Department of Informatics — TUM
Scientific **C**omputing in **C**omputer **S**cience
Michael Bader, Alexander Breuer

WS 2014/15

Masterpraktikum Scientific Computing (High Performance Computing) Exercise Sheet 1: Vectorization

Tutorial on 10/13/14

Exercise 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?

Exercise 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?
- 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)!

Exercise 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?

Exercise 4 "Matrix-Matrix-Multiplication I"

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

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

First, please complete following tasks:

- Examine the memory access of the given implementation!
- Use different compiler-directive (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 `_mm_loaddup_dp`-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!

Deadline: 10/27/14, 08:00 AM! Please mail to **breuera AT in.tum.de**. If there is no submission until this deadline, the exercise sheet is graded with 0 points!

Please download applications-frames from
`http://www5.in.tum.de/wiki/index.php/Masterpraktikum_Scientific_Computing_-_High_Performance_Computing_-_Winter_14`