HPC Lab

Assignment 1

Erkin Kirdan - Manuel Rothenberg - Gabrielle Poerwawinata

1. CoolMUC3 warm-up

a.)
LRZ uses the modules system to manage the user environment for different software, library or compiler versions. The distinct advantage of the modules approach is that the user is no longer required to explicitly specify paths for different executable versions, library versions and locations of other entities needed for the execution environment.
To make the application-specific settings for <package> (the denotation is abstract) available in the current shell, the user types the command:

```
module load <package>
```

If, subsequently, the user types

```
module unload <package>
```

The default settings is performed automatically at login:

```
ssh -l <hpc_username> lxlogin.lrz.de
```

There are several commands that can be performed under this module system.

b.)
All programs in the segments of the cluster must be started up using either an interactive SLURM shell, where users type commands to run their jobs or a SLURM batch script, where users prepare their job scripts such as myjob.cmd which is similar to a bash script and then submit their jobs.
In the interactive mode you request resources with the command `salloc` and add parameters to specify how many nodes and how many tasks per node you need. After the resource was granted, you're able to run your program in this configuration my executing `mpiexec ./myprog.exe` right from the shell. In the batch-mode you create a script which contains your configuration needs. You start a job by submitting this script with the command `sbatch my_batch_script.cmd`. See an example of the batch-mode in our GIT-Repository. The interactive mode is more for testing, while the batch-mode should be used for production use.

c.) We developed a simple Hello World Program, which also prints the Hostname.

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char** argv){
     char hostname[1024];
     gethostname(hostname, 1024);
     printf("Hello World! (from: %s)\n", hostname);
     return 0;
}
```

We compiled it and ran it as a Batch-Job with the following settings:
(see GIT-Repository for full settings)

```
#SBATCH --nodes=2-2
#SBATCH --ntasks-per-node=1
```

This specifies, that the program should be executed on exactly two nodes with exactly one task per node. This resulted in the following output:

