Code optimization
Exercise 4

1. 6-bit floating-point representation

Fill in the non-negative numbers that can be represented with a 6-bit floating-point format based on the IEEE standard in the table below. The 6-bit format uses 3 exponent bits and 2 significand bits, and one sign bit. Since the exponent is 3 bits, the bias is $2^{3-1}-1 = 3$.

Description	Bit pattern	Biased exponent	Unbiased exponent	Exponent value	Significand value	Floating-point value
Zero	0 000 00	0	-2	$2^{-2} = 1/4$	0/4	0/16 = 0.0
Denormal values	0 000 01	0	-2	$2^{-2} = 1/4$	1/4	1/16 = 0.0625
	0 000 10					
	0 000 11					
Normal values	0 001 00	1	-2	$2^{-2} = 1/4$	4/4	4/16 = 0.25
	0 001 01					
	0 001 10					
	0 001 11					
	0 010 00					
	0 010 01					
	0 010 10					
	0 010 11					
	0 011 00	3	0	$2^{0} = 1$	4/4	4/4 = 1.0
	0 011 01					
	0 011 10					
	0 011 11					
	0 100 00					
	0 100 01					
	0 100 10					
	0 100 11	4	1	$2^{1} = 2$	7/4	14/4 = 3.5
	0 101 00					
	0 101 01					
	0 101 10					
	0 101 11					
	0 110 00					
	0 110 01					
	0 110 10					
	0 110 11	6	3	$2^{3} = 8$	7/4	56/4 = 14.0
Infinity	0 111 00	–	–	–	–	$+\infty$
NaNs	0 111 01	–	–	–	–	NaN
	0 111 10	–	–	–	–	NaN
	0 111 11	–	–	–	–	NaN

2. Show how the value -275.875 is represented as a 32-bit floating-point number in the IEEE 754 standard.
Please include all the steps of the calculation and describe in detail how you do the conversion to IEEE 745 format

3. Which is the smallest single-precision floating-point value *X.0* for which *X*+1.0f = *X* ?

In other words, what is the smallest single-precision floating-point value *X*.0 such that when you increment it by 1.0, the value does not increase, but remains unchanged?

An alternative formulation of the question is: which is the smallest single-precision floating-point value such that the next representable value is not the next larger integer value?

Hint 1: the value *X* is has a fractional part of zero, i.e. it is of the form *x.0*.
Hint 2: you can either write a small program that computes the value, or calculate the answer with pen and paper.

4. Below is a small program that computes the same value of *10*f* in three different ways. Execute the program and explain why the three results differ.

```
#include <stdio.h>
int main() {
  float f = 0.1f;
  float sum1 = 0.0f;

  // Calculate the result by adding
  for (int i = 0; i < 10; ++i)
    sum1 += f;

  // Calculate the result by multiplying
  float sum2 = f * 10.0f;

  // Print results, with a third calculation of the value
  printf("sum1 = %1.15f, sum2 = %1.15f, sum3 = %1.15f\n",
           sum1, sum2, f*10.0);
  return 0;
 }
```

5. Attached in the program `mm_blas.c` is yet another version of matrix multiplication, this time using the BLAS library procedure `dgemm` from OpenBLAS. The BLAS routines (Basic Linear Algebra Subprograms) are highly optimized and take advantage of multi-threaded execution and vectorization.

There are many BLAS implementations of high quality, for instance

- ATLAS – Automatically Tuned Linear Algebra Software (http://math-atlas.sourceforge.net)
- Intel Math Kernel Library (https://software.intel.com/en-us/mkl)
- AMD Core Math Library (https://developer.amd.com/amd-aocl)
- OpenBLAS (https://www.openblas.net)

a) Compile the program on Dione with full optimization (-O3) and test it with a matrix size 4000*4000. Compare the run time with the previous implementations of matrix multiplication. Also verify that the result is correct.

b) For this program execution, calculate how many floating-point operations per second (FLOPS) were achieved, in the same way as in Exercise 1.

The program must be compiled with an older version of GCC: gcc 7.3, since the OpenBLAS library on Dione was built with this version. This version of GCC is automatically loaded with the OpenBLAS module.

To compile the program, unload all previous modules and load the module OpenBLAS. This will automatically load the correct compiler version. You can use the Makefile to build the executable program.

When running the program, reserve a whole node (i.e. all cores on the node), since OpenBLAS uses multithreaded execution. The program takes two arguments: the size of matrices and the number of threads to use.

An example of how to compile and run the program is below.

```
$ module purge
$ module load OpenBLAS
$ make mm_blas
gcc -O3  -lm -lopenblas     mm_blas.c   -o mm_blas
$ srun -N 1 mm_blas 4000 20
  . . . output from the program
```