

# Linear algebra on multi-core and heterogeneous architectures

## A single core

Web page:

<http://www.math.ethz.ch/~kressner/gpucomp.php>

Daniel Kressner

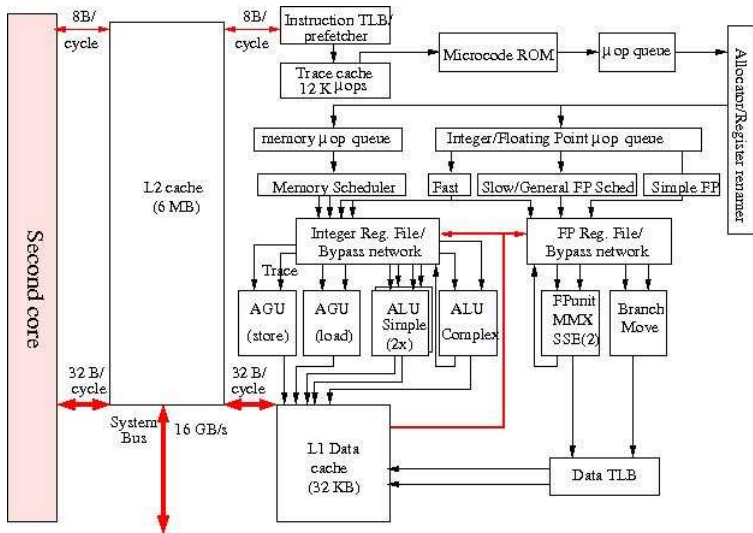
## Motivation

- ▶ No application runs at “peak” performance.
- ▶ Much of performance loss on single core level, e.g., code runs on one core at only 10 – 20% of core peak performance.
- ▶ Most of single core performance loss due to memory transfers.
- ▶ These issues will persist on multi-cores (and get accentuated).

## Outline

- ▶ Parallelism within single cores.
- ▶ Memory hierarchies.
- ▶ Matrix-vector and matrix-matrix multiplies.
- ▶ Case study: QR factorization.

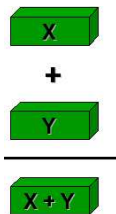
# Inside a modern core (Intel Xeon)



# SIMD: Single Instruction stream – Multiple Data

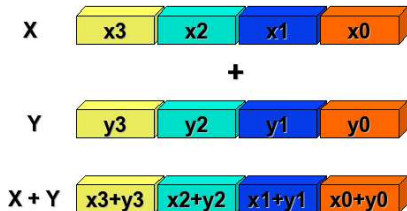
## Scalar processing

One operation produces one result.



## SIMD processing

One operation produces multiple results.

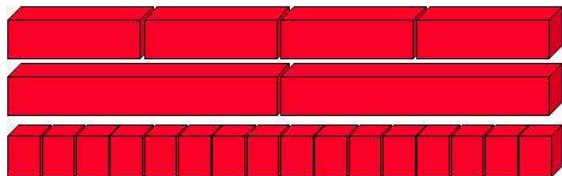


Instruction set supported by most processors: [SSE2](#) (Streaming SIMD Extensions 2).

Picture Source: Alex Klimovitski & Dean Macri, Intel Corporation

# SSE2 SIMD on Intel

SSE2 data types = 128bit registers



4x floats

2x doubles

16x bytes

- ▶ Instructions perform add, multiply etc. on 128bit registers in parallel.
- ▶ Many modern cores (AMD Phenom, Intel Core 2 Quad, Intel Core i7) have two SSE units.
- ▶ Data needs to be contiguous in memory and aligned.
- ▶ In theory, compiler turns your code into SIMDs.
- ▶ In practice, compiler might need your help:
  - ▶ Choose suitable compiler, optimization flags, etc.
  - ▶ Rearrange your code to make things more obvious.
  - ▶ **Worst case** (only in critical kernel routines): Use intrinsics or write in assembler.

