# Eigensolvers

# Eigenvalue problems

- The standard eigenvalue problem $\mathbf{Ax}=\lambda\mathbf{x}$
  - **A** is a square *n*x*n* matrix
  - Any vector **x** that satisfies the equation is an eigenvector of **A** (usually the eigenvectors are normalized after determination)
  - $\lambda$ is an eigenvalue corresponding to **x**
- **A** has *n* linearly independent eigenvectors
- Matrix **X** whose columns are the eigenvectors will *diagonalize* **A** in a similarity transformation

  $$\mathbf{A}'=\mathbf{X}^{-1}\mathbf{AX}$$

  - **A**' will be a diagonal matrix with the eigenvalues of **A** in its diagonal

# Basic power method

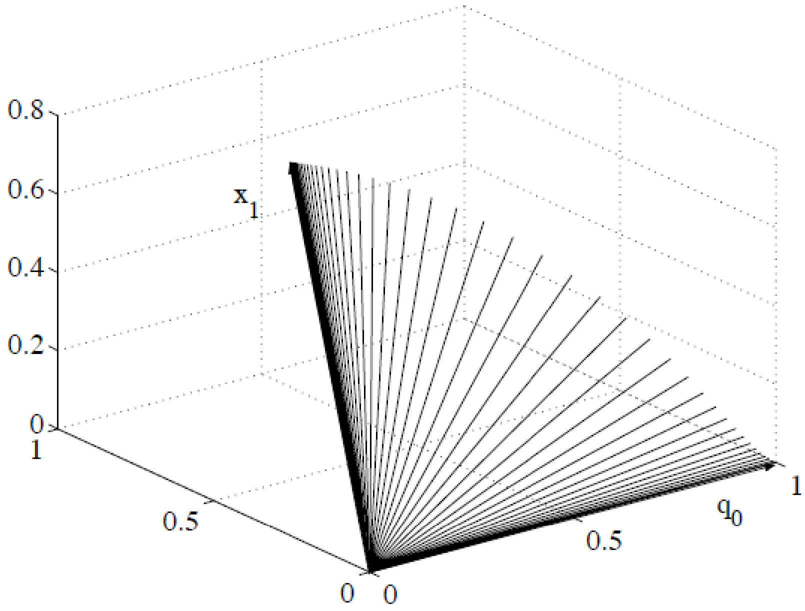
- The power method is a simple method for finding the maximum eigenvalue
- The main idea is to obtain successive iterates from $\mathbf{x}^{k+1} = c\mathbf{A}\mathbf{x}^{k}$
  - c is a normalization constant
  - When k becomes large, $\mathbf{x}^{k+1} \rightarrow \mathbf{v}$, where $\mathbf{v}$ is the eigenvector corresponding to the maximum eigenvalue

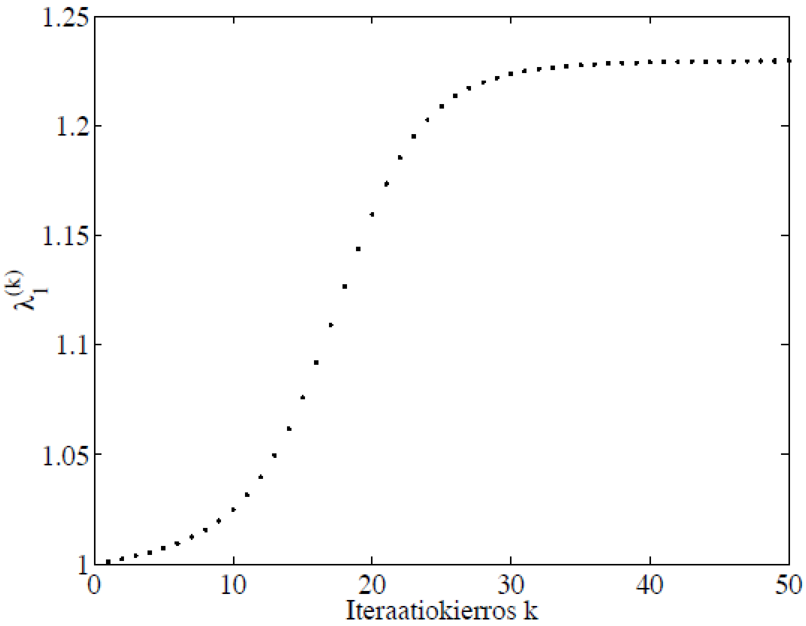
```
(initialize x)
do k = 1, n
    x = matmul(A,x)
    x = x/maxval(x)
end do
```

