Literature

[1] GEMM: From pure C to SSE optimized micro kernels. `http://apfel.mathematik.uni-ulm.de/~lehn/sghpc/gemm/index.html`. Accessed: 2016-10-21.

[2] How to optimize GEMM. `https://github.com/flame/how-to-optimize-gemm/wiki`. Accessed: 2016-10-21.

[3] Intel C++ compiler 16.0 user and reference guide. `https://software.intel.com/en-us/intel-cplusplus-compiler-16.0-user-and-reference-guide`. Accessed: 2016-10-21.

[4] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[5] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Softw.*, 41(3):14:1–14:33, June 2015.

1 Auto-vectorization

Please read the chaper "Key Features: Automatic Vectorisation" of **Intel C++ Compiler User and Reference Guides** and answer following questions:

- Which kinds of loops can be vectorized automatically?
- Which datatypes and operations are allowed in order to enable auto-vectorization of loops?
- Which types of dependency analysis do the compiler perform?
- How does programming style influence auto-vectorization?
- Is there a way to assist the compiler through language extensions? If yes, please give details.
- Which loop optimizations are performed by the compiler in order to vectorize and pipeline loops?

2 Dependency checks and vectorization

The file `vector.c` contains several routines with loops. Some of them can be executed using vector instructions. Please indicate which routines/loops are vectorizable. Please give conditions, which allow or prohibit vectorization. Additionally, please change parts of the code and use compiler directives in order to increase the portion of vectorized sub-routines. Use the Intel compiler for all these experiments and look for hints within `vector.c`.

Tasks to complete:

- Please provide your personal opinion w.r.t. vectorization for each function.
- Which reports does the compiler provide? Which loops does the compiler vectorize and how does the compiler vectorize them?
- Please re-write functions/loops and use compiler-directives in order to vectorize additional loops.
- Hand-in the different compiler outputs (for compilations of `vector.c` with and without manipulations).

3 Vectorization across equal-shaped problems/sub-tasks

Application `gauss.c` provides an implementation of Gaussian-elimination without pivot-search but including backwards-substituation. It solves a system of linear equations with rank n directly. The system is given in form $A\vec{x} = \vec{b}$, with A as coefficient-Matrix, $\vec{b}$ as right hand side and $\vec{x}$ as solution.

First task: Sketch the given algorithm by using a figure of your choice.

Now, we want to solve a system of rank $n = 3$ for 2000 different right hand sides. However, solving a single linear system of equations with rank 3 cannot be implemented efficiently using vectorization. Please explain why.

Your third task is to change `gauss.c` accordingly in order to have a perfectly vectorized solver for systems with different right-hand sides. Consider the use of an optimal data-structure and please measure MFLOPS-rates for the original code and your optimized implementation. How large are your improvements?

4 Matrix-Matrix-Multiplication I

`dgemm.c` gives an implementation of a matrix-matrix multiplication.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \ , \qquad i, j = 1 \dots n.$$

First, please complete following tasks:

- Examine the memory access of the given implementation.
- Use different compiler-directives (Intel compiler) in order to vectorize the given code.
- Measure MFLOPS rates and cache-miss-rates for different problems sizes and plot your results. Please use `Valgrind` for measuring the cache-miss rate.
- Do different compiler options (please use the compiler manual for more aggressive options) influence the performance of the implementation.

- For which problem size do you get the best performance?
- Is the obtained performance stable?
- Please explain the obtained performance using hardware properties (such as cache size, memory bandwidth, or similar).

Afterwards, try to increase the MFLOPS-rate by employing *cache-blocking*. This can be done by spitting two loops in order to block the given matrix multiplication in L1 cache. How do performance and cache-miss rate now change with increasing the problem size?

Starting from your optimally blocked version, please implement your matrix multiplication using vector-intrinsics (tip: consider the `_mm256_broadcast_sd`-instruction). Is your new code faster than the version generated by the Intel compiler?

Tips for using the cluster

Using valgrind on the cluster

- Load module valgrind with `module load valgrind`.
- Compile your application with an additional flag: `-g`.
- Start valgrind by type executing: `valgrind --tool=cachegrind myprog [myprog-options]`.

Have fun!

Deliverables

The following deliverables have to be handed in no later than 08:00 AM, Monday, 7th November, 2016. If there is no submission until this deadline, the exercise sheet is graded with 0 points. Small files (<1 MB in total) can be send as an attachment directly to *uphoff AT in.tum.de* and *chaulio.ferreira AT tum.de*, larger files have to be uploaded at a place of your choice, e.g. `https://github.com/`, `http://home.in.tum.de/`, `https://syncandshare.lrz.de`. In either case inform us about the final state of your solution via e-mail.

- A short report which describes your work and answers all questions in this assignment.
- All of your code.
- Slides for the presentation during the next meeting; remember to address all questions in the tasks.
- Output of all runs. Figures (e.g. scaling graphs) if applicable.
- Documentation how to build and use your code.