# A simple loop, $k = 10^9$ , gcc -O3

0.46s

```
for (j = 0; j < k; j++) {  
    out[j] = inp1[j] + inp2[j];  
}
```

0.47s

```
for (j = 0; j < k; j+=2) {  
    out[j] = inp1[j] + inp2[j];  
    out[j+1] = inp1[j+1] + inp2[j+1];  
}
```

0.59s

```
for (j = 0; j < k; j++) {  
    if ( inp1[j] == 0 ) {  
        out[j] = inp2[j];  
    } else if ( inp2[j] == 0 ) {  
        out[j] = inp1[j];  
    } else {  
        out[j] = inp1[j] + inp2[j];  
    }  
}
```

0.33s

```
for (j = 0; j < k; j+=1000) {  
    out[j] = inp1[j] + inp2[j];  
}
```

# Making things more obvious to the compiler

**Toy problem:** Given a vector  $b$  of length  $n$ , overwrite  $b$  as follows

$$\begin{bmatrix} b_j \\ b_{j+1} \end{bmatrix} \leftarrow \begin{bmatrix} c_j & s_j \\ -s_j & c_j \end{bmatrix} \begin{bmatrix} b_j \\ b_{j+1} \end{bmatrix}, \quad j = 1, \dots, n-1,$$

where  $c_j, s_j$  are cosines/sines of Givens rotations.

**Two options:**

- (a) Perform operation by calling specialized (potentially optimized) subroutine for applying rotations.
- (b) Write out  $2 \times 2$  matrix-matrix-multiplication explicitly.

Which one is faster?

# Making things more obvious to the compiler

1.42s

```
DO 10 J = 1, N-1
    CALL SROT( 1, B(J), 1, B(J+1), 1,
$           CS(J), SN(J) )
10 CONTINUE
```

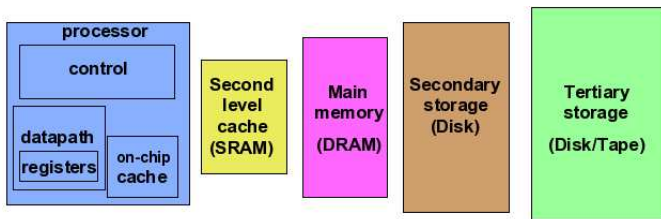
0.59s

```
DO 20 J = 1, N-1
    W      = B(J)
    Z      = B(J+1)
    B(J)   = CS(J)*W+SN(J)*Z
    B(J+1) = -CS(J)*W+SN(J)*Z
20 CONTINUE
```

This attains an impressive 10.2 Gflops (peak = 17.6 SP-Gflops).

Explicitly unrolling code for tiny kernel tasks pays off.

# Memory hierarchy



Speed	1ns	10ns	100ns	10ms	10sec
Size	KB	MB	GB	TB	TB

- ▶ Most programs have high degree of **locality** in their accesses.
  - ▶ **spatial locality**: accessing things nearby previous accesses
  - ▶ **temporal locality**: reusing an item that was previously accessed
- ▶ Memory hierarchy tries to exploit locality.

# CAS latency

Memory bandwidth has improved more significantly than latency.

(CAS latency is the delay between the moment the memory is requested and the moment the data is available.)

Two Examples:

- ▶ **PC100 SDRAM** has a data rate of 100 million transfers/s, and a command rate of 100 MHz. CAS latency is 2 cycles, so 1st word after **20ns**, and 8th word after **90ns**.
- ▶ **DDR3-1600 SDRAM** has a data rate 1600 million transfers/s, and a command rate of 800 MHz. CAS latency is 9 cycles, so 1st word after **13.1ns**, and 8th word after **15.6ns**.

Lessons:

- ▶ Eliminate memory operations by saving values in small, fast memory (cache) and reusing them (temporal locality).
- ▶ Take advantage of better bandwidth by getting a chunk of memory and saving it in small fast memory (cache) and using whole chunk (spatial locality).

# Data layout is important

Column major: Fortran, MATLAB, OpenGL.

Row major: C-arrays, bitmaps, Python.

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Column major:

A_arr	1	4	7	2	5	8	3	6	9
-------	---	---	---	---	---	---	---	---	---

Row major:

A_arr	1	2	3	4	5	6	7	8	9
-------	---	---	---	---	---	---	---	---	---

0.2 sec ( $k = 7000$ ):

```
for (i = 0; i < k; i++) { for (j = 0; j < k; j++) {  
    out[i][j] = inp1[i][j] + inp2[i][j];  
} }
```

1.01 sec ( $k = 7000$ ):

```
for (i = 0; i < k; i++) { for (j = 0; j < k; j++) {  
    out[j][i] = inp1[j][i] + inp2[j][i];  
} }
```

# Matrix-Vector-Multiplication, $k = 10\,000$

$A$  is  $k \times k$  matrix containing single precision floating point numbers

$$c = Ab \Leftrightarrow c = \sum_{j=1}^k a_{:,j} b_j \Leftrightarrow c_i = \sum_{j=1}^k a_{ij} b_j$$

Which one is faster?

