NTNU – Trondheim
Norwegian University of
Science and Technology

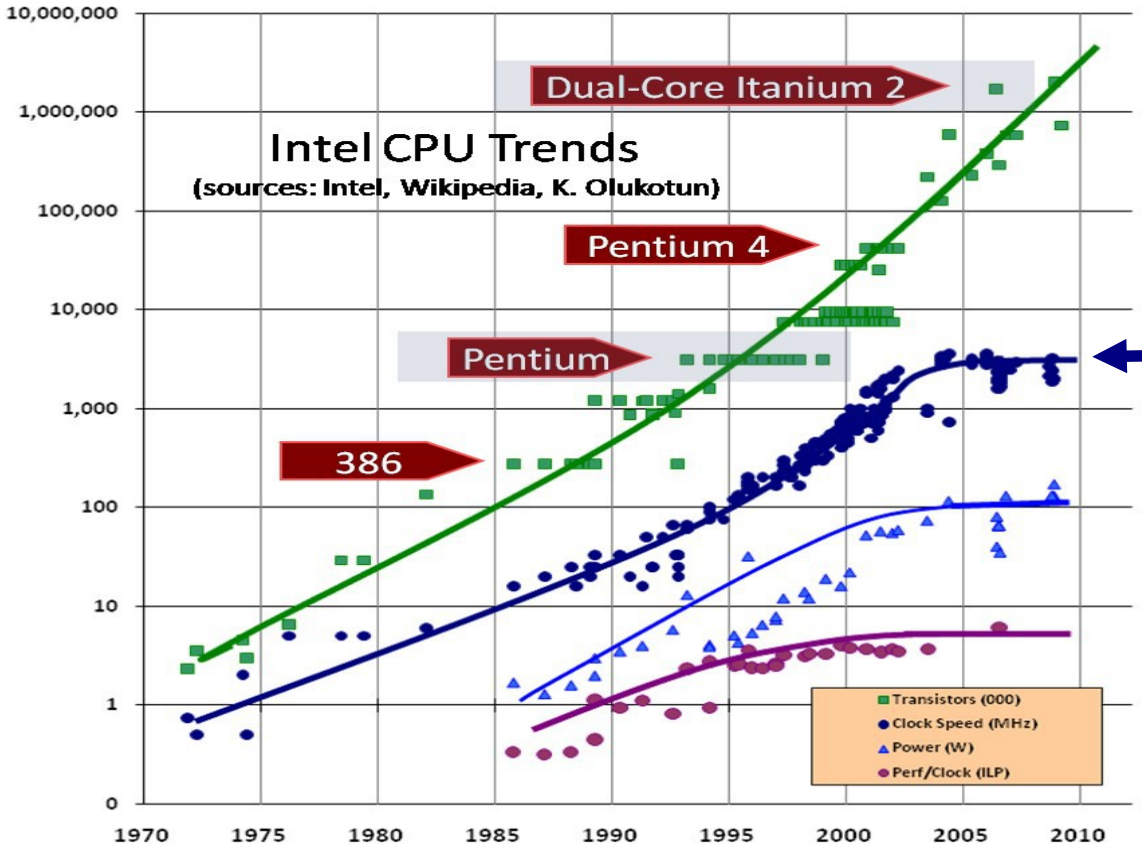
Agenda

1. CPU History
2. Compilator optimization
3. Cache
4. Libraries
5. Memory allocation

www.hpc.ntnu.no

CPU HISTORY

- Moore's law (1965): Number of transistores doubles every two years.
- The clock frequency has flat out since 2005.



Clock speed

CPU energy consumption

CPU Transistor effect: **P ~ C * F * U^2**

C: Capacitance , F: Frequency , U: Transistor voltage

Dennard scaling law (1974):

