Chair of Applied Computer Science III
Prof. Dr. Guido Moerkotte
Email: moerkotte@uni-mannheim.de
Daniel Flachs
Email: daniel.flachs@uni-mannheim.de
UNIVERSITY OF MANNHEIM

Database Systems II
Spring Semester 2019

Exercise Sheet 2
Created March 4, 2019

Exercise 1

Download the `zip` archive for this exercise sheet from the website and use the provided make file to compile `experiments1.cc`.

Exercise 1 a)

First, run `./experiment1`. Then, run `./experiment1 -r`. You will get different outputs. Analyse the code and interpret the observed outputs.

Exercise 1 b)

Install the profiling tool *valgrind* (not available for Windows): `http://www.valgrind.org`. Rerun the above experiments, but this time, track the number of cache misses using valgrind:

```
valgrind --tool=cachegrind ./experiment1 -n 1000000
valgrind --tool=cachegrind ./experiment1 -n 1000000 -r
```

Exercise 2

In this exercise, we implement some functions using SIMD instructions to speedup computation.

Exercise 2 a)

Make yourself familiar with the SIMD instructions applicable to your computer's architecture. For Intel machines, the following resources might be interesting for you:

- SIMD tutorial by Jacco Bikker, Utrecht University: `http://www.cs.uu.nl/docs/vakken/magr/2017-2018/files/SIMD%20Tutorial.pdf`.
- A practical guide to SSE SIMD with C++ by Tuomas Tonteri: `http://sci.tuomastonteri.fi/programming/sse`.
- Intel Intrinsics Guide: `https://software.intel.com/sites/landingpage/IntrinsicsGuide/#`.

When compiling your program, the following compiler flags are relevant[1]:

- Optimization level `-O<number>`:
  `-O0` (no optimization, default setting) to `-O3` (highest optimization).

- `-ftree-vectorize`, `-fno-tree-vectorize`: compiler option (starts with `-f`) that enables/disables auto vectorization, i. e., the automatic replacement of simple instructions by SIMD instructions. Note that, if `-f[no-]tree-vectorize` is not specified, auto vectorization is implicitly disabled for `-O2` and below, but enabled for `-O3`.

- `-msse`, `-msse2`, `-msse3`, `-msse4`, `-msse4a`, `-msse4.1`, `-msse4.2`, `-mavx` and the respective `-mno-*` flags: machine-dependent compiler options (start with `-m`) that enable/disable the respective SSE/AVX instructions. To check which of the above-mentioned are enabled by default, you can ask the compiler:
  ```
  g++ -Q --help=target | grep -P "\-m(sse|avx).*\[enabled\]"
  ```
  Observe how the output of the above command changes if you add `-march=native` (see below) as another compiler flag.

- `-march=native`: "This flag selects the CPU to generate code for at compilation time by determining the processor type of the compiling machine. Using `-march=native` enables all instruction subsets supported by the local machine (hence, the result might not run on different machines)."

Exercise 2 b)

Implement a function that takes two integer arrays by pointer and adds them componentwise into a third array. To allow for dynamic allocation at runtime, the arrays should be stored on the heap. First, use non-SIMD instructions to implement the function in a naïve, unoptimized way. Then, reimplement your function using 128- or 256-bit SIMD instructions. What performance gain do you expect? Measure the execution time of your functions using different compiler optimization settings.

Note that the three arrays need to be aligned in memory if you want to load values directly into the SIMD registers. To ensure alignment, you can allocate memory using

```
int posix_memalign(void **memptr, size_t alignment, size_t size);
```

from the `cstdlib` header. The provided file `memalign_example.cc` gives an example of how to use this function.

Exercise 2 c)

Implement a function that takes one integer array representing an n-dimensional vector and computes this vector's Euclidian norm[2]. First, use non-SIMD instructions to implement the function in a naïve, unoptimized way. Then, reimplement your function using

[1] Sources: `https://gcc.gnu.org/onlinedocs/gcc-8.2.0/gcc/x86-Options.html` and `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

[2] See `https://en.wikipedia.org/wiki/Norm_(mathematics)#Euclidean_norm`.

128- or 256-bit SIMD instructions. What performance gain do you expect? Measure the execution time of your functions using different compiler optimization settings.

Exercise 3

Hint: Read "Programming Pearls" (2nd edition) by Jon Bentley, Column 14.

Exercise 3 a)

Implement a min priority queue class based on a heap data structure.
Apart from the constructor, your class must provide two public functions:

- `insert(item)`: Insert an item
- `extractmin()`: Delete smallest element from priority queue and return it

Exercise 3 b)

For performance comparison, implement two more priority queue classes, one based on an unsorted array and the other based on a sorted array.
The asymptotic runtimes of the different priority queue implementations look as follows:

Date Structure	`insert`	`extractmin`	n of each
Heap	$O(\log(n))$	$O(\log(n))$	$O(n \cdot \log(n))$
Sorted Array	$O(n)$	$O(1)$	$O(n^2)$
Unsorted Array	$O(1)$	$O(n)$	$O(n^2)$

Create an array A of n random elements. Then, for each priority queue class implementation,

- create a priority queue object,
- call `insert` for each element in A, and
- call `extractmin` n times.

Measure the runtimes. For what sizes of n does the heap-based implementation run faster than its competitors on your computer?

Exercise 3 c)

Use your heap-based priority queue class to implement the sorting algorithm *heapsort*. You do not have to sort in-place, i. e., space complexity $O(n)$ is okay for this exercise. For comparison, implement *insertion sort* as well.

Create an array A of n random elements. For what sizes of A does heapsort run faster than insertion sort on your computer?