



# Introduction to High Performance Computing



# Performance

Which performance delivers a CPU?

Which measure is adequate?

- In numerical calculations the number of floating point operations per second is a reasonable measure (abbreviated FLOPS).
- How many CPU-cycles are required to finish one operation?

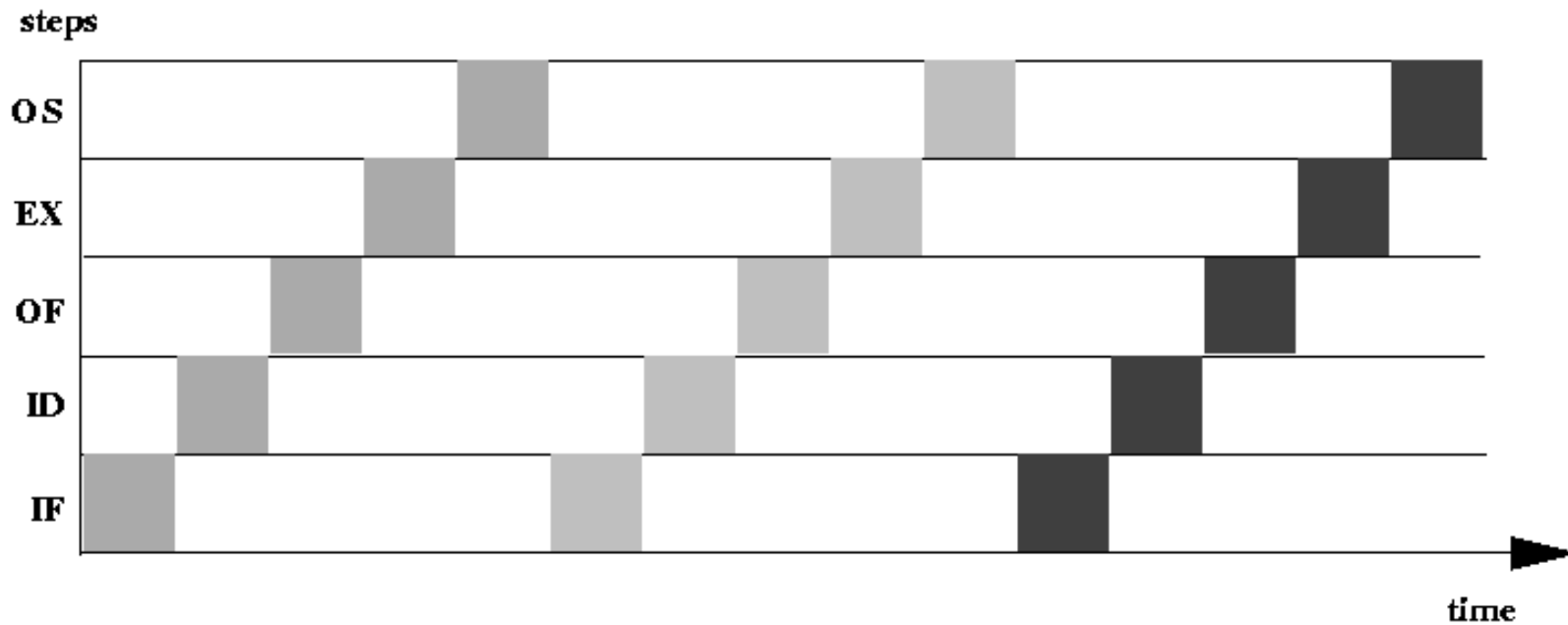
# Operations

How many cycles are necessary to compute  
 $y = a + x$  ?

Introduce schematic steps for further analyzation:

	Name	Description
IF	instruction fetch	the instruction is fetched from memory
ID	instruction decode	the instruction is decoded
OF	operand fetch	operands are fetched from memory
EX	execution	the instruction gets executed
OS	operand store	operands are stored in memory or registers