Transistors get smaller and power density stays constant

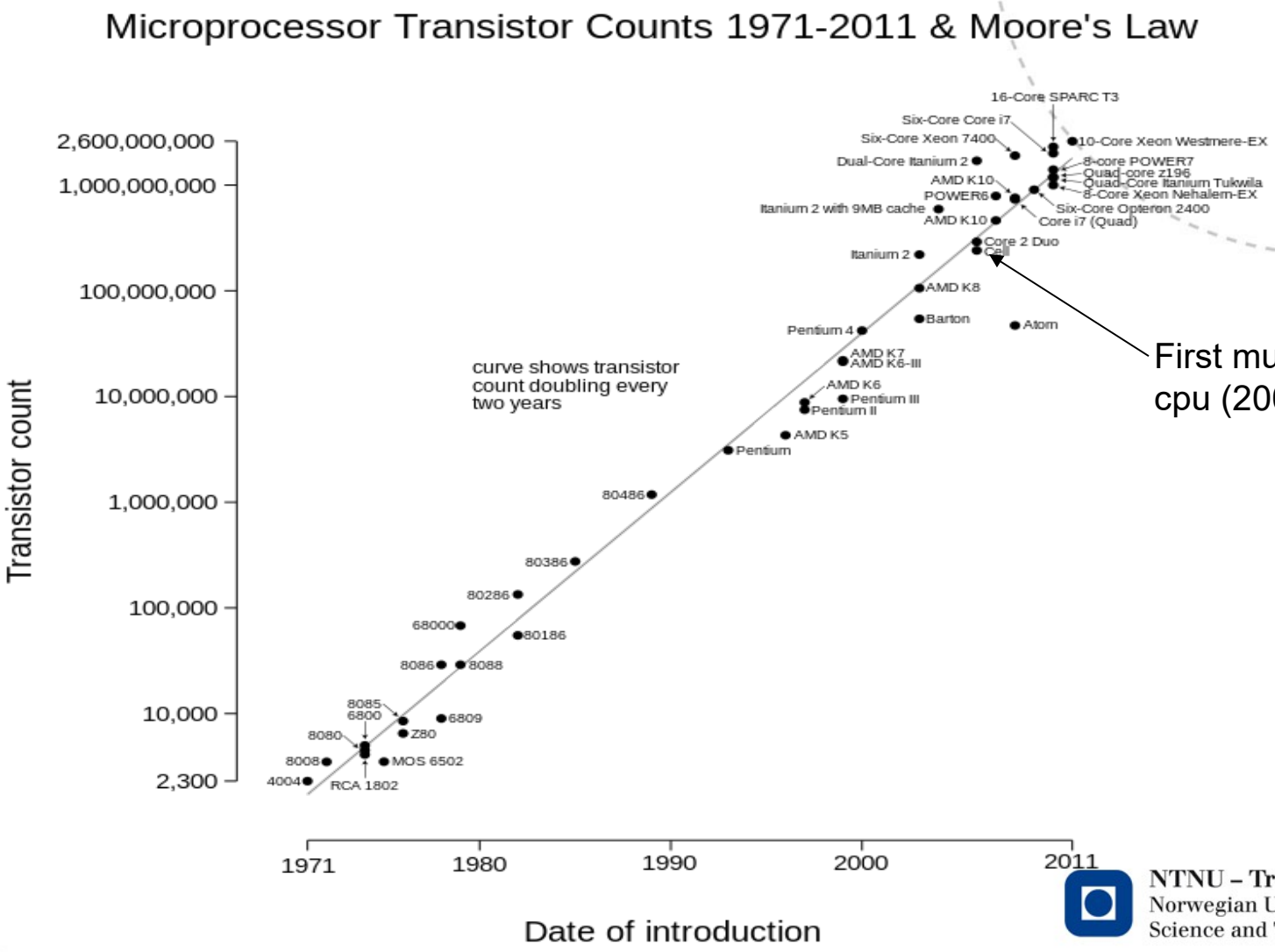
Dannard law	(new vs old cpu)	Today
Lnew = L / 2	(transistor length)	Lnew = L / 2
Dnew = 1 / L^2	= 4D (Density)	Dnew = 4D
Unew = U / 2	(Voltage)	Unew ~ U (same voltage)
Fnew = 2 * F	(Frequency)	Fnew = 2 * F
Pnew = P	(Power)	Pnew = 4P

(POWER CRISES IF THE FREQENCY INCREASE!!!)

- The tranisitor voltage can not be decreased much more.
- Transistor temperature and power can not increase.
- CPU frequency can not increase much more.
- Dennard scaling law do not hold anymore.