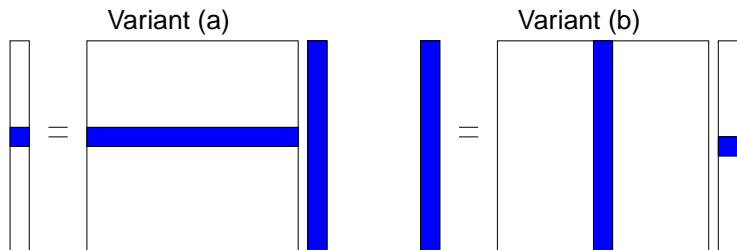
Variant (a)

```
for (i = 0; i < k; i++) { for (j = 0; j < k; j++) {  
    c[i] = c[i] + mat[i][j] * b[j];  
} }
```

Variant (b)

```
for (j = 0; j < k; j++) { for (i = 0; i < k; i++) {  
    c[i] = c[i] + mat[i][j] * b[j];  
} }
```

# Matrix-Vector-Multiplication



	C	Fortran
Variant (a)	0.15s	1.09s
Variant (b)	1.10s	0.15s

Goto BLAS SGEMV takes 0.14 seconds.

Test run on single core of 2.2 GHz Intel Core Duo  
(peak = 17.6 SP-Gflops, attained = 1.4 SP-Gflops).

# Matrix-Matrix-Multiplication

$A, B$  is  $n \times n$  matrices containing single precision FP numbers

$$C = AB \quad \Leftrightarrow \quad c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

Fortran 77 code:

```
      DO 13 J = 1, N
        DO 12 I = 1, N
          DO 11 K = 1, N
            C(I,J) = C(I,J) + A(I,K)*B(K,J)
          11      CONTINUE
        12      CONTINUE
      13      CONTINUE
```

Performance for  $n = 2000$ :

	Fortran
Vanilla code	56s
Goto BLAS SGEMM	0.94s

Test run on single core of 2.2 GHz Intel Core Duo  
(peak = 17.6 SP-Gflops, attained = 17.0 SP-Gflops).

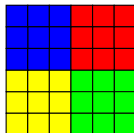
# BLAS (Basic Linear Algebra Subroutines)

- ▶ Industry standard interface
- ▶ Vanilla implementations: [www.netlib.org/blas](http://www.netlib.org/blas)
- ▶ Vendors, others supply optimized implementations
- ▶ Each routine comes in four variants: SGEMM (single precision real), DGEMM (double precision real), CGEMM (single precision complex), ZGEMM (double precision complex).
- ▶ **BLAS level 1**
  - ▶ vector operations: dot product, xAXPY ( $y = \alpha x + y$ ), etc
  - ▶ #flops / #memory accesses  $\approx 1$
- ▶ **BLAS level 2**
  - ▶ matrix-vector operations: xGEMV ( $y = \alpha Ax + \beta y$ ), etc
  - ▶ #flops / #memory accesses  $\approx 1$
- ▶ **BLAS level 3**
  - ▶ matrix-matrix operations: xGEMM ( $C = \alpha AB + \beta C$ ), etc
  - ▶ #flops / #memory accesses  $\approx n$ , potentially much faster than BLAS level 2
  - ▶ Good linear algebra algorithms use BLAS3 whenever possible!

# BLAS (Basic Linear Algebra Subroutines)

Popular non-vendor implementation:

- ▶ ATLAS (Automatically Tuned Linear Algebra Software)
  - ▶ Copies matrices into block major storage such that blocks fit into  $L1$  cache.



