INSTITUTE FOR MICROELECTRONICS
nssc@iue.tuwien.ac.at

Numerical Simulation and Scientific Computing I
Exercise 2

Submission is due on Tuesday, November 24th 2022, 8am (submit via TUWEL)

- Include the name of all group members in the documents you submit.
- Submission is done via TUWEL (same as Exercise 1): only one group member is required to submit for the group.
- The binaries have to be callable with command line parameters as specified below.
- Submit your answers (including the plots/visualizations) in one pdf-file per task.
- Submit everything as one zip-file including **one subfolder per task**.

General information

- Use the GNU compiler collection to compile your binaries with at least these flags: `-std=c++14 -Wall -pedantic`
- Use (and assume) the `double` precision floating point representation for all matrices/vectors.

Task1 - Vector Triad

Benchmark a *Vector Triad* `A[i] = B[i] + C[i] * D[i]` on your system utilizing the provided C++ source code in `benchmark_triad.cpp`.

1.1 Benchmark and Visualization (1 points)

Adopt the provided source code to your needs (i.e., by changing/extending the dimensions of the vectors to better refelect the cache sizes of your system, or changing the way results are reported). More specifically you should

- create a plot displaying your results: vector length (x-axis) vs. FLOPS/s (y-axis),
- indicate the cache sizes of your system (i.e., add vertical lines where the cummulated memory footprint of all four vectors equals the respective cache level size), and
- discuss your results.

Task2 - Matrix-Matrix Multiplication

Estimate and benchmark the performance of different implementations of dense matrix-matrix multiplication (MMM) executed using a single thread.

2.1 Performance Modeling (1 points)

(a) Calculate a theoretical single threaded *machine balance*

$$B_m = \frac{\text{memory bandwidth [Byte/s]}}{\text{peak performance [FLOPs/s]}}$$

for the CPU which you will later use for benchmarking.

(b) Calculate the theoretical ratio of required data traffic and required number of FLOPs (*code balance*)

$$B_c = \frac{\text{data traffic [Bytes]}}{\text{floating point operations [FLOPs]}}$$

for a MMM of square matrices of size $N \times N$.

(c) Find if and for which matrix size N a MMM would utilize the peak performance assuming idealized conditions.

(d) Calculate the theoretical single threaded runtime of a MMM on your benchmark system using N=1000, N=2000, N=5000 assuming the MMM utilizes peak performance.

2.2 Performance Benchmarking (2 points)

Benchmark the single threaded performance of a MMM

- using your implementation from the previous exercise[1](= option `CUSTOM`) ,
- using `cblas_dgemm` routine of the *OpenBLAS*[2]) (= option `OPENBLAS`), and
- using the `Matrix<double,Dynamic,Dynamic>` class (aka `MatrixXd`) of *EIGEN*[3] (= option `EIGEN`),

by creating an executable which takes two command line arguments and is callable by
`./benchmark impl size` e.g., `./benchmark CUSTOM 1024`
where `impl` is one of the three strings {`CUSTOM,OPENBLAS,EIGEN`} and `size` is the dimension N of a $N \times N$ square matrix.

More specifically, your programm should:

- create and initialize a matrix M of requested `size` using $m_{ij} = (i + j * N)$ where $i, j = \{1, 2, ..., N\}$,
- create a second independent matrix M^Twhich is the transpose of M,
- multiply M with M^T using the requested `impl` and track the runtime of this multiplication,
- perform a check if the resulting matrix is symmetric (as expected), and
- report the result of the symmetry check and the runtime (only of the MMM) to the console.

Finally, measure the runtime for all three implementations for $N = \{64, 128, 256, 500, 512, 1000, 1024, 1500\}$ and generate a plot of the results. Additionally, visualize the theoretical peak performance in the plot and provide information about the CPU and briefly discuss your results.

Hint: To disable multithreaded execution you can use the `-DEIGEN_DONT_PARALLELIZE` compile flag for *EIGEN* and the function `openblas_set_num_threads(1)` for *OpenBLAS*. If you are curious, you can investigate the effect of the compiler flags `-march=native` and `-DEIGEN_DONT_VECTORIZE` on *EIGEN*'s performance.

Task 3 - Finite Difference Discretization

Derive a finite difference (FD) discretization for a given elliptic partial differential equation (PDE), discuss properties of the resulting linear system of equations (LSE), implement an iterative solver for the LSE, and analyse its convergence.

3.1 Test Function (1 point)

Show that

$$u_p(x, y) = sin(2\pi x)sinh(2\pi y) \tag{1}$$

is the solution to the elliptic PDE of the form

$$-\Delta u(x, y) + k^2 u(x, y) = f(x, y) \quad \text{, with } k = 2\pi \tag{2}$$

on the two-dimensional unit square domain

$$\Omega = [0, 1] \times [0, 1] \tag{3}$$

for the right-hand side (RHS)

$$f(x, y) = k^2 u_p(x, y) \tag{4}$$

and boundary conditions (BC) on $\delta\Omega$

$$u(0, y) = 0 \tag{5}$$
$$u(1, y) = 0 \tag{6}$$
$$u(x, 0) = 0 \tag{7}$$
$$u(x, 1) = sin(2\pi x)sinh(2\pi) \tag{8}$$

[1] you can also use the reference implementation provided with this exercise
[2] `https://github.com/xianyi/OpenBLAS`
[3] `https://gitlab.com/libeigen/eigen`

3.2 Discretization (1 point)

Consider a FD discretization of the PDE (2) using the domain (3), RHS (4), and BCs (5)-(8) given above. Assume a discretization of the unit square with a regular grid spacing $h_x = h_y = h = \frac{1}{N-1}$ resulting in an LSE

$$A_h u_h = b_h \tag{9}$$

where A_h is the system matrix, b_h is the RHS and u_h is the solution to the LSE.