Also other norms can be used for the normalization, e.g. the L2 norm

  - The largest eigenvalue can be obtained as the maximum value (or an inner product $\mathbf{v}^{T}\mathbf{A}\mathbf{v}$ if L2 norm was used in the iteration) from the converged iterate

# Power method



# Inverse power method

- To selectively compute the minimum eigenvalue an inverse power method can be applied

```
x = 1
do k = 1, n
   y = solve(A,x)
   x = y/norm(y)
end do
```

Solution of $\mathbf{A}\mathbf{x}=\mathbf{y}$

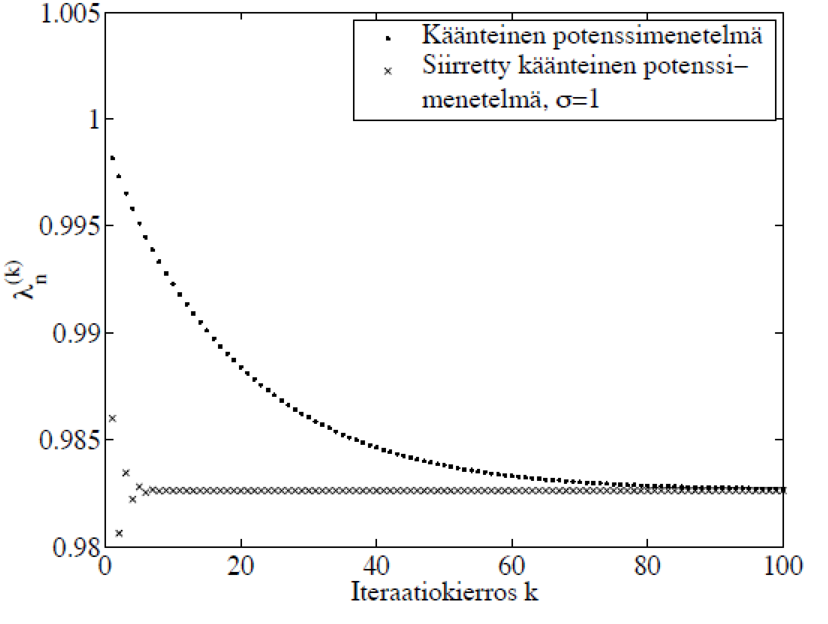
  - It is beneficial to form e.g. the LU decomposition of A before the iteration for a faster solution of the linear equations
- The smallest eigenvalue is obtained from the converged iterate as $1/\lambda \approx \mathbf{x}_{m-1} \bullet \mathbf{y}_m$
- Better convergence is obtained with a proper shift, i.e. to modify the iteration to read $(\mathbf{A}-\sigma\mathbf{I})\mathbf{x}^{k+1}=c\mathbf{x}^k$

# Inverse shifted power method

- Better convergence for the inverse power method is obtained with a proper shift, i.e. to modify the iteration to read

  $(\mathbf{A}-\sigma\mathbf{I})\mathbf{x}^{k+1}=c\mathbf{x}^{k}$

- The thus found eigenvalues are also shifted, i.e. iterations converge to the eigenvector corresponding to $|\lambda-\sigma|$