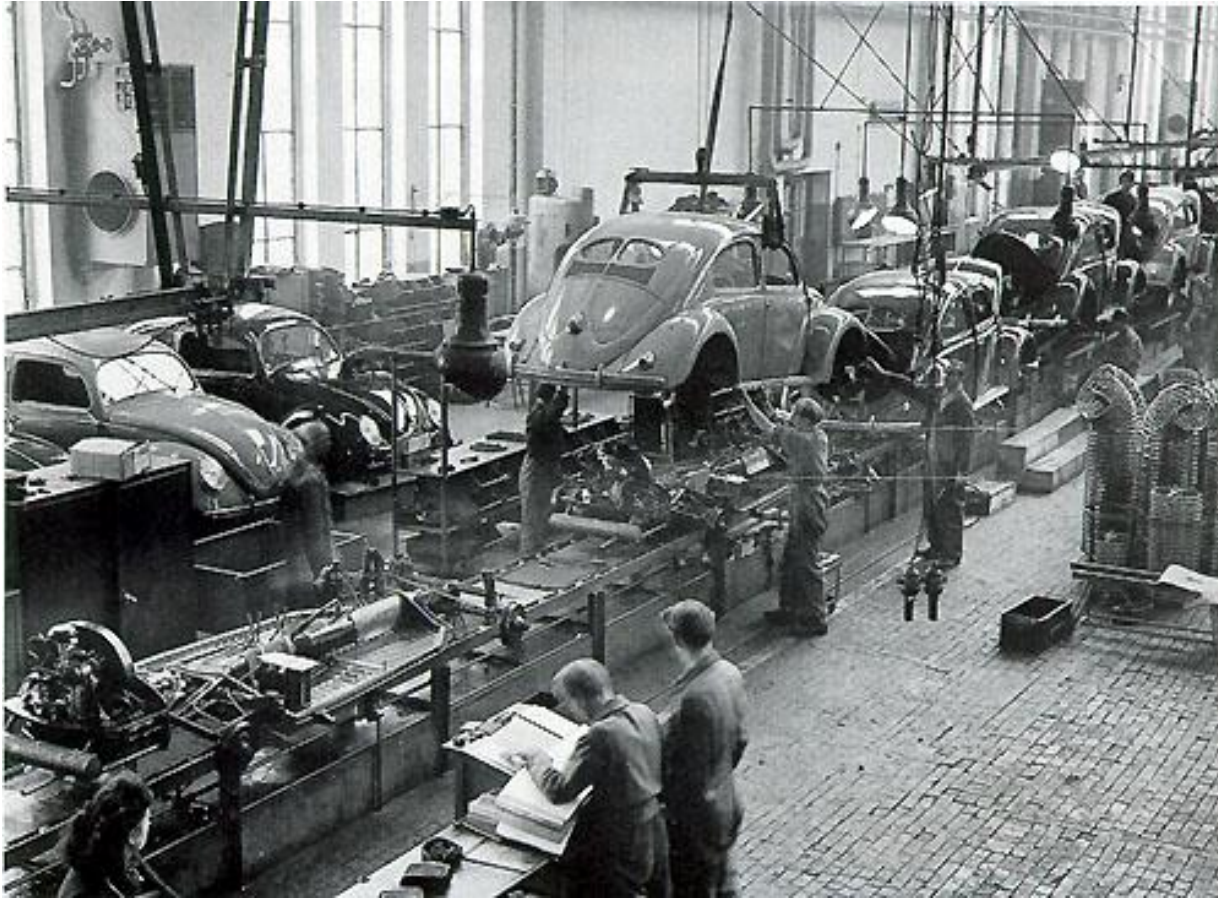
# Operations



processor cycles (one or several per step)