Moore's Law and number of cores

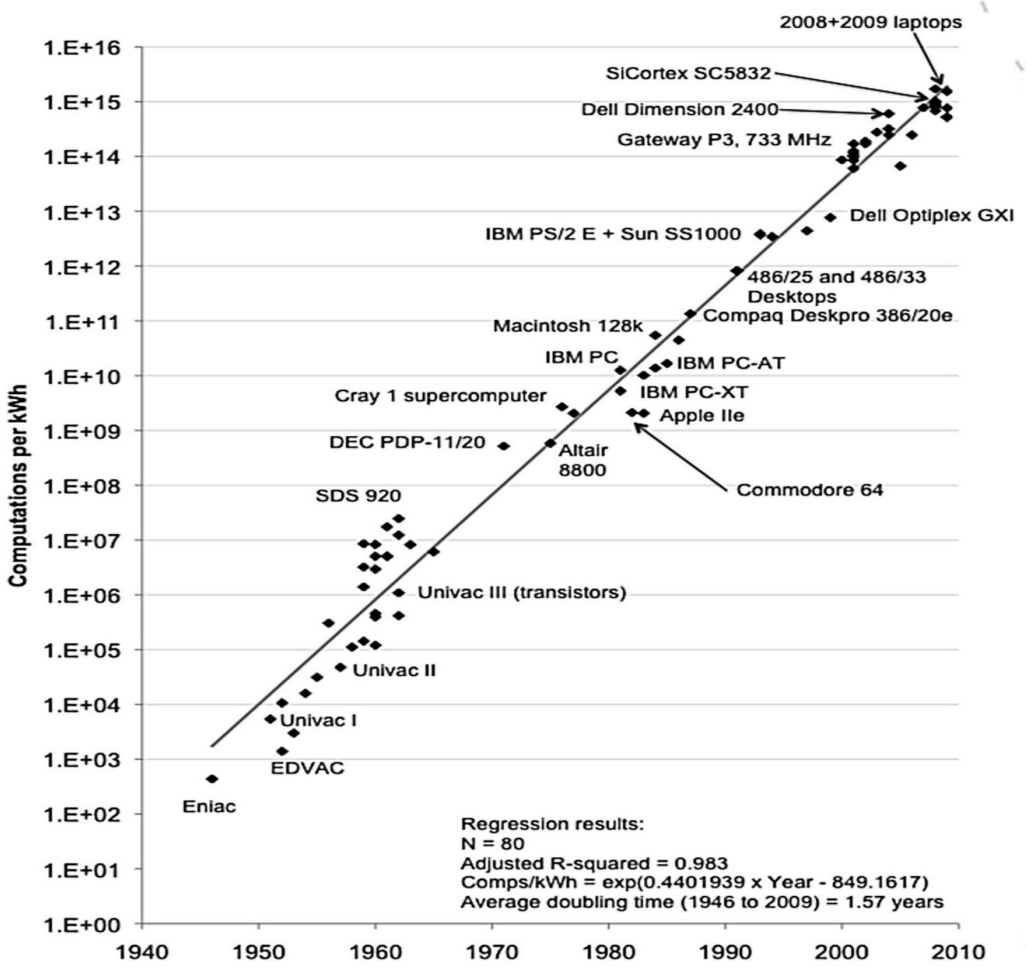
Number of cores double every 1.8 years. Moore's law still hold



NTNU – Trondheim
Norwegian University of Science and Technology

Koomey's performance law:

Computation per watt has been doubling every 1.57 years



NTNU – Trondheim
Norwegian University of Science and Technology

Vilje (SGI Altix 8600)

	Each Node	Total
Cores	16	23040
Nodes	-	1440
Memory	32GB	46TB
Storage	-	1000TB
Flops		479 Tflops (peak)

Lille (Dell)

	Each Node	Total
Cores	20	520
Nodes	-	27
Memory	128GB	3.4 TB

Compiler optimization

icc -O2 -o myprogram myprogram.c

See compiler options: man icc. Fortran **ifort**

The optimization code -O2 is default

(Note! For gnu compiler; use gcc -O2 ….)

Optimization code:

O0: Optimization is disabled

O1: Enables optimizations for speed and disables some optimizations that increase code size and affect speed.

O2: Enables optimizations for speed (default).This is the generally recommended optimization level. Vectorization is enabled at O2 and higher levels.