# Deflation

- Assume we have determined the smallest/largest eigenvalue with a power method
- Before we can reapply these and compute another eigenvalue, we need to “deflate” the matrix
  - This means forming a matrix from which the “impact” of the largest/smallest eigenvalue has been neglected
  - For example a matrix $\tilde{\boldsymbol{A}} = \boldsymbol{A} - \lambda_1 \boldsymbol{v_1} \boldsymbol{v_1^T}$ has the eigenvalues 0, $\lambda_2$, $\lambda_3$,...
- It should be noted that the deflation methods are efficient for finding a couple of eigenvalues, for a larger number global eigensolvers should be used

# Householder deflation

- An always stable deflation method is the Householder deflation, a process consisting of the following steps
  - Compute the maximum eigenvalue and the corresponding eigenvector
  - Obtain a Householder matrix **H** using the eigenvector **w** as

$$\boldsymbol{H} = \boldsymbol{I} - 2\frac{\boldsymbol{w}\boldsymbol{w}^T}{\boldsymbol{w}^T\boldsymbol{w}} \; \forall \, \boldsymbol{w} \neq \boldsymbol{0}$$

  - Form a matrix $\mathbf{G} = \mathbf{H}\mathbf{A}\mathbf{H}^{T}$
  - Extract the eigenvalue at $G_{11}$
  - Define $\mathbf{A}_{n-1} = \mathbf{G}_{2:n,2:n}$
  - Repeat the process on the newly defined matrix $\mathbf{A}_{n-1}$

# The QR eigensolver

- Global semidirect method
- Construct a sequence of similar matrices $\{\mathbf{A}_k\}$, $k=1,\ldots,n$, where $\mathbf{A}_1=\mathbf{A}$ and $\mathbf{A}_n$ approaches, as $k$ gets large, an upper triangular matrix with the eigenvalues on the diagonal

```
T = A
do k = 1, n
   call QR_decomposition(T,Q,R)
   T = matmul(R,Q)
end do
```

Form a QR decomposition $\mathbf{T}=\mathbf{QR}$

- A computationally more feasible method is obtained by starting by transforming A into the tridiagonal form or into a Hessenberg form

# More global eigensolvers

- Orthogonal iteration
- Krylov subspace methods
  - Lanczos method
  - Arnoldi method
- Divide-and-conquer

# Literature recommendations

- J.W. Demmel, *Applied Numerical Linear Algebra* (SIAM 1997)
- Y. Saad, *Numerical Methods for Large Eigenvalue Problems* (PWS Publishers 1996)

# Application performance

# General considerations

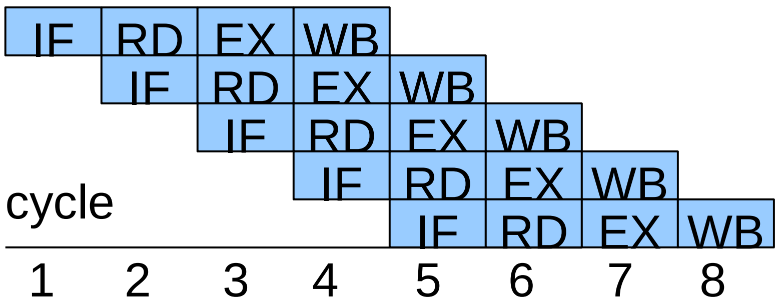
- Select the algorithm carefully
- Choose the right machine (type) for your code
- Be aware of the scaling properties of your code and the problem you are looking at
- Use the machine properly

# Understanding parallel scalability

- Parallel overhead
  - additional operations which are not present in serial calculation
  - synchronization
  - redundant computation
  - communication
- Load balance
  - distribution of workload to different cores
  - for an application to scale, all processes must do an equal amount of work
- Improving parallel performance = minimizing parallel overhead