(a) Derive a FD stencil for an active grid point in the domain (i.e., blue points in Figure 1) when using a second-order central FD to approximate the second derivatives in (2).

(b) Represent (e.g., using block matrix notation) the system matrix A_h of the LSE resulting form the discretization of the PDE. Use the stencil derived above and follow the odering for the unknowns illustrated on the right side of Figure 1.

(c) Show whether the system matrix is *strictly diagonally dominant* for a grid point numbering scheme illustrated in Figure 1.

(d) Show whether the *Jacobi method* converges for the LSE resulting from the discretizetion.

3.3 Stencil-Based Matrix-Free Jacobi Solver (3 point)

For the LSE derived in Section 2.2., implement a stencil-based matrix-free Jacobi solver using your own data structures (i.e., only the C/C++ standard libraries).

Create a program which is callable by

`./stenciljacobi resolution iterations` e.g., `./stenciljacobi 32 500`

where `resolution` defines the grid spacing as $h = 1.0/(\texttt{resolution} - 1)$, and `iterations` defines the number of Jacobi iterations to perform. Further and more specifically, your program should

- use $\bar{u}_h = \mathbf{0}$ as initial approximation to u, and (after finishing all iterations)
- print the Euclidean $\| \cdot \|_2$ and Maximum $\| \cdot \|_\infty$ norm of the residual $\| A_h \bar{u}_h - b_h \|$ and of the total error $\| \bar{u}_h - u_p \|$ to the console, and
- print the runtime for the total number of iterations of the Jacobi method to the console.

Finally, use your program and create two plots:

- Plot both norms (Euclidean and Maximum) of the residual and total error in a plot using `resolution`=256 and `iterations`=$10^3 \cdot \{1, 5, 10, 50, 75, 100, 150\}$.
- Plot both norms (Euclidean and Maximum) of the total error in a table or plot using `resolution`=$\{16, 32, 64, 128, 256\}$ using a sufficient number of iterations.

Briefly discuss the results in your plots.

Hints: The regularity of the discretrization in this exercise results in identical FD-stencils for all active grid points. This makes it attractive to perform the update of each solution entry directly using only the stencil and an indexing scheme (without constructing any system matrix). You should **NOT explicitly construct any representation of the full system matrix of iteration matrix**.

3.4 Dense Jacobi Solver (1 point)

For the LSE derived above, implement a Jacobi solver using the dense-matrix functionality of the *EIGEN* library (i.e., explicitly storing the iteration matrix and performing the Jacobi iterations in a loop). Create a program which is callable by

`./densejacobi resolution iterations` e.g., `./densejacobi 32 200`

where the parameters have the same meaning as for the `./stenciljacobi` Jacobi solver. The program should print the same norms and runtime to the console as `./stenciljacobi`.

Finally, compare and briefly discuss the runtimes for the Jacobi iterations using `./stenciljacobi` and `./densejacobi` using `resolution`=32 and `iterations`=1000.

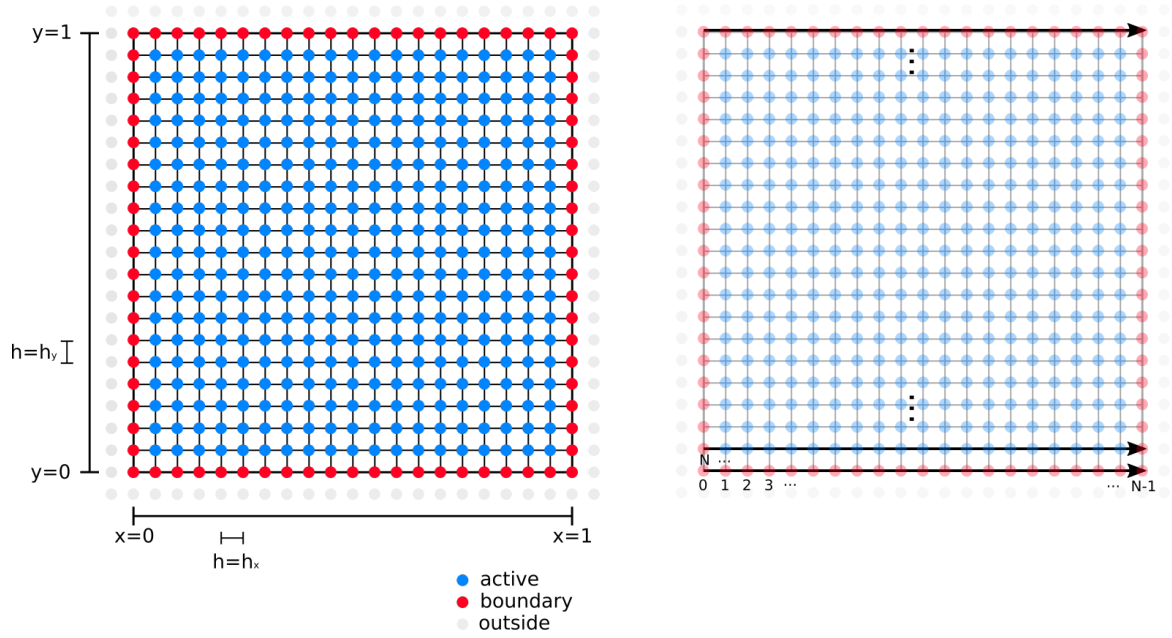

Figure 1: Left: discretization of the unit square using a regular grid with spacing h. Right: numbering scheme for the unknowns.