O3: Enables O2 optimizations plus more aggressive optimization.

AVX: Intel AVX vector extension (only for icc) (Sandy bridge and newer processors)

(See http://en.wikipedia.org/wiki/Advanced_Vector_Extensions)

icc -O3 -xAVX -o myprogram myprogram.c (gcc: -mavx or -mpku)

Parallel: Automatically parallelizing of the code, with -parallel as (not on gcc):

icc -O3 -parallel -o myprogram myprogram.c (gcc: ftree-parallelize-loops=n)

Fast: The -fast option maximizes speed across the entire program. Longer compiling time.

icc -O3 -fast -o myprogram myprogram.c

Results Lille (gcc compiler)

	-O0	-O1	-O2	-O3	-O3 -xAVX	-O3 -parallel
array1	14.8 s	5.1s	5.1s	4.55s		
pi	12.3s	4.8s	4.8s	4.8s		

Results Vilje (Intel compiler)

	-O0	-O1	-O2	-O3	-O3 -xAVX	-O3 -parallel
array1	28.1s	10.8s	3.6s	0.98s	0.49s	1.1s
pi	18.4s	8.5s	4.3s	4.2s	4.2s	0.3s

Pi:
```
for(i=0;i<nsteps;i++){
        x = (i + 0.5) * step;
        sum += 4.0/(1.0 + x * x);
    }
```
nsteps=10^9

Array1:
```
for (t=0;t<T;t++)
          for (i=0;i<n;i++)
            C[i]+=A[i]*B[i];
```