Conventional storage (Fortran)



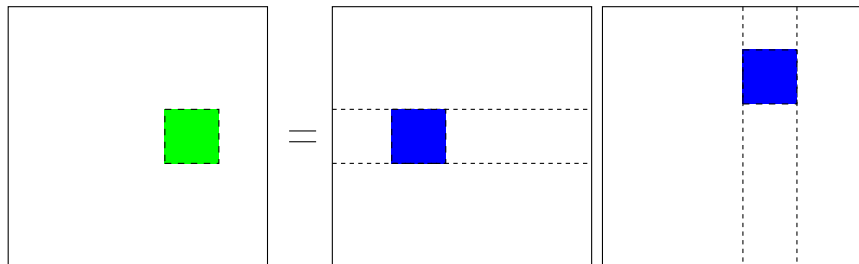
Block Major Storage



# BLAS (Basic Linear Algebra Subroutines)

Popular non-vendor implementation:

- ▶ ATLAS (Automatically Tuned Linear Algebra Software)
  - ▶ Copies matrices into block major storage such that blocks fit into L1 cache.
  - ▶ Block-wise multiplication:  $C_{ij} \leftarrow \sum_{k=1}^{N/NB} A_{ik} B_{kj}$ .
  - ▶ Blocks are contiguous and fit into L1 cache.
  - ▶ L1 kernels for multiplying blocks: unrolled code+hand tuning+compiler optimization.



# BLAS (Basic Linear Algebra Subroutines)

Another popular non-vendor implementation:

- ▶ Goto BLAS

- ▶ See <http://www.tacc.utexas.edu/resources/software/>
- ▶ Layered approach to deal with issues related to Translation Look-aside Buffer (TLB).
- ▶ In some cases significantly faster than ATLAS.

Vendor-supplied BLAS

- ▶ MKL (Intel), ACML (AMD), ...
- ▶ On novel HPC architectures (Cell, GPUs, ...), vendor-supplied BLAS often the only choice, but not necessarily a good choice (optimization/testing typically focuses on xGEMM and somewhat neglects other BLAS).

# Case Study: QR factorization

Given  $n \times n$  matrix  $A$ , compute orthogonal  $Q$  and upper triangular  $R$   
s.t.

$$A = QR.$$

Basic outline of algorithm:

$$\begin{bmatrix} \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \\ \times & \times & \times & \times \end{bmatrix} \xrightarrow{Q_1} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & \times & \times & \times \end{bmatrix} \xrightarrow{Q_2} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & \times & \times \end{bmatrix} \xrightarrow{Q_3} \begin{bmatrix} \times & \times & \times & \times \\ 0 & \times & \times & \times \\ 0 & 0 & \times & \times \\ 0 & 0 & 0 & \times \end{bmatrix}$$

$A$                        $Q_1 A$                        $Q_2 Q_1 A$                        $Q_3 Q_2 Q_1 A$

With  $R := Q_3 Q_2 Q_1 A$  and  $Q := (Q_3 Q_2 Q_1)^T \rightsquigarrow A = QR$ .

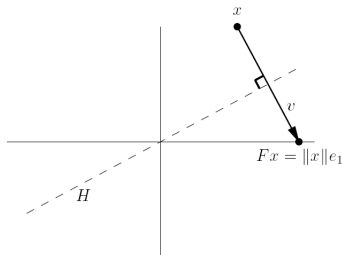
# Householder reflectors

Fundamental op in QR: map  $x$  to scalar multiple of  $e_1 = [1, 0, \dots, 0]^T$ .

## Variant 1:

- ▶ Reflect  $x$  along hyperplane  $H$  orthogonal to  $v = \|x\|_2 e_1 - x$ .
- ▶  $Fx = \|x\|_2 e_1$  with

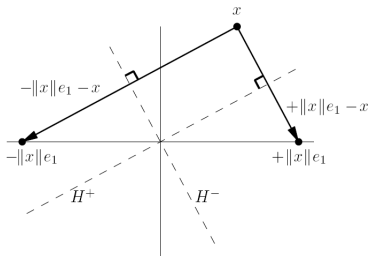
$$F = I - \frac{2}{v^T v} v v^T.$$



## Variant 2:

- ▶ Reflect  $x$  along hyperplane  $H^-$  orthogonal to  $v = -\|x\|_2 e_1 - x$ .
- ▶  $Fx = -\|x\|_2 e_1$  with

$$F = I - \frac{2}{v^T v} v v^T.$$



**Numerically safe choice:**  $v = -\text{sign}(x_1)\|x\|_2 e_1 - x$

# QR factorization based on Householder reflectors

## MATLAB script for computing R

```
for k = 1:n-1,
    x = A(k:n,k); v = x;
    if x(1)>=0,
        v(1) = v(1) + norm(x);
    else
        v(1) = v(1) - norm(x);
    end
    v = v / v(1); tau = 2 / norm(v)^2;
    w = tau * A(k:n,k:n)' * v;      % level 2 BLAS (SGEMV)
    A(k:n,k:n) = A(k:n,k:n) - v * w'; % level 2 BLAS (SGER)
end
```

