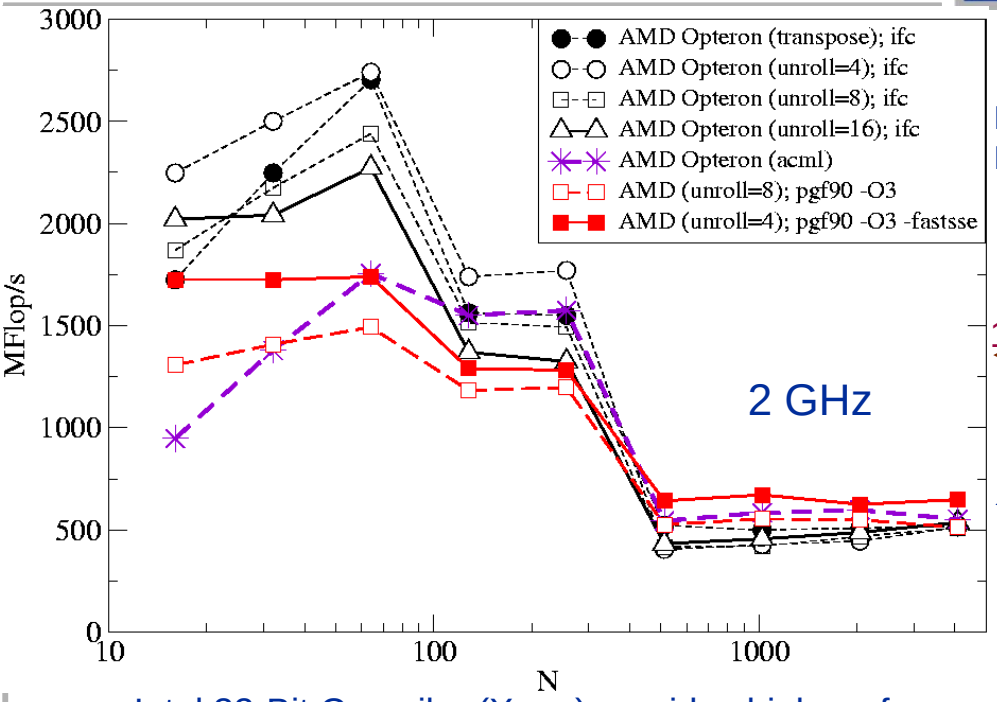
N^2: 1600 MFlop/s

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?

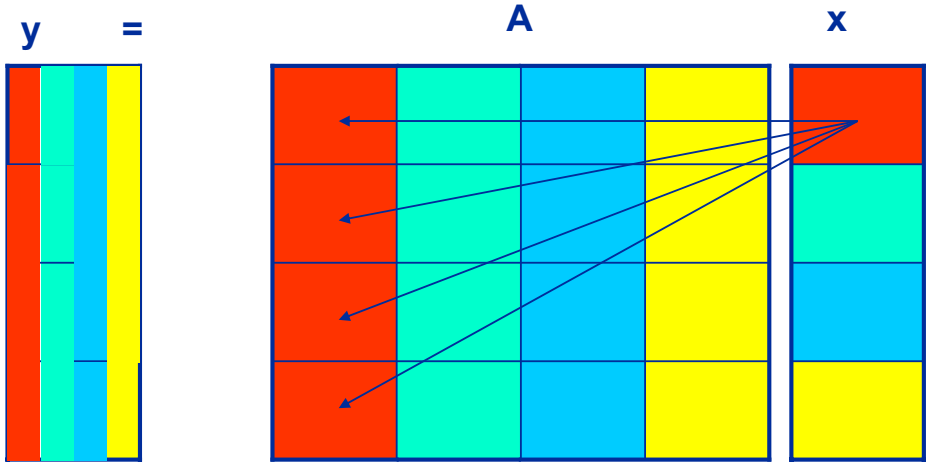
Vector x is loaded only once and (re-)used cacheline by cacheline.

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):

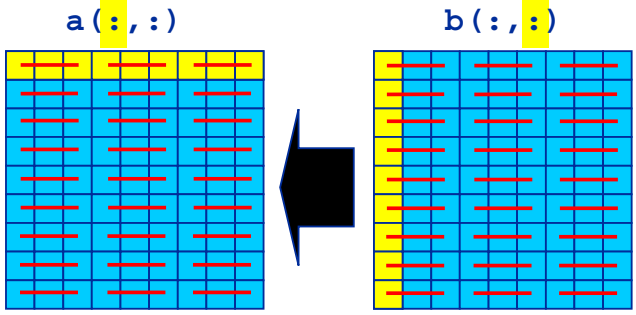
- Vector x is loaded only once instead of **N** times
- Vector y is loaded **nb** times instead of only once

Optimize Matrix transpose !

