Institut für Informatik — TUM
Scientific **C**omputing in **C**omputer **S**cience
R. Wittmann, G. Buse

WT 2013

Tutorial
(Advanced Programming)
Worksheet 7:

This worksheet revolves around the topic of *vectorization*. We want to get a feeling of how hard it is to achieve vectorization in simple linear algebra tasks and want to study the performance impact. Further, we want to make sure that our code actually uses vector registers. To do so, we will have a short glimpse at the assembly generated by the compiler.

Background Knowledge 1: Disassembly

In order to judge whether the compiler maps our source code calculations to actual vectorized code we can either turn on verbose output for the compiler's vectorizer, or we can have a look at the intermediate assembler code that was generated.

The first variant is rather easy to accomplish (man page). The second variant is rather advanced but offers more control, as it is directly revealed if the compiler introduces unnecessary instructions in the machine code. Activating compiler optimizations makes the task even more difficult, as single computations might get shifted around. Still, we want to do this kind of analysis on this worksheet just for the basic example of adding two vectors.

- If you prefer the command line, the command `objdump` is useful. `objdump -d BINARY` for instance shows the assembler instruction of the executable part of the binary file `BINARY`.
- A more convenient way is provided by the eclipse IDE. The disassembly is matched with the source code during debugging sessions, helping the programmer to understand which high level statements correspond to which instructions in the processor. To open the disassembly view, start a debugging session and choose "Window→Show View→Disassembly" from the menu.
  Make sure you compile your code with debugging symbols (`-g` option).

Background Knowledge 2: Intrinsics

With the Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX), Intel has released extensions to the x86 architecture's instruction set, which have by now been adopted by other processor manufacturers. The extensions come with an API consisting of so-called compiler *intrinsics* or *builtins*, functions that are built into the compiler to ensure maximum efficiency. The "Intel Intrinsics Guide" is a very useful tool for developers who want to get an overview of available intrinsics. It can be freely downloaded from Intel's website.

Consult this guide (available for various platforms) to find out whether the instructions you find in the disassembly are vector instructions or not.

Assignment 1: Using Auto-Vectorization for Adding Two Vectors

Write a program with a simple loop that adds two float vectors of equal length. Make sure the vector size is sufficiently large so vectorized code can be generated.

Compile the program with different optimization flags (`-O` option). Check the disassembly and whether it changes for different optimization levels. Try to match the instructions in the disassembly with those listed in the Intel Intrinsics Guide (focus on "Floating Point" combined with "Load" and "Arithmetic").

Hint:
When using GCC under linux, use the `-march=native` compiler switch (you can look up its meaning in the gcc man page).

Assignment 2: Using Intrinsics for Adding Two Vectors

Rewrite the program from the previous assignment, only this time use intrinsics to generate vectorized code.

1. Can you come up with the same disassembly as generated by the auto-vectorizer?
2. What happens if you port your code from Intel's current Haswell architecture to an older one, e.g. Intel Westmere from 3 years ago? Is your code still executable?
3. What is different, if you want to use intrinsics to compute the inner product of the vectors?

Homework Assignment 1: Aligned Memory

On a previous worksheet we have analyzed how the compiler arranges variables in memory. Intrinsics data types are defined such that the compiler ensures their alignment to 16 (SSE) or 32 bytes (AVX) in memory, respectively. However, filling the vector registers with data is also most efficiently done when loading from arrays that are 16- / 32-byte-aligned.

Look up C/C++ routines that allocate aligned memory and use them in your code. Enforce efficient loading by choosing the right statements from the list of intrinsics provided by the Intel Intrinsics Guide.

Can you imagine how aligned, vectorized loading can be ensured without the explicit use of intrinsics?