# Pipelining



Quelle: Stiftung Haus der Geschichte der Bundesrepublik Deutschland



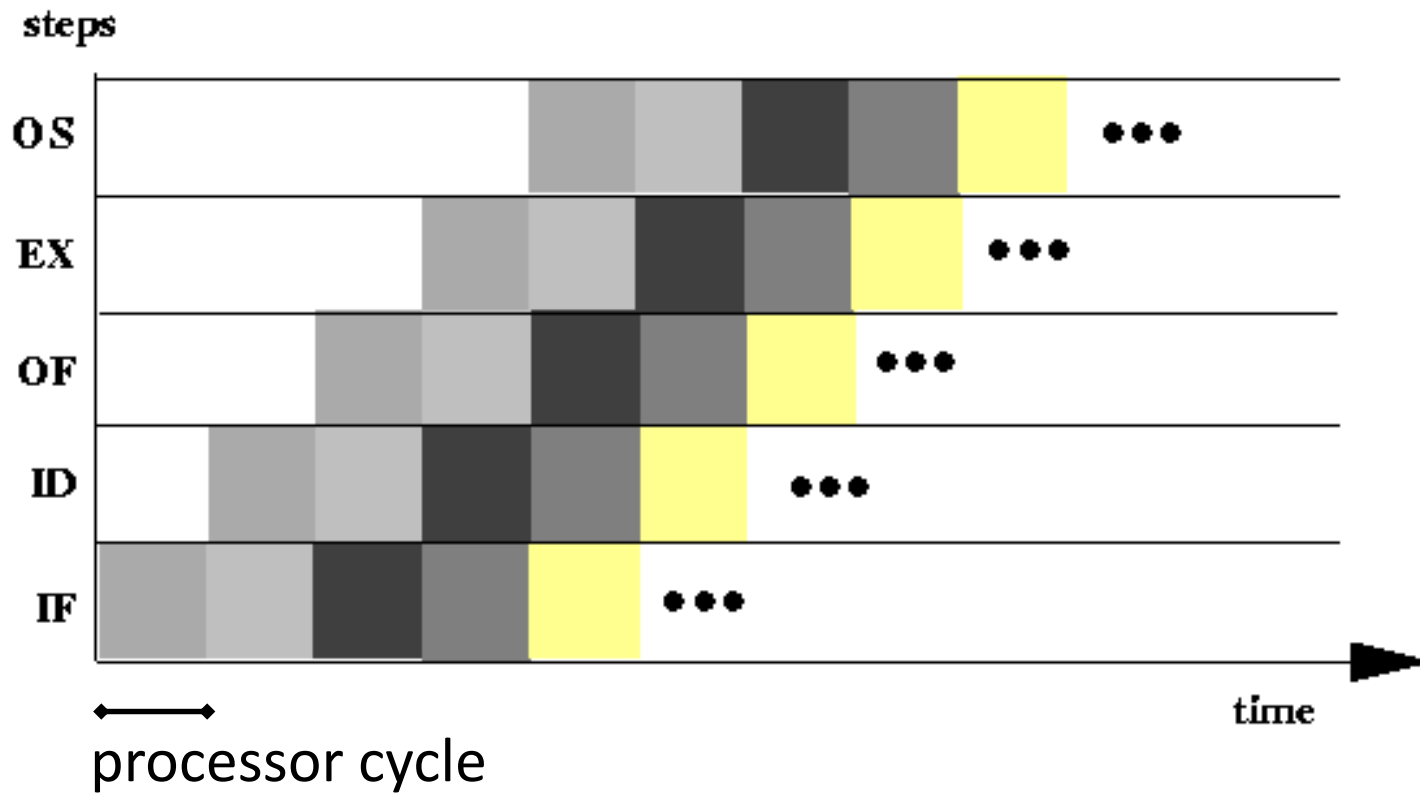
# Operations

assumption: 1 cycle per step

more details: D.A. Patterson and J.L. Hennesy, Computer Organization and Design, MK.

- **arithmetic latency:** estimated to 5 cycles, time to complete one specific single operation.
- **arithmetic throughput:** 2 operations per cycle (=peak), if ALUs are kept busy and consist of one  $*/+$  (multiply and add).

# Operations





# Operations

Thus the performance is the cycle time (1/2.6 GHz)?

## Assumptions

- unique clock cycles per instruction -  $CPI=1$
- #ALUs in one processor core -  $\#ALUs=2$  (in most cases)
- #Ops per cycle of one processor core -  $\#ALUs/CPI$
- peak performance of a processor core -  $\#ALUs/CPU * \text{clock rate}$

Example:

2GHz, 2 ALUs, 4 processor cores:

$$\text{Peak performance} = 2 * 4 * (2 * 10^9) = 16 \text{ GFLOPS}$$



# Pipelining

a0= y+x;	
b0=a0*c;	Waits for a0
a1=a0+x;	Waits for a0
b1=a1*c;	Waits for a1
a2=a0+y;	Waits for a0
b2=a2*c;	Waits for a2

Data dependencies prevent pipelining.

What we can do:

- allow „out of order execution“
- turn on a proper optimization level
  - -O0 – no optimization, order preserved (debugging)
  - -O1 – branch prediction, delayed branch, ....
  - -O2 – alignment, reordering
  - -O3 – optimize yet more

In benchmarks, production code – always use the highest optimization level possible!



