November 7, 2017

DVA336 Parallel Systems

# Assignment 2

## Vectorization (SIMD)

In this assignment, you will get acquainted with vectorization by writing programs that explicitly utilize the SIMD functionality that is available in modern CPUs. To access the SIMD instructions, you will use built-in functions or intrinsics, i.e., functions that map directly to SIMD instructions and that the compiler is intimately aware of. More specifically, the instruction set that you will use is the Intel Advanced Vector Extensions (AVX) and its extension (AVX2). To be able to use AVX intrinsics, you have to include the correct header file. You may also need to set certain compiler flags. See the documentation of your compiler for more details. Furthermore, it is here recommended that you build your programs as 64-bit, rather than, 32-bit, applications. The programming problems you need to solve as parts of this assignment are given below. You can work either in pairs or individually.

### 2.1 Simple statistics

1. Study the program in the file `simple_stat_AVX.cpp`. As you can see, there is a simple function written in standard C/C++ that finds the maximum value in a given array. Regard this function as the sequential baseline when you measure parallel speedup[1]. A corresponding AVX version of this function is also available. Study the program and make sure you understand how it works in detail. Then go on and measure the execution time of this computation for both versions. Run tests for several input sizes, say $n = 10^3, 10^4, 10^5, 10^6, \ldots$. How many times faster is the AVX version?

2. Modify the functions (both the standard C and the AVX version) so that they return, not only the maximum value, but also the average and the minimum values. As before, run and compare the relative performance of the serial and the AVX versions. How does this change effect the achieved speedup? What is the reason?

3. Extend the program so that the same computations can be carried out, not only for single precision 32-bit values (`float`), but also for arrays storing 64-bit values (`double`). What speedups do you get in this case? How would you explain the results?

[1] Of course, if your compiler applies auto-vectorization or other significant parallelization techniques to this function, this may not be appropriate.

## 2.2 Computing vector lengths

A. In the file `vector_len_AVX.cpp`, a sequential function is given that finds and stores the lengths of $n$ given vectors in $\mathbb{R}^2$. The vectors are represented by two arrays, one with the $x$-coordinates and another with the $y$-coordinates. The resulting lengths are written to a third array. Use AVX to implement a SIMD version of this computation. What speedup do you get compared to the given standard C/C++ version?

B. Modify the functions (both the standard C/C++ and your own AVX version) so that the index of the vector that has the maximum length is returned as well, besides the output array with all the vector lengths. How does your modification affect the speedup?

C. Suppose the vectors would instead have been given in a single flat array with the $x$ and $y$ coordinates intertwined ($x_0y_0x_1y_1x_2y_2\ldots$). How would you implement the AVX version in this case? How would it affect the performance?

## 2.3 Finding the closest point pair (*Optional*)

Given a set of $n$ points in $\mathbb{R}^2$, find the closest point pair, i.e., the two points that has the minimum Euclidean distance to each other. There are efficient algorithms that solve this problem in $O(n \log n)$ time. Here, however, you are allowed to use a naive $O(n^2)$ algorithm that considers all possible point pairs. Design the algorithm and implement both a serial version and an AVX version of it. Convince yourself that both versions give correct answers for any input. Then measure the execution times and make a nice plot to visualize the obtained speedups for various input sizes.

## 2.4 Vectorization and multithreading

Use OpenMP to further parallelize one of the programs you have created so far in this assignment. In this way, your program will also take advantage of multiple processor cores. Clearly, since each core has its own SIMD unit, the combined effect of multi-threading and SIMD vectorization should be interesting. When using $m$ cores and $k$-way data parallel SIMD operations, what is the maximum theoretical speedup you can get? What is the theoretical maximum on the computer you use? What speedup do you actually get in practice? Try to explain your results.

# Examination

Make an appointment with the lab assistant and show your programs in action. Also, be prepared to answer—individually—any questions concerning your solutions. Furthermore, send in your source code with your solutions together with a lab report that present your experimental results as well as the answers to the questions above in a compressed archive file by e-mail to the lab assistant. Use either the zip or gzip archive format.