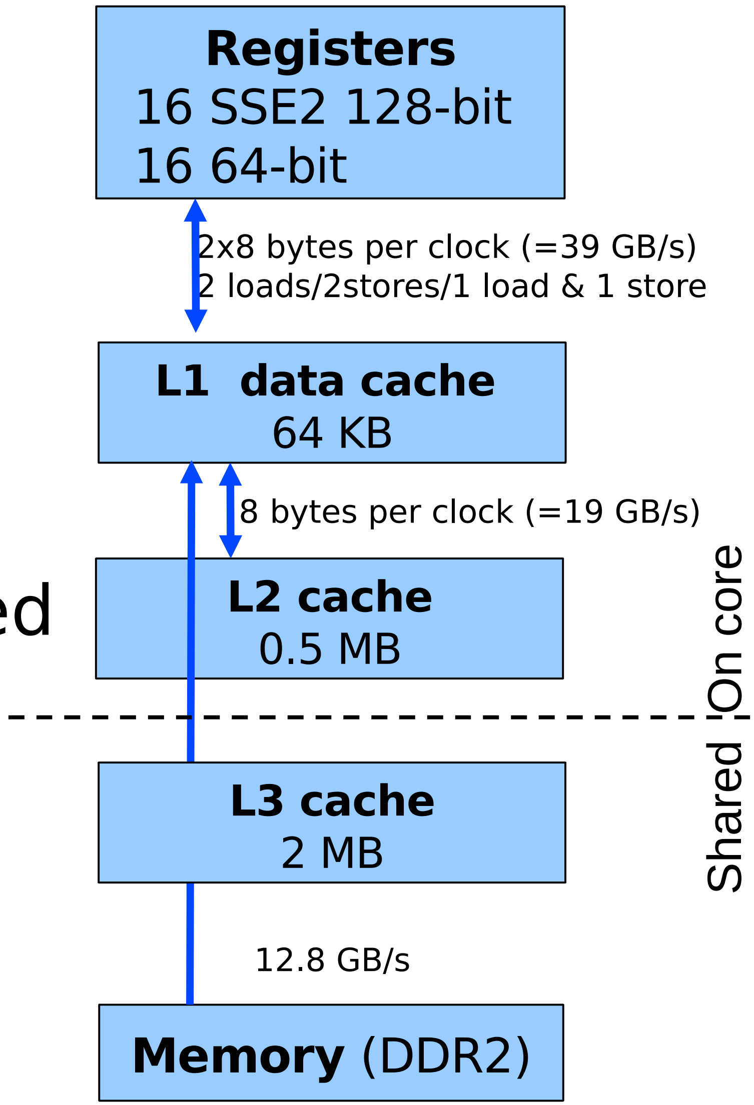
# Understanding serial performance

- Several stages are required to execute one instruction, for example:
  - Instruction fetch and decode (IF)
  - Read data (RD)
  - Execution (EX)
  - Write-back (WB).
- Independent instructions can be executed concurrently in the different pipe-line stages
- Works like an factory assembly-line



# Understanding serial performance

- Memory hierarchy
  - Only the data in registers can be accessed within one CPU cycle
  - Otherwise it has to be fetched from memory
- Improving serial performance = minimizing the amount of needed CPU cycles
  - Amount of instructions
  - Avoiding empty cycles
  - Utilizing special registers