# Pipelining

What we can do:

- arrange if-branches
  - if possible prevent random conditions (not predictable)
  - allow out-of-order execution
- avoid function calls
  - unconditional branch
  - pipelines are emptied at start and end of the function
  - evacuate small functions (use preprocessor)
  - use inlining

What the hardware does:

- branch prediction
  - speculative calculation of most probable branch
  - if wrong predicted, results are discarded

# Pipelining

What we can do:

- increase arrangeable statements to the compiler
  - loop unrolling

```
for(i=0,max=0.;i<n;i++)  
    if(fabs(a[i]) > max) max=fabs(a[i]);
```

```
max=m0=m1=m2=m3=0.;  
for(i=0; i<n;i+=4) {  
    a0=fabs(a[i+0]) ; a1=fabs(a[i+1]) ;  
    a2=fabs(a[i+2]) ; a3=fabs(a[i+3]) ;  
    if(a0 > m0) m0=a0;      if(a1 > m1) m1=a1;  
    if(a2 > m2) m2=a2;      if(a3 > m3) m3=a3;  
}  
m0=m0>?m1;      m2 = m2>?m3;  
max=m0>?m2;
```

Alternative:  
#pragma unroll  
automatic AVX or SSE

# Processor


E5-2670  
installed: 2012,2014



Intel® Xeon® Processor E5-2670  
(20M Cache, 2.60 GHz, 8.00 GT/s Intel® QPI)

## Spezifikationen

### Hauptdaten


Status	End of Life
Einführungsdatum	Q1'12
Voraussichtliche Produkteinstellung	Q2'15
Prozessornummer	E5-2670
Intel® Smart Cache	20 MB
Intel® QPI	8 GT/s
Anzahl der QPI-Links	2
Befehlssatz	64-bit
Erweiterungen des Befehlssatzes	AVX
Embedded-Modelle erhältlich	 No
Lithographie	32 nm
Skalierbarkeit	2S Only
VID-Spannungsbereich	0.60V-1.35V
Empfohlener Kundenpreis	TRAY: \$1552.00 BOX : \$1556.00
Datenblatt	<a href="#">Link</a>

### Leistung

Anzahl der Kerne	8
Anzahl der Threads	16
Grundtaktfrequenz des Prozessors	2.6 GHz
Max. Turbo-Taktfrequenz	3.3 GHz
Verlustleistung (TDP)	130 W

# Processors

E5-2670  
installed: 2012,2014

- Speicherspezifikationen	
Max. Speichergröße (abhängig vom Speichertyp)	384 GB
Speichertypen	DDR3 800/1066/1333/1600
Max. Anzahl der Speicherkanäle	4
Max. Speicherbandbreite	51,2 GB/s
Unterstützung von ECC-Speicher †	 Yes
+ Erweiterungsoptionen	
- Formatspezifikationen	
Max. CPU-Bestückung	2
T <sub>CASE</sub>	80.0°C
Gehäusegröße	52.5mm x 45.0mm
Geeignete Sockel	FCLGA2011
Halogenarme Modelle erhältlich	Siehe MDDS
- Innovative Technik	
Intel® Turbo-Boost-Technik †	2.0
Intel® vPro-Technik †	 Yes
Intel® Hyper-Threading-Technik †	 Yes
Intel® Virtualisierungstechnik (VT-x) †	Yes
Intel® Directed-I/O-Virtualisierungstechnik (VT-d) †	 Yes
Intel® VT-x mit Extended Page Tables (EPT) †	 Yes
Intel® 64 †	 Yes
Inaktivitätsstatus	Yes
Erweiterte Intel SpeedStep® Technologie	 Yes
Intel® Demand-based-Switching	 Yes
Thermal-Monitoring-Technik	Yes
Intel® Flex-Memory-Access	No
Intel® Identity-Protection-Technik †	No



# Processors

E5-2640 v3  
installed: 2016

Summer

Source: ark.intel.com



Intel® Xeon® Processor E5-2640 v3  
(20M Cache, 2.60 GHz)

## Spezifikationen

### Hauptdaten


Status	Launched
Einführungsdatum	Q3'14
Prozessornummer	E5-2640V3
Intel® Smart Cache	20 MB
Intel® QPI	8 GT/s
Anzahl der QPI-Links	2
Befehlssatz	64-bit
Erweiterungen des Befehlssatzes	AVX 2.0
Embedded-Modelle erhältlich	Yes
Lithographie	22 nm
Skalierbarkeit	25
VID-Spannungsbereich	0.65V–1.30V
Empfohlener Kundenpreis	BOX : \$944.00 TRAY: \$939.00
Datenblatt	<a href="#">Link</a>
Produktbeschreibung	<a href="#">Link</a>
URL für weitere Informationen	<a href="#">Link</a>