n=10^6, T=5000

CACHE

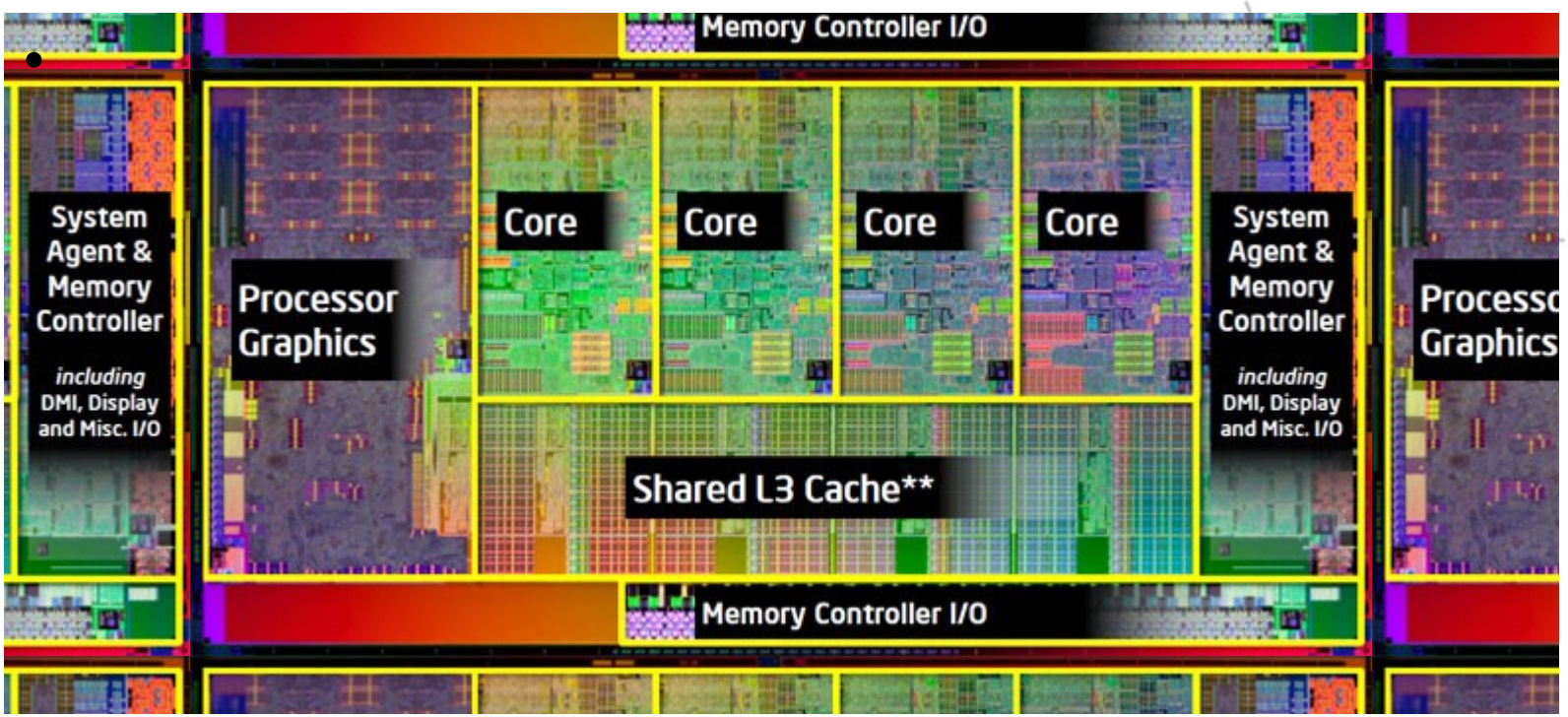


Fig. Laptop/pc CPU (Intel i7) with 4 cores

Multi-core processor

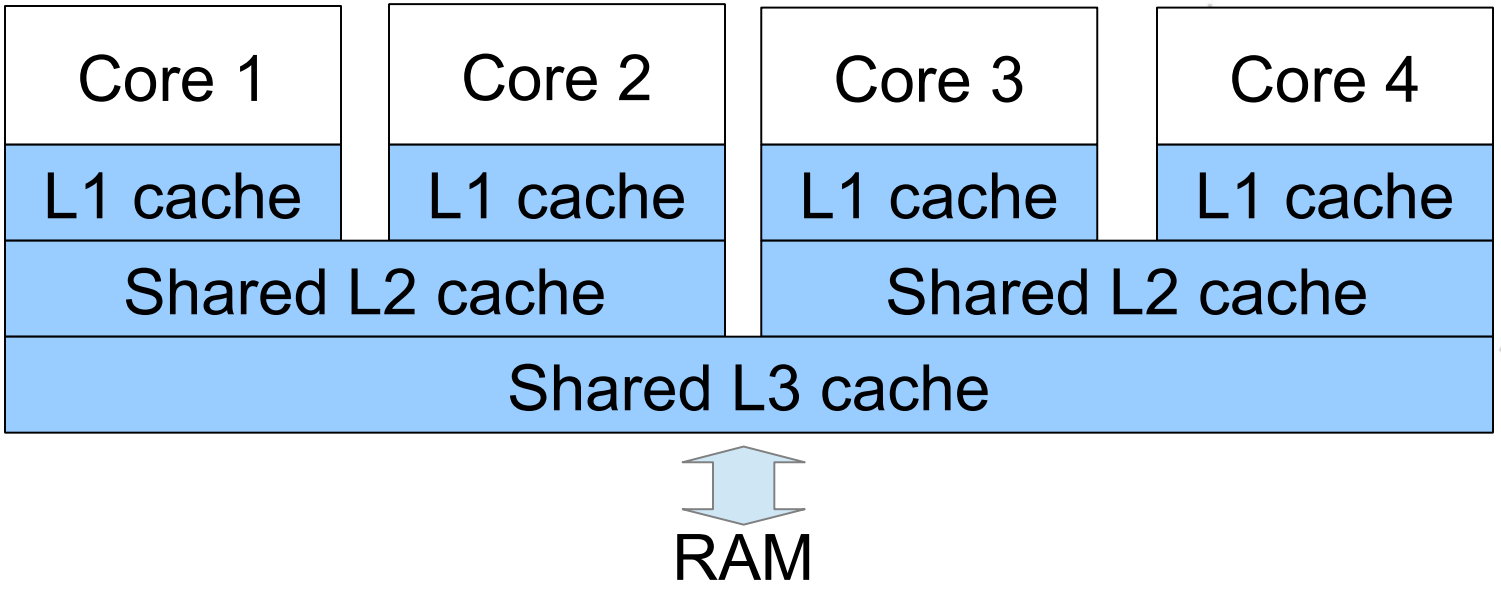


Figure 1. Intel i7 Sandy Bridge, 4 core processor with cache memory.
(L1: 64kB ~4cycles, L2: 256kB ~10cycles, L3: up to 20MB ~40cycles, RAM 32GB~120cycles)