- ▶ Upper triangular part of  $A$  overwritten by  $R$ .
- ▶ Lower triangular part can be used to store  $v$  (Add  $A(k+1:n,k) = v(2:end)$ .)
- ▶ Requires

$$\sum_{k=1}^{n-1} \underbrace{2(n-k)^2}_{\text{SGEMV}} + \underbrace{2(n-k)^2}_{\text{SGER}} = \frac{4}{3}n^3 + O(n^2) \text{ flops.}$$

# QR factorization based on Householder reflectors

MATLAB script for computing Q.

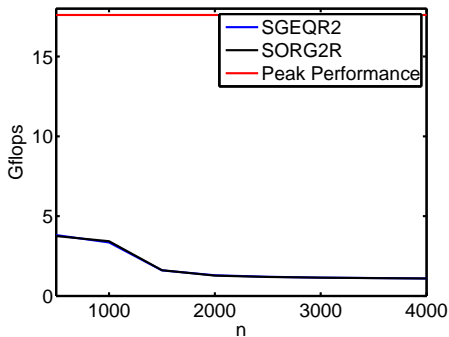
(Assuming the vectors  $v$  are initially stored in lower triangle of Q and  $\tau$  in vector `tau` ).

```
for k = n-1:-1:1,
    v = Q(k:n,k); Q(k+1:n,k) = 0;
    w = tau(k) * Q(k:n,k:n)' * v;      % level 2 BLAS (SGEMV)
    Q(k:n,k:n) = Q(k:n,k:n) - v * w'; % level 2 BLAS (SGER)
end
```

## ► Requires

$$\sum_{k=1}^{n-1} \underbrace{2(n-k)^2}_{\text{SGEMV}} + \underbrace{2(n-k)^2}_{\text{SGER}} = \frac{4}{3}n^3 + O(n^2) \text{ flops.}$$

## Observed performance of QR



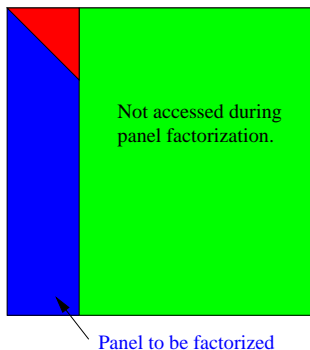
- ▶ Example: For  $n = 2000$ , SGEQR2 requires 8.2s to perform QR factorization, SORG2R requires 8.4s to generate  $Q$ . (Comparison: The “more expensive” SGEMM requires 0.94s.)
- ▶ Reason: Everything performed in level 2 BLAS, flops/memory access ratio  $\approx 1$ .

# Bringing level 3 BLAS into QR

Two basic ideas:

1. Delay updates of most of  $A$ .
2. Accumulate Householder reflectors.

QR factorization of panel:



Think of a  $2000 \times 2000$  matrix and a panel that contains  $nb = 64$  columns. **Green part** is updated *after* panel factorization.

# Bringing level 3 BLAS into QR

Accumulation of Householder reflectors:

$$Q_1 = I - \tau_1 v_1 v_1^T, \quad Q_2 = I - \tau_2 v_2 v_2^T.$$

$$\begin{aligned} Q_1 Q_2 &= (I - \tau_1 v_1 v_1^T)(I - \tau_2 v_2 v_2^T) \\ &= I - \tau_1 v_1 v_1^T - \tau_2 v_2 v_2^T + \underbrace{\tau_1 \tau_2 v_1^T v_2}_{=:-t_{12}} v_1 v_2^T \\ &= I - [v_1, v_2] \begin{bmatrix} \tau_1 & t_{12} \\ 0 & \tau_2 \end{bmatrix} [v_1, v_2]^T \\ &=: I - VTV^T. \end{aligned}$$

For general  $nb$ :

$$Q_1 Q_2 \cdots Q_{nb} = I - VTV^T, \quad V = [v_1, \dots, v_{nb}], \quad T = \begin{array}{|c} \triangle \\ \hline \end{array}.$$

So called **compact WY representation**.