### Leistung

Anzahl der Kerne	8
Anzahl der Threads	16
Grundtaktfrequenz des Prozessors	2.6 GHz
Max. Turbo-Taktfrequenz	3.4 GHz
Verlustleistung (TDP)	95 W

# Processors

E5-2640 v3  
installed: 2016

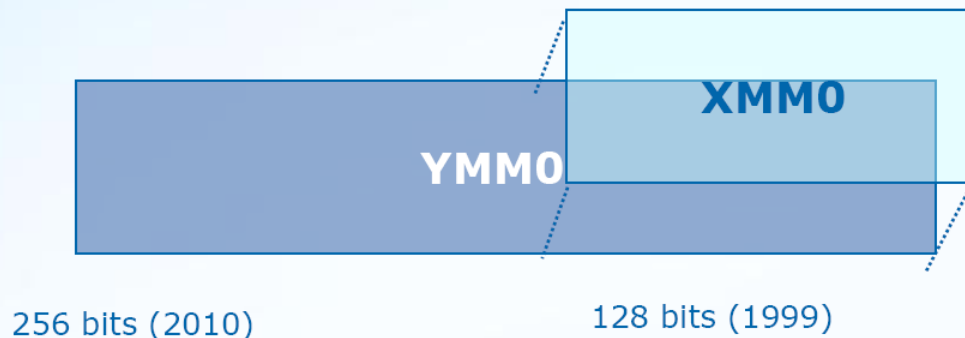
- Speicherspezifikationen	
Max. Speichergröße (abhängig vom Speichertyp)	768 GB
Speichertypen	DDR4 1600/1866
Max. Anzahl der Speicherkanäle	4
Max. Speicherbandbreite	59 GB/s
Erweiterungen der phys. Adresse	46-bit
Unterstützung von ECC-Speicher †	 Yes
+ Grafikspezifikationen	
+ Erweiterungsoptionen	
+ Formatspezifikationen	
- Innovative Technik	
Intel® Turbo-Boost-Technik †	2.0
Intel® vPro-Technik †	 Yes
Intel® Hyper-Threading-Technik †	 Yes
Intel® Virtualisierungstechnik (VT-x) †	Yes
Intel® Directed-I/O-Virtualisierungstechnik (VT-d) †	 Yes
Intel® VT-x mit Extended Page Tables (EPT) †	 Yes
Intel® TSX-NI	No
Intel® 64 †	 Yes
Inaktivitätsstatus	Yes
Erweiterte Intel SpeedStep® Technologie	 Yes
Intel® Demand-based-Switching	 Yes
Thermal-Monitoring-Technik	Yes
Intel® Flex-Memory-Access	No
Intel® Identity-Protection-Technik †	No

# Advanced Vector Extensions

AVX introduces:

- parallel handling of 256 bit registers leading to a doubled peak performance
- advanced data rearrangement (mask loads, permute data, broadcast)
- flexible unaligned memory access

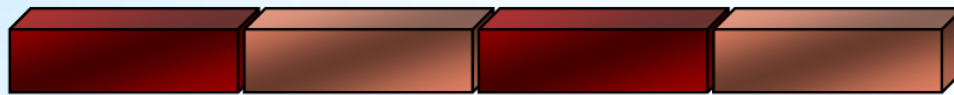
Intel® AVX extends all 16 XMM registers to 256bits