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

- **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
```

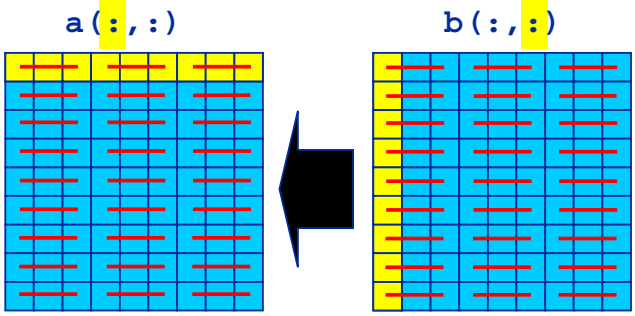


- **Problem: Stride-1 access for `a` implies stride-N access for `b`**
  - Access to `a` is perpendicular to cache lines (—)
  - Possibly bad cache efficiency (spatial locality)
- **Remedy: Outer loop unrolling and blocking**

Minimize Memory References
Exercise: Dense matrix transpose

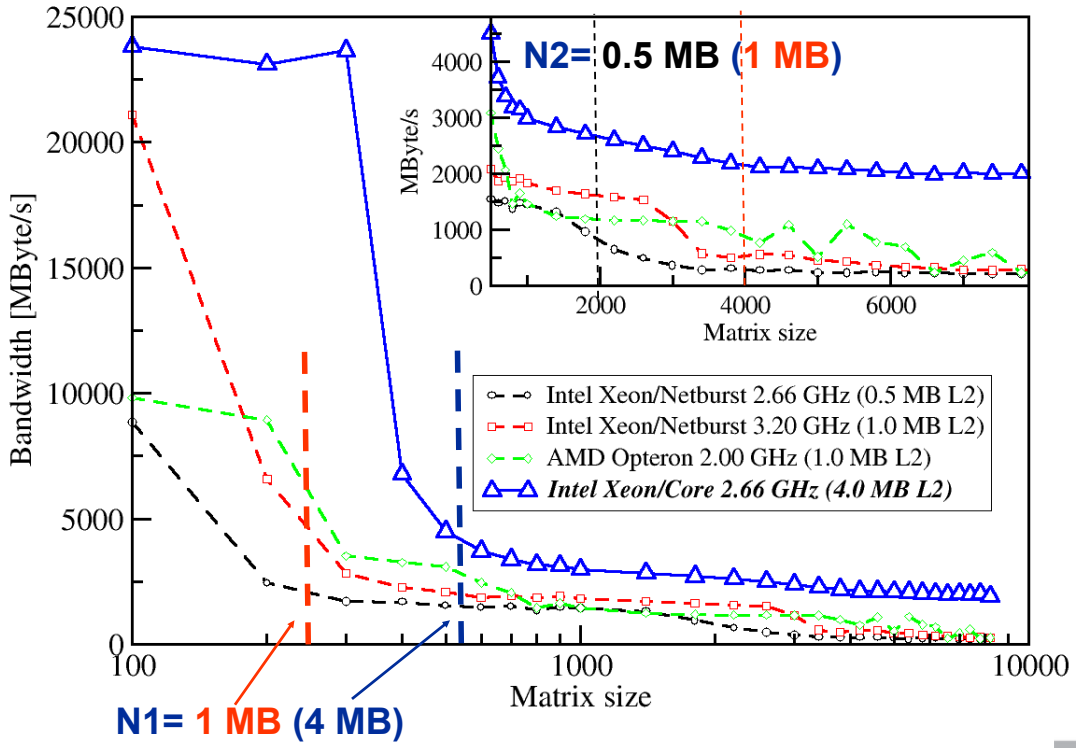
- **Data transfer analysis**