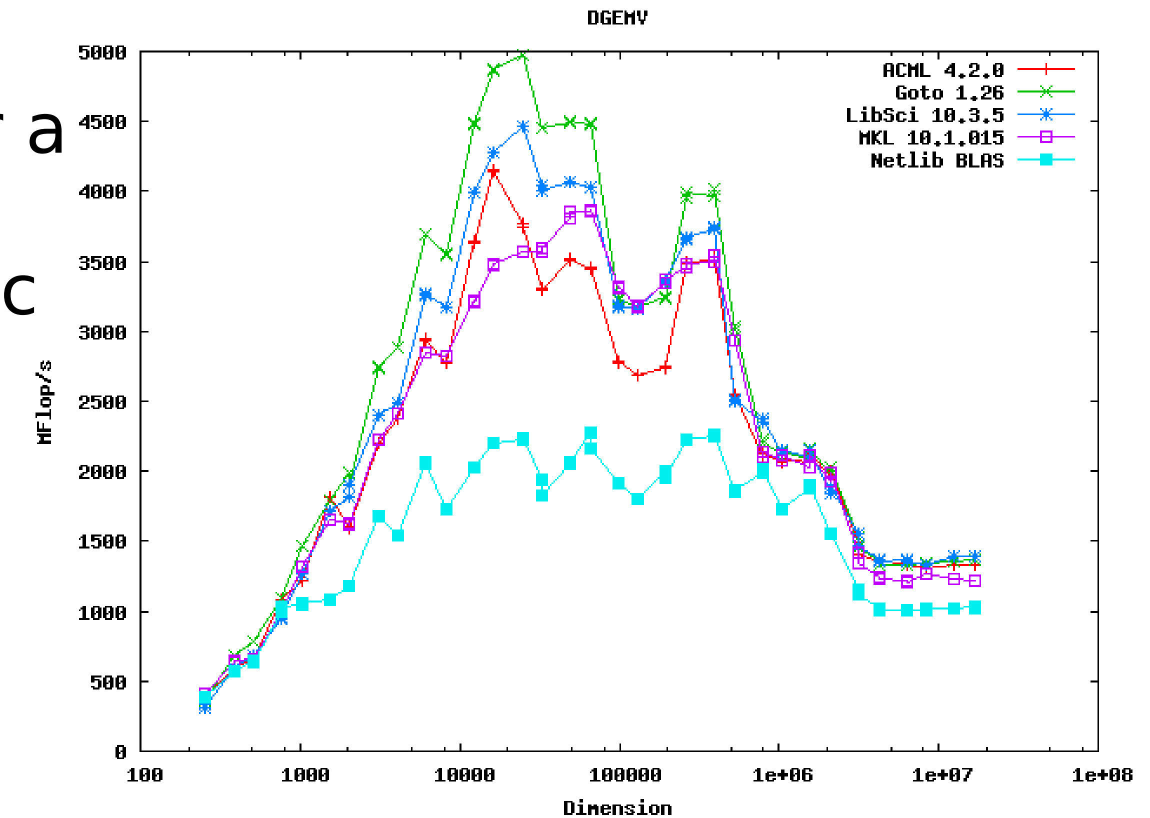


# Four easy steps to decent performance

- Utilize tuned libraries everywhere
- Employ compiler optimization
- Find suitable settings for environment variables
- Mind the I/O

# Numerical libraries

- Numerical libraries usually yield significant performance improvement
- Do not reinvent the wheel but use a performance library correspond of a mathematical operation whenever applicable!
- The libraries tuned for a system or a processor outperform the generic open source implementations



# Enabling compiler optimization

- Data dependencies (actual or potential) and conditionals forbid the compiler to optimize the code
- To obtain better optimization by the compiler
  - Avoid bulky loops
  - Write simple and readable structures
  - Avoid conditionals inside loops
  - Avoid function/subroutine calls from loops
  - Guide the compiler with pragmas
  - Use local constants

# Enabling compiler optimization

- “An optimization flag improved the performance but changed the results”
- Find out where the error arises
- Does that piece of code take a significant portion of the execution time?
    - No: optimize elsewhere on the code with the beneficial but problematic flag
    - Yes: try to restructure the loop such that the result remains correct

# Compiler optimization flags

| | PGI | Pathscale | GNU |
|---|---|---|---|
| Basic | `-O`*n*<br>`-tp` *arch* | `-O`*n*<br>`-march=`*arch* | `-O`*n*<br>`-march=`*arch* |
| Interprocedual optimization | `-Mipa=fast`<br>`-Minline` | `-ipa`<br>`-inline` | `-finline-functions` |
| SSE2 vectorization | `-Mvect=sse`<br>`-Mscalarsse` | `-LNO:simd=`*n* | `-ftree-vectorize` |
| Faster floating point math | `-Mfprelaxed` | `-ffast-math`<br>`-OPT:fast_math=ON` | `-ffast-math` |