Source: Intel, compiling-for-avx-kb.pdf,  
2011

# Advanced Vector Extensions

**SSE**



**4x floats**

**SSE-2**



**2x doubles**



**16x bytes**



**8x 16-bit shorts**



**4x 32-bit integers**



**2x 64-bit integers**

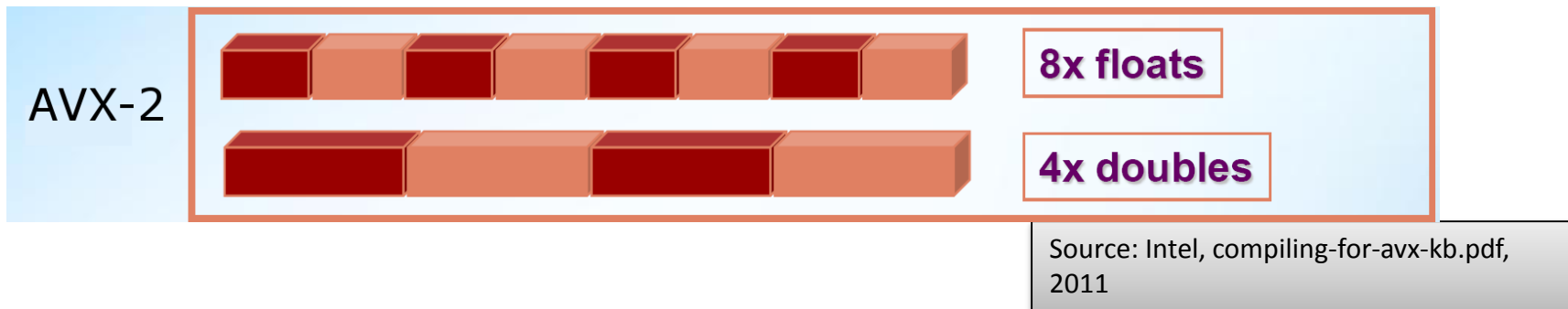


**1x 128-bit(!) integer**

Nehalem CPUs, old technology

Source: Intel, compiling-for-avx-kb.pdf, 2011

# Advanced Vector Extensions



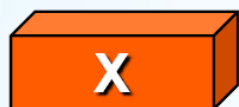
- Currently no integer 256 bit instructions
- main advantage for floating point operations
- some new memory loading functionality with AVX.2



# Advanced Vector Extensions

## Scalar mode

- one instruction produces one result



+



=



X



+

Y



=

X + Y



Source: Intel, compiling-for-avx-kb.pdf, 2011

# Advanced Vector Extensions

How to use:

- use newer gcc (4.4.1 or later) and an assembler > 2.20
- or INTEL compiler icc

- use option –xavx (icc)

- control vectorization of loops:

```
icc -O2 -vec-report3 -c source.c
```

- control assembly code:

```
icc -S -fargument-noalias -c source.c | grep ymm  
source.s
```



# Vectorization

- Must be an inner loop.
- Only inlined or intrinsic functions.
- Iteration count must be known at entry of loop.
  - No change of variable inside
  - No break
- Help with directives
  - `#pragma ivdep ....` (no data dependency)
  - `#pragma vector always` (even if compiler assumes non-efficiency)

# Vectorization

## Vectorization?

```
for (i=0; i<N; i++)  a[i]=a[i-n]+b[i];
```

- if  $n \leq 0$
- if  $n \geq N$  (#pragma ivdep)

```
for (i=0; i<N; i+=2)  a[i] += x[i] * b[i];
```

- inefficient – non-unit stride

```
for (i=0; i<N; i++) {  
    for (j=0; j<M; j++)  b[i] += a[j][i] * x[j];  
}
```

- inefficient – non-unit stride
- automatic interchanging loops is possible

# Vectorization

## Vectorization?

```
for (i=0; i<N; i++) a[i]=b[i]*x[ind[i]];
```

- inefficient due to indirect addressing of x

```
for (s=0., i=0; i<N; i++) s+=a[i];
```

- reductions are vectorizable

```
for (i=0; i<N; i++)  
    if (m>0) x[i]=x[i]/sqrt(m);
```

- masked operations are possible (if)
- inlining of intrinsic function is possible



# Vectorization

Problem with pointers – are they aliased?

```
void daxpy(int n, double a, double *x, double *y) {  
    for(int i=0; i<n; i++) y[i] += a*x[i];  
}
```

may be called:

```
daxpy(n-1, 4.5, &x[1], &x[0]);  
    results in x[i] += 4.5*x[i-1];
```

Thus, compiler is not able to vectorize loop in daxpy !

# Vectorization

```
void daxpy(int n, double a, double *x, double *y) {  
    #pragma ivdep  
    for(int i=0; i<n; i++) y[i] += a*x[i];  
}
```