- Each core has small memory cache L1 (level 1).
- Level 2 and 3 cache are shared between cores

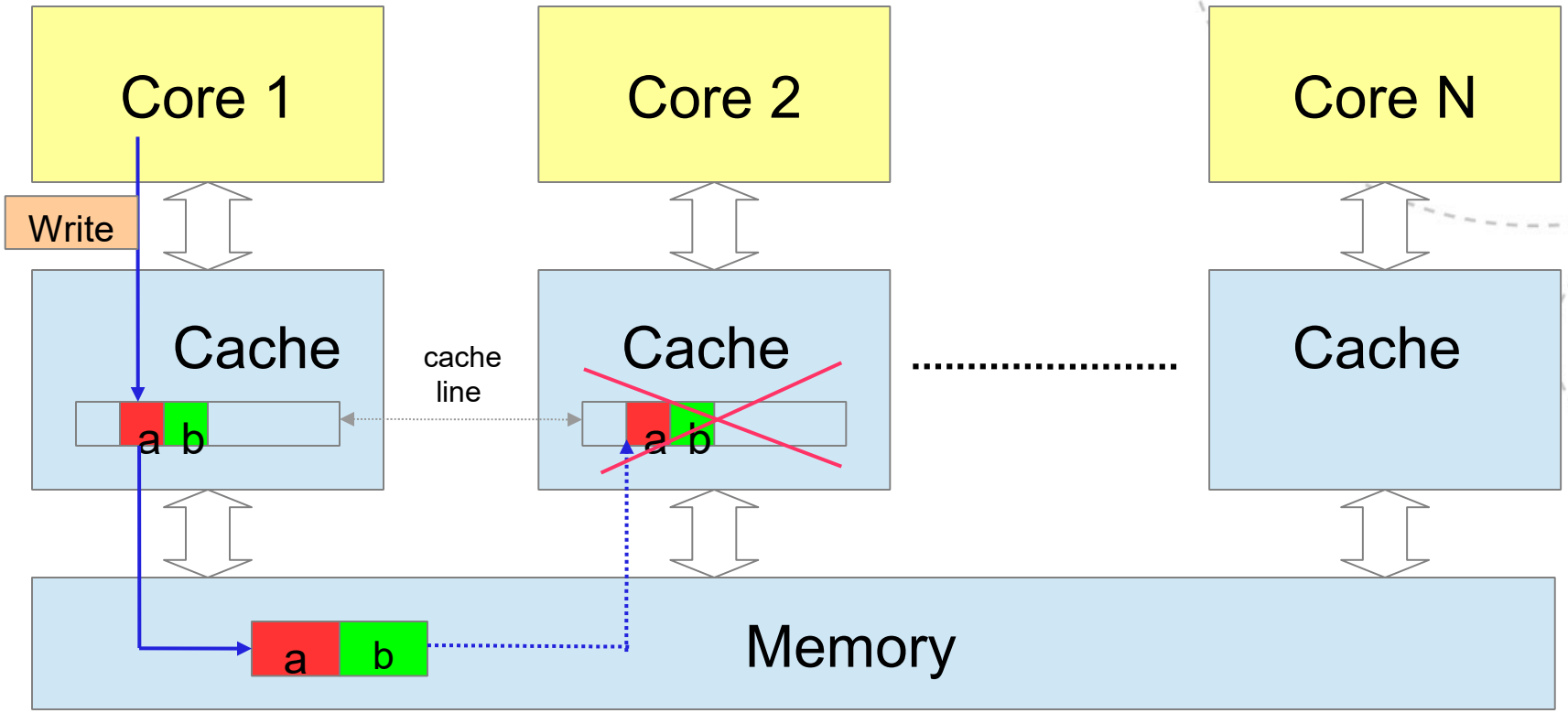
False Sharing

(See http://en.wikipedia.org/wiki/False_sharing)

The memory addresses are grouped into cache lines.

If one element in the cache line is changed of one core; the whole line will be automatically synchronized with the other cores cache. Drawback is that this will slow down because of data update latency.

False Sharing.