```
Hello World! (from: mpp3r03c02s01)
Hello World! (from: mpp3r03c02s02)
```

In the resulting log, we can see, that the code was executed on two different nodes. Both printed “Hello World!” and their hostname (mpp3r03c02s01 and mpp3r03c02s02).

2. Auto-vectorisation

Data structure alignment is the adjustment of any data object about other objects. It can be data alignment, data structure padding or packing. By the alignment, it is aimed at fitting data into memory words so that they can be used efficiently without any alignment fault. Data structure alignment is essential to start at specific addresses to speed up memory access since misaligned memory accesses can incur significant performance losses.

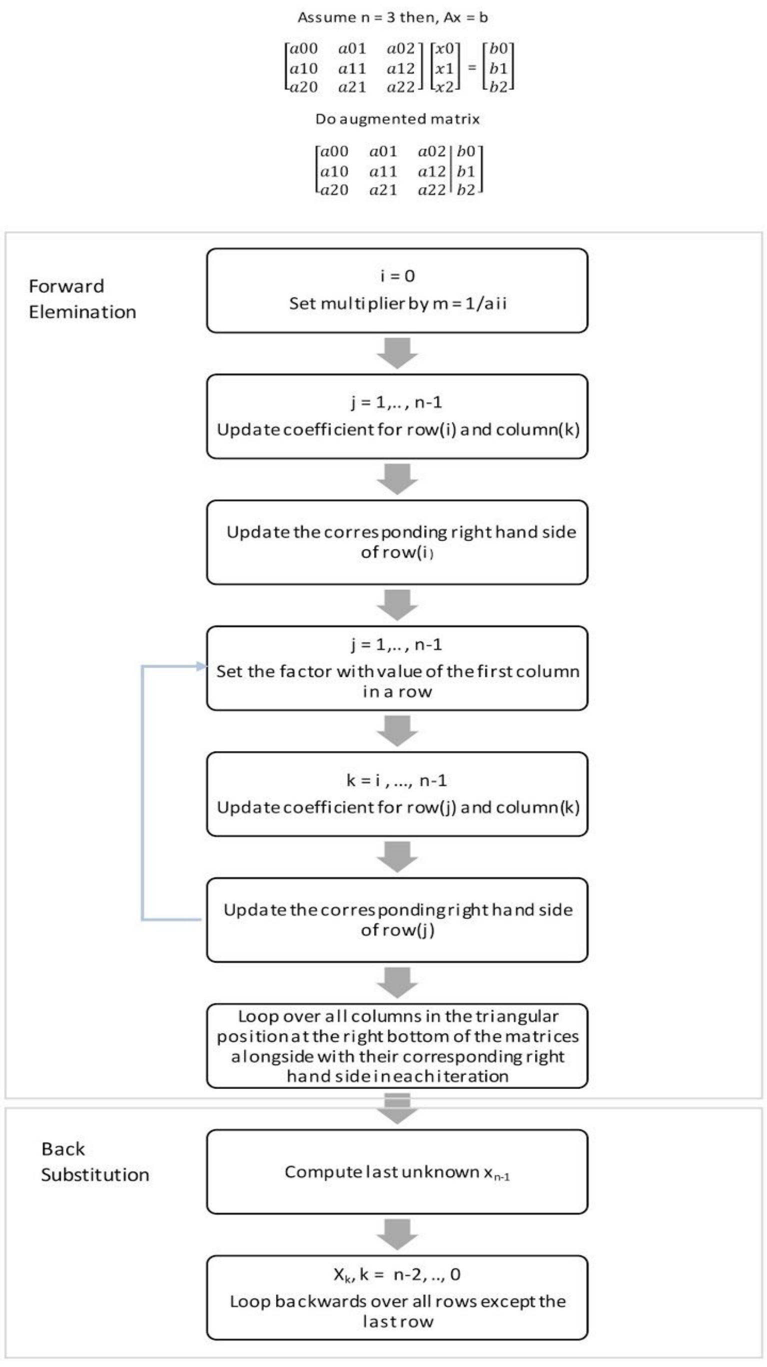
Non-contiguous memory access can prevent vectorisation. In such a case, integers must be loaded separately using multiple instructions, which is considerably less efficient. Data dependencies can also avert vectorisation is forms of read-after-write, write-after-read or write-after-write cases. These dependencies cause data hazards which lead to miscalculation. So to calculate a correct result, vectorisation is utilised less or not at all.

Yes, language extensions can be used to assist compiler for vectorisation. Preferring vector data types, intrinsic functions and especially putting pragmas to indicate options and blocks to be vectorised helps the compiler to vectorise the code more appropriately. Therefore, it is the developer's responsibility to think about vectorisation beforehand designing and developing the program so that during the vectorisation compiler could efficiently vectorise the code.

Loop optimisation aims at increasing execution speed and reducing the overheads associated with loops so that cache performance and parallel processing capabilities can be improved. Since most of the execution time of a typical scientific program is spent on loops, many optimisation techniques have been developed to make them faster. Some significant examples are fission (distribution), fusion (combining), the interchange (permutation), inversion, tiling (blocking), splitting (peeling).

3. Vectorisation across equal-shaped problems

The following figure shows the the given algorithm.


Assume n = 3 then, Ax = b
$\begin{bmatrix} a00 & a01 & a02 \\ a10 & a11 & a12 \\ a20 & a21 & a22 \end{bmatrix} \begin{bmatrix} x0 \\ x1 \\ x2 \end{bmatrix} = \begin{bmatrix} b0 \\ b1 \\ b2 \end{bmatrix}$
Do augmented matrix
$\left[\begin{array}{ccc|c} a00 & a01 & a02 & b0 \\ a10 & a11 & a12 & b1 \\ a20 & a21 & a22 & b2 \end{array}\right]$
Forward Elemination
i = 0 Set multiplier by m = 1/aii
j = 1,.. , n-1 Update coefficient for row(i) and column(k)
Update the corresponding right hand side of row(i)
j = 1,.. , n-1 Set the factor with value of the first column in a row
k = i , ..., n-1 Update coefficient for row(j) and column(k)
Update the corresponding right hand side of row(j)
Loop over all columns in the triangular position at the right bottom of the matrices alongside with their corresponding right hand side in each iteration
Back Substitution
Compute last unknown x_{n-1}
X_k, k = n-2, .., 0 Loop backwards over all rows except the last row

Solving a single linear system of equations with rank 3 cannot be implemented efficiently using vectorisation since each iteration is dependent on the preceding one. So data dependencies between iterations make it harder to vectorise such algorithms. For example, in the first iteration, vectorisation of row operations to obtain zeros in the first elements of the rows can be done. But further elements in the row is dependent on the preceding one. What can be done and actually what we have done in the exercise is that vectorisation of different linear system equations is possible since the data among them is independent.
The gauss.c file is changed so that can be vectorised. Basically, instead of having 2K A matrices and 2K different b vectors, we generated a huge A, and b vectors which can be vectorised easily since there is no dependency between each subset of b vectors. We also give a compiler hint by a pragma, to tell the compiler, it should vectorize our code. After the modification, the vectorisation is done, and the following results are obtained:

	Before	**After**
Gflops/s	0.117	0.427
s [us]	343	93.8
Gflops	0.4	0.4
Speedup	1	3.66

All files to reproduce results and optimization reports can be found under gitlab repository.

The optimization report (see gauss.optrpt in GIT) shows, that the outer loop was vectorized. That tells us, that not the inner loops which processes one matrix, were vectorized, but that the process itself was vectorized. A lot of matrices are processed parallel by vectorization, but not a single matrix.

4. The perfect DGEMM micro-kernel

To achieve a better performance for our DGEMM micro-kernel, we considered the following methods. We will discuss them in the following chapter.

- Memory Alignment
- Auto Vectorization
- Further Caching Optimization

In the given Makefile is a parameter ALIGNMENT, which is used, to get memory parts that start with an address that is a multiple of ALIGNMENT. First we set this parameter to 64 (byte), because the Vector Processing Unit (VPU) of the Intel Xeon Phi is 512-bit wide and the L2 Cache is composed of 64-byte cache lines. This setting helps to avoid cache-line splitting and therefore should boost Cache Performance.
Now we have memory, that is aligned to 64 byte. In our function we gave a hint to the compiler by the following commands/pragmas:

```
__assume_aligned(C, ALIGNMENT);
__assume_aligned(B, ALIGNMENT);
__assume_aligned(A, ALIGNMENT);
#pragma vector aligned
```

We added these lines right before the most nested loop, to tell the compiler, that A,B and C are aligned to 64 byte. The rest of the optimization should be done by the compiler.

Besides the memory alignment, we also told the compiler, that it is safe to vectorize our function. We did that by adding the following pragma right before the most nested loop.

```
#pragma ivdep
```

This tells the compiler, to ignore assumed dependencies, so that it will compile the code to use with vectorization. This should give us a better performance.

Furthermore we discovered, that in the most nested loop, it’s always C[n*M +m] that is added up. It is not depended on k, which is the loop variable of the most nested loop. That means, that for K successive loop cycles, we always access the same memory location.
It should give us a much better performance, if we can keep the content of this memory location as close to the processor as possible (L1, L2 Cache), to minimize Cache misses.
We did this, by adding the following code right before the content of the most nested loop.

```
__builtin_prefetch(&C[n*M + m], 1, 3);
__builtin_prefetch(&C[n*M + m], 0, 3);
```

This tells the compiler, to prefetch the memory content of the given location, because it is heavily used within the loop. The first parameter is the address of the memory location, the second one specifies, if we need the memory content for reading (0) or for writing (1). The third parameter specifies, how important the temporal locality is. 0 means not important and 3 means, very important (Keep it in as many cache levels as possible).
Since we need the memory content for reading and for writing and it is also very important, we added the above shown code.

The unchanged given code was executing with a performance of ~2.37 GFLOP/s. After adding all of our promising performance boosters, we still got just a result of ~3.1 GFLOP/s, which means a speedup of ~1.3. It's something, but we were sure, we can optimize that.

We tried all kind of things, to get a better performance. The main idea was, to optimize caching of C. But everything we tried resulted in the same GFLOP/s range between ~2.37 and ~3.1 or sometimes even worse.

For example, we tried:

- to vectorize not on the most nested loop, but also or exclusively on the outer loops
- to prefetch not on the most nested loop, but also or exclusively on the outer loops
- to add `#pragma omp parallel for` with the Compiler/Linker Flags
- to prefetch by other commands, pragmas and compiler parameters

We also wanted to try the following, but couldn't do/test that, since the CoolMUC3 was occupied or under maintenance on roughly 4 days this week.

- to add the `__restrict` hint for the function parameters
- to implement Loop Blocking (see commented code in dgemm.cpp)
- to add `-ansi-alias` as compiler parameter
- to add `-no-prec-div` as compiler parameter
- to tell the compiler, that C is nontemporal `#pragma vector nontemporal (C)`

From all of these possibilities, we think Loop Blocking would be one of the best (at least for bigger M and N), since it uses efficiently the data content of one complete cache line. If this cache line data is processed, then it uses the next one. With this trick, it should be possible, to speed up the process, because without Loop Blocking, we would have much more cache misses.

UPDATE:
We were now able to execute the code with all of the above ideas. But nothing really helped to improve performance. The peak performance rate was ~3.17 GFLOP/s .With Loop Blocking, the rate was with ~1.74 GFLOP/s even worse.
Then we tried it again with the `#pragma omp parallel for` for the outer loop, included the `omp.h` file and also the compiler and linker flags `-qopenmp` and `-parallel`. Before running, we also exported `OMP_NUM_THREADS=4`, because there are 4 Hyperthreads per Core. When we then execute our code, we get to a peak performance of ~4.6 GFLOP/s.