compiled with options

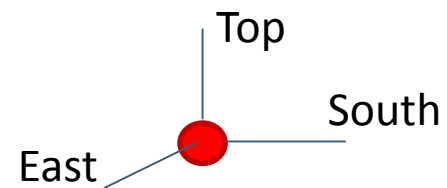
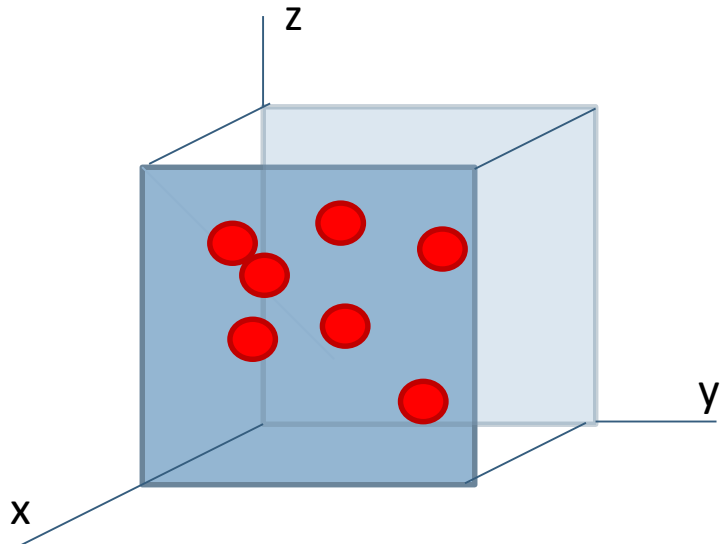
**-restrict or -std=c99**

# vectorization

## Exercise code

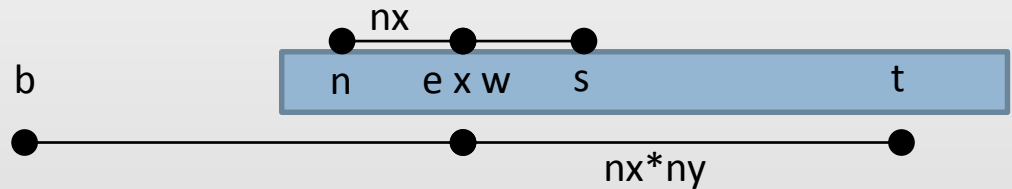
```
for (int z = 0; z < nz; z++) {  
    for (int y = 0; y < ny; y++) {  
        for (int x = 0; x < nx; x++) {  
            int c, w, e, n, s, b, t;  
            c = x + y * nx + z * nx * ny;  
            w = (x == 0) ? c : c - 1;  
            e = (x == nx-1) ? c : c + 1;  
            n = (y == 0) ? c : c - nx;  
            s = (y == ny-1) ? c : c + nx;  
            b = (z == 0) ? c : c - nx * ny;  
            t = (z == nz-1) ? c : c + nx * ny;  
            f2[c] = cc * f1[c] + cw * f1[w] + ce * f1[e]  
                + cs * f1[s] + cn * f1[n] + cb * f1[b] + ct * f1[t];  
        }  
    }  
}
```

# vectorization



# vectorization

```
for (int z = 0; z < nz; z++) {  
  for (int y = 0; y < ny; y++) {  
    for (int x = 0; x < nx; x++) {  
      int c, w, e, n, s, b, t;  
      c = x + y * nx + z * nx * ny;  
      w = (x == 0) ? c : c - 1;  
      e = (x == nx-1) ? c : c + 1;  
      n = (y == 0) ? c : c - nx;  
      s = (y == ny-1) ? c : c + nx;  
      b = (z == 0) ? c : c - nx * ny;  
      t = (z == nz-1) ? c : c + nx * ny;  
      f2[c] = cc * f1[c] + cw * f1[w] + ce * f1[e]  
        + cs * f1[s] + cn * f1[n] + cb * f1[b] + ct * f1[t];  
    }  
  }  
}
```





# vectorization

Compilation:

```
icc -vec-report=3 -O3 -std=c99 diffusion_base.c
```

```
diffusion_base.c(110): (col.11) remark: vector dependence:  
assumed ANTI dependence between f1 line 110 and f2 line  
110
```

```
diffusion_base.c(110): (col.11) remark: vector dependence:  
assumed FLOW dependence between f2 line 110 and f1 line  
110
```



# vectorization

```
void diffusion_base(REAL *f1, REAL *f2, ...
```

- pointer aliasing prevents vectorization
- arrays f1 and f2 could overlap

```
void diffusion_base(REAL *restrict f1, REAL *restrict f2, ...
```

```
diffusion_base.c(101): (col.9) remark: LOOP was VECTORIZED
```

Will be back to this example with further optimizations.

For the moment: Twice as fast as original version.



Einführung in das Hochleistungsrechnen  
Introduction to High Performance Computing

**VIELEN DANK**  
**THANK YOU**