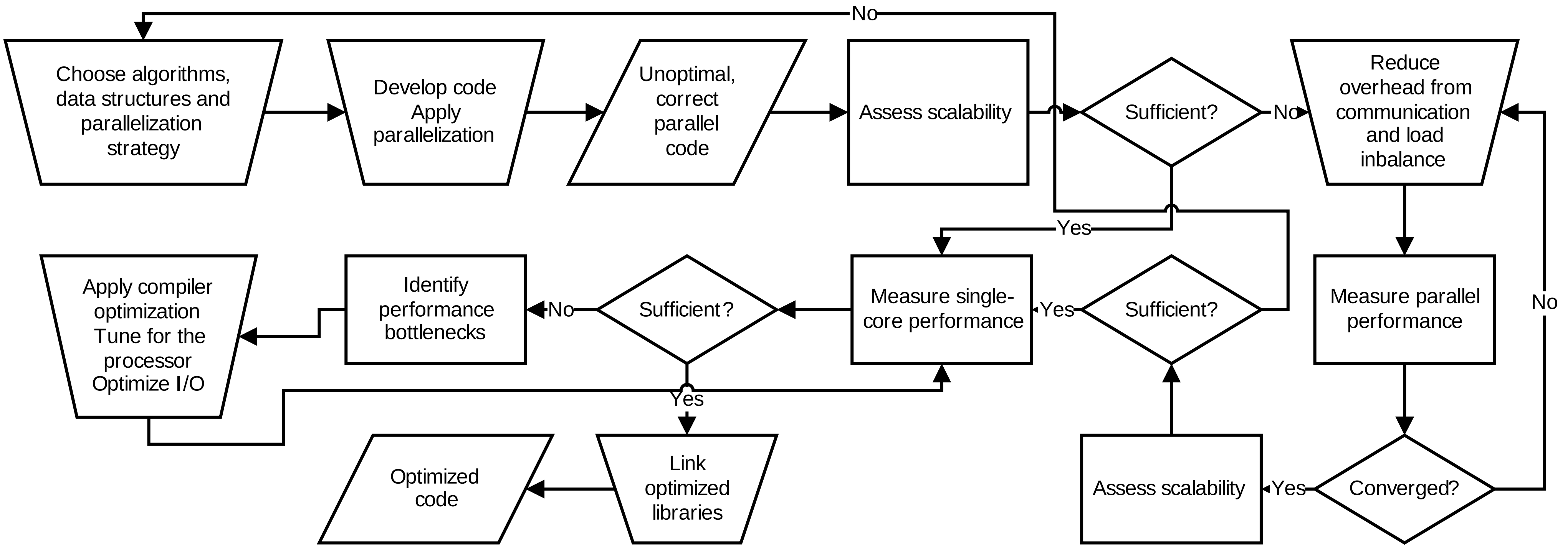
# Quick guide for impatient people

| | PGI | Pathscale | GNU |
|---|---|---|---|
| Basic | `-O2` | `-O2` | `-O2` |
| Intermediate | `-fastsse` | `-O3`<br>`-OPT:Ofast`<br>`-LNO:simd=1` | `-O3` |
| Aggressive | `-fastsse`<br>`-O3`<br>`-Mipa=fast,inline`<br>`-Mfprelaxed` | `-Ofast`<br>`-LNO:simd=2` | `-O3`<br>`-funroll-loops`<br>`-ffast-math` |

# Efficient I/O

- I/O is always an issue - make sure that you ask only for the minimum amount of I/O
  - Do you need all that checkpointing?
  - Do you need all that output?
- Use parallel I/O
  - Create natural partitioning of data so that it will go to disk in a way that makes sense
  - The easiest way for parallel I/O with good performance is to use MPI-I/O or some high-level library (HDF5, netCDF)
- Filesystem parameters (e.g. Lustre striping) have a major role

# Application optimization



# Further information

- If interested in code optimization, participate
  - Code optimization workshop at CSC March 22-24 http://www.csc.fi/english/csc/courses/archive/code-opt-10
  - CSC Summer School June 12-20 http://www.csc.fi/hpc/summerschool
- A. Hoisie: *Performance Optimization of Numerically Intensive Codes* (SIAM 2001)
- R. Gerber *et al. The Software Optimization Cookbook* (Intel Press, 2005)