False Sharing: Example codes (for loop)

```
int i,j,N=5000;
float data[N][N];
```

Code A

```
for (i=0;i<N;i++)
   for (j=0;j<N;j++)
       data[i][j]=i*j;
```

```
...
```

Code B

```
for (i=0;i<N;i++)
   for (j=0;j<N;j++)
       data[j][i]=i*j;  //Give lot of false sharing
```

(Note! The indexing is opposite for Fortran)

Vilje: (j,i) 1.48s (i,j) 0.75s Idun: (j,i):3,1s (i,j) 0.36s

Cache miss

Cache miss is a state where the data requested for processing by a component or application is not found in the cache memory. It causes execution delays by requiring the program or application to fetch the data from other cache levels or the main memory.

Example that can cause cache miss

```
for (i=0;i<N;i++)
   A[i] += A[i-1]*A[i+1];
```

Example of avoiding cache misses

A, B and C arrays of type double and N elements.

```
for (i=0;i<N;i++)
   C[i] += A[i]*A2[i];
```

Divide forloop in to blocks of cache line size

If cache size = 64kB

int blocksize=64000/sizeof(double);

```
for (i=0;i<n;i+=blocksize)
   for (block=0;block<blocksize;block++)
      C[i+block] += A[i+block] * B[i+block];
```

LIBRARIES

Intel MKL Components (Lille: OpenBLAS)

Linear Algebra

BLAS • LAPACK/ ScaLAPACK • PARDISO* • Iterative sparse solvers

Fast Fourier Transforms

Multidimensional (up to 7D) FFTs • FFTW interfaces • Cluster FFT

Summary Statistics

Kurtosis • Variation coefficient • Quantiles, order statistics • Min/max • Variance/ covariance • ...

Data Fitting

Splines • Interpolation • Cell search

Other Components

Vector Math • Trigonometric • Hyperbolic • Exponential, Logarithmic • Power/Root Rounding • Vector Random Number Generators • Conguential • Recursive • WichmannHill • Mersenne Twister • Sobol • Neiderreiter • RDRANDbased • Poisson Solvers • Optimization Solvers

Example: Matrix multiplication with MKL library cblas_dgemm

Multiply Matrices Using dgemm

C <= αA*B + βC

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
        m, n, k, alpha, A, k, B, n, beta, C, n);
```

The arguments provide options for how Intel MKL performs the operation. In this case:

`CblasRowMajor` :Indicates that the matrices are stored in row major order, with the elements of each row of the matrix stored contiguously as shown in the figure above.

`CblasNoTrans` :Enumeration type indicating that the matrices A and B should not be transposed or conjugate transposed before multiplication.

A: `m` rows by `k` columns

B: `k` rows by `n` columns

C: `m` rows by `n` columns

`alpha` :Real value used to scale the product of matrices A and B.

`beta`: Real value used to scale matrix C.

(Example: https://software.intel.com/en-us/node/429920)

Matrix multiplication, standard algorithme

```
for (i=0;i<n;i++)
   for (j=0;j<m;j++)
   {
     tmp=0.0;
     for (k=0;k<p;k++)
        tmp += A[i][k] * B[j][k];  //B is transposed
     C[i][j]=tmp;
   }
```

Matrix multiplication using MKL/BLAS library:

```
cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,
            m, n, k, alpha, A, k, B, n, beta, C, n);
```

Results

Lille (gcc compiler -O3): Matmult: 128s BLAS: 0.9s

Vilje (icc compiler -O3): Matmult: 78.8 s MKL: 2.48s

Memory allocation.

Auto allocation of a memory.

Example:

```
{ // Inside a scop
  int a[10][10];
}
```

- The array is not reachable outside a scop and will be deleted when instruction pointer exit the scop. Scop is the program between {...}.
- The array will be stored into the stack. The stack has fast accessibility.
- The stack is limited. If you allocate more memory than available can result in a program crash due to stack overflow.
- For large memory allocation; use dynamic memory allocation.

Dynamic memory allocation.

```
{
  int n=1000; // Size of the array;
  double *array;  // Init array
  array = (double *) malloc ( sizeof (double) * n );
  ....
  myfunction (n,array);
  ....
  free (array);
}
```

- Dynamic memory allocated arrays will be stored in the heap.
- You can allocate all memory space.
- Heap has slower accessibility than the stack.
- The array will not be deleted when the instruction pointer go outside the scop. You have to use free(array);