```
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
```

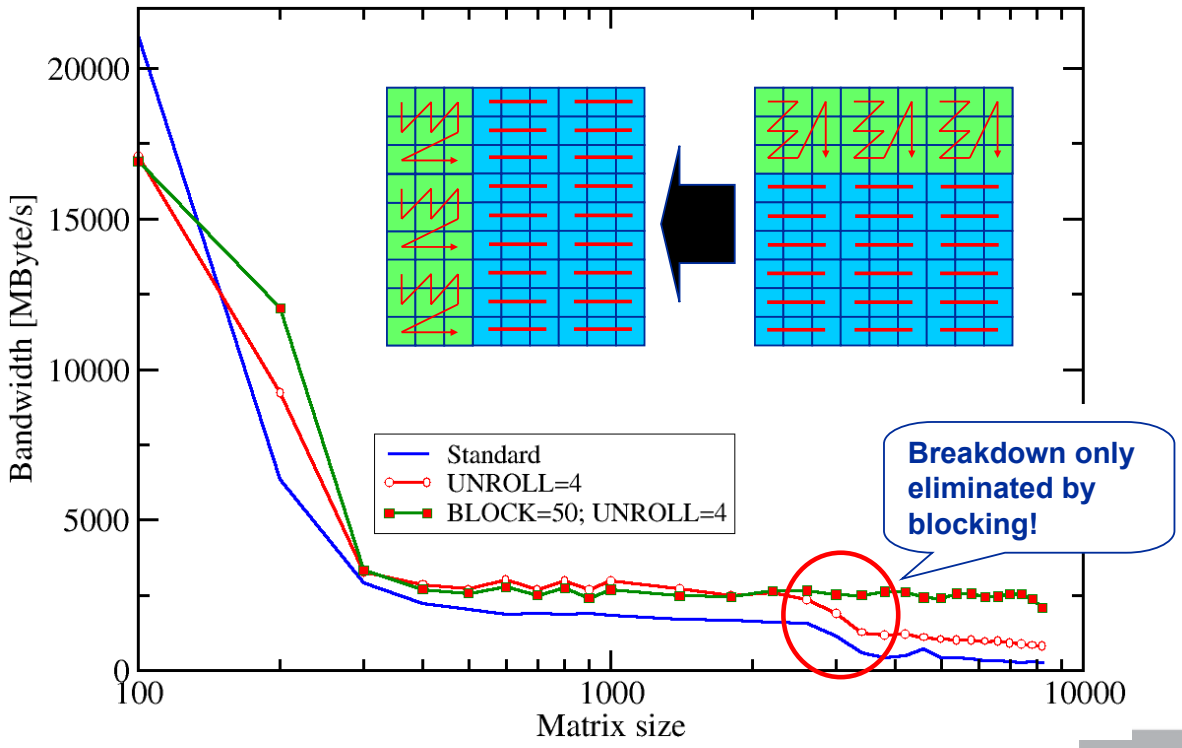
block

```
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
```

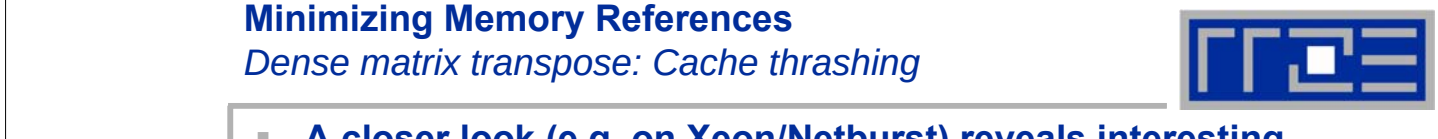
Blocking and unrolling factors (B,U) can be determined experimentally; be guided by cache sizes and line lengths

Minimizing Memory References

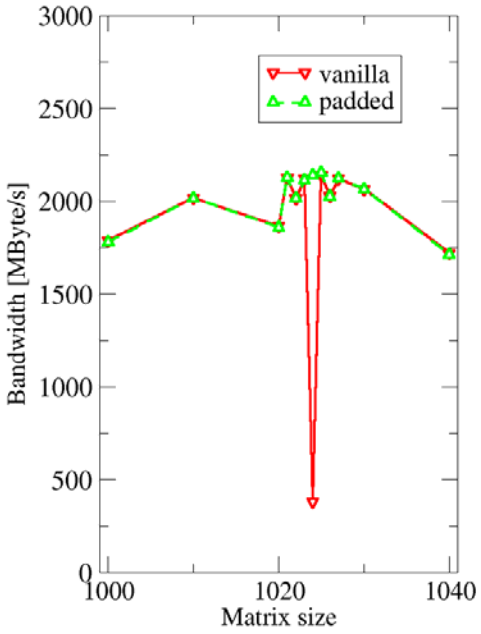
Dense matrix transpose: Block/unroll - Intel Xeon 3.2 GHz



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

- 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)