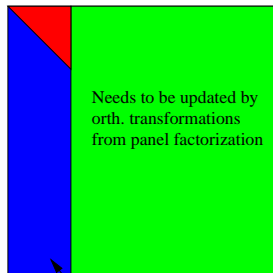
# Bringing level 3 BLAS into QR

Matlab script for computing  $T$ :

```
function T = compactWY(V,tau)
[m,nb] = size(V); T = zeros(nb);
for j = 1:nb,
    T(1:j-1,j) = -tau(j) * T(1:j-1,1:j-1) * ( V(:,1:j-1)'*V(:,j) );
    T(j,j) = tau(j);
end
```

Once compact WY representation is available, update of  $A$  performed entirely with level 3 BLAS:

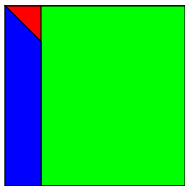
```
% SGEMM
Y = A(:,nb+1:n)' * W;
% STRMM
Y = Y * T';
% SGEMM
A(:,nb+1:n) = A(:,nb+1:n) - W * Y';
```



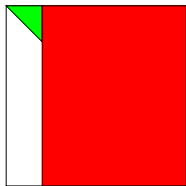
Needs to be updated by  
orth. transformations  
from panel factorization

Panel already factorized

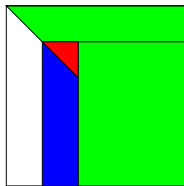
Panel reduction (BLAS 2)



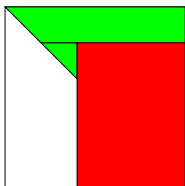
Update (BLAS 3)



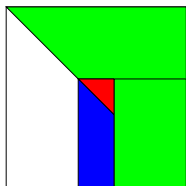
Panel reduction (BLAS 2)



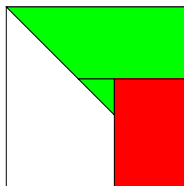
Update (BLAS 3)



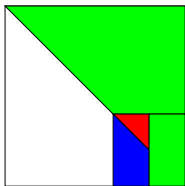
Panel reduction (BLAS 2)



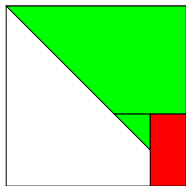
Update (BLAS 3)



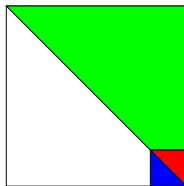
Panel reduction (BLAS 2)



Update (BLAS 3)

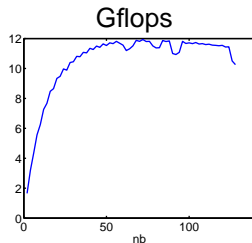
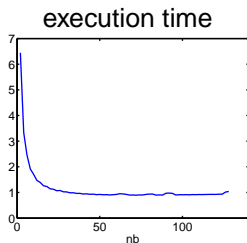


Panel reduction (BLAS 2)

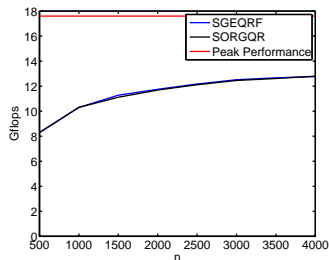


# Bringing level 3 BLAS into QR – Experiments

2000 × 2000 matrix, sensitivity of performance w.r.t.  $nb$ :



$nb = 64$ , performance w.r.t.  $n$



# Summary

- ▶ Data locality (temporal, spatial) important to attain high performance on single core.
- ▶ Unroll codes in kernels.
- ▶ Use as much as possible highly optimized level 3 BLAS  $\rightsquigarrow$  portable performance.
- ▶ Sometimes nontrivial to reformulate algorithm in terms of level 3 BLAS.
- ▶ Further reading:
  - S. Goedecker, A. Hoisie: Performance Optimization of Numerically Intensive Codes, SIAM 2001.
  - J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst: Numerical Linear Algebra on High-Performance Computers (Software, Environments, Tools). SIAM 1999.