Assignment 8

Task 1

Code snippet 1

```
a[0] = 0;
#pragma omp parallel for
for (i=1; i<N; i++) {
    a[i] = 2.0*i*(i-1);
    b[i] = a[i] - a[i-1];
}
```

the for loop is splitted in equal sizes for every thread, so a[i-1] could be computed by another thread and the value of this cell might
not exist at this point. -> not a good solution
Because of this dependency to the cell before (a[i-1]), this code is not that easy to parallelize. Probably the easiest way to do it, is
to generate the values of array a seperately, cause it doesn't depend on anything else than i:

```
a[0] = 0;
#pragma omp parallel for
for (i=1; i<N; i++) {
    a[i] = 2.0*i*(i-1);
}
#pragma omp for
for (i=1; i<N; i++) {
    b[i] = a[i] - a[i-1];
}
```

Code snippet 2

```
a[0] = 0;
#pragma omp parallel
{
    #pragma omp for nowait
    for (i=1; i<N; i++) {
        a[i] = 3.0*i*(i+1);
    }
    #pragma omp for
    for (i=1; i<N; i++) {
        b[i] = a[i] - a[i-1];
    }
}
```

This is actually almost the solution i gave for the first snippet, but with `nowait`. This causes kind of race conditions, because now
the values in array `a` could be already initialized when they needed, but for sure, because the threads don't wait till a is completely calculated.

So the solution for this snippet would be the same as for the first snippet.

Code snippet 3

```
#pragma omp parallel for
for (i=1; i<N; i++) {
    x = sqrt(b[i]) - 1;
    a[i] = x*x + 2*x + 1;
}
```

x and b are bot shared variables. So here we have race conditions for sure, because x is set from every thread in every iteration, so the values of a could differ every time the program is executed. So the solution for this snipped might be to change x to a local variable, if it is not needed outside of the for loop.

Code snippet 4

```
f = 2;
#pragma omp parallel for private(f,x)
for (i=1; i<N; i++) {
    x = f * b[i];
    a[i] = x - 7;
}
a[0] = x;
```

This is almost a good implementation, cause compared to the snippet before, f and x are private now, so there are local copies for every thread and the values won't differ for different executions anymore. Although it's unnecessary to copy `f` cause it is a constant. No dependencies on a cell before or anything like in the first two code snippets.

But the last line: `a[0] = x` this actually causes a race condition again, cause depending on which thread executes last, it's set to the x value of that thread. It is difficult to say how to fix this code, because i don't know what we actually want in a[0]. If we want it to be the last value and to be always the same value, we could set it to a[0] = a[N - 1]

Code snippet 5

```
sum = 0;
#pragma omp parallel for
for (i=1; i<N; i++) {
    sum = sum + b[i];
}
```

This code causes again race conditions, because sum is a shared variable and it highly affects the output of every thread/iteration. Luckily this is a good case for using reduction, so it will be easy to fix:

```
sum = 0;
#pragma omp parallel for reduction (+:sum)
for (i=1; i<N; i++) {
    sum = sum + b[i];
}
```

Task 2

Compilation

analysis.c has been compiled with gcc 8.2.0 and the following command:
`gcc -Wall -Werror -std=gnu11 -O2 -ftree-vectorize -fopt-info-vec-all analysis.c -o analysis.out`

Findings

Can you find any information pointint to successful or unsuccessful vectorization?

The compiler prints the result for each vectorization attempt it makes (so there is a section for every loop).
If a loop was vectorized successfull, the compiler outputs `NOTE: LOOP VECTORIZED` at the end of an output block.
If the vectorization was unsuccessfull, the compiler prints a message with the reason why the loop could not be vectorized, e.g.:

```
analysis.c:22:9: note: not vectorized: not suitable for gather load _5 = a[_4];
analysis.c:22:9: note: bad data references.
```

and

```
analysis.c:27:9: note: not vectorized: loop contains function calls or data references that
cannot be analyzed
```

What information about dependence analysis can you find in the output?

The compiler marks which kind of anylysis is performed. For example, there are output lines like

```
analysis.c:18:9: note: === vect_analyze_data_ref_accesses ===
```

which indicate that the data access patterns in the loop are analyzed. It also performs dependency analysis (`analysis.c:18:9: note: === vect_analyze_data_ref_dependences ===`) and cycle analysis (`=== vect_analyze_scalar_cycles ===`).

Information about the dependency distance and direction can also be found:

```
analysis.c:18:9: note: dependence distance  = 4.
analysis.c:18:9: note: dependence distance negative.
```

Does the compiler perform any analysis beyond checking for dependencies and semantic correctness?

The compiler analyses whether SLP can be performed. According to this source SLP stands for "Superword-level parallelism, which means that similar independent instructions get merged into vector insturctions (`=== vect_analyze_slp ===`).
It also seems like the compiler checks how data is alligned (`=== vect_analyze_data_refs_alignment ===`).

Useful resources

Source that explains SLP
GCCs source code with useful comments

Task 3

Snippet 1

This code snippit is embarrassingly parallel.

Original variant

```
for(int i = 0; i < 1024; i++) {
    y[i] = x[i];
}
```

Manually parallelized

```
#pragma omp parallel
    {
#pragma omp for schedule(static)
        for(int i = 0; i < 1024; i++) {
            y[i] = x[i];
        }
    }
```

Manually it is pretty easy to parallelize with OpenMP and I am certain that the compiler is able to parallelize this, I do not see any circumstance that it would not be able to.

Snippet 2

Original variant

```
for(int i = 4; i <= N; i += 7) {
    for(int j = 0; j <= N; j += 3) {
        A[i] = 0;
    }
}
```

Half normalized variant

After (half) the normalization of the loops both of their iterator variables start at 1. As an additional condition the variables get incremented by 1 with each iteration. The index of the accessed array gets adjusted, so that in each iteration the same element of the array gets accessed as in the original.

```
for(int i = 0; i <= (N - 4) / 7; ++i) {
    for(int j = 0; j <= N / 3; ++j) {
        A[i + 4] = 0;
    }
}
```

Normalized variant

Unfortunately I did not come up with a fully normalized variant, since there are still nested loops. I was not able to get it into one single for loop.

The normalized would look something like this, where the condition in the for loop is the multiplication of the conditions of `i` and `j`. Furthermore the index `x` should be increment by 1 with every `N/3`-th iteration in this variant.

```
for(int i = 0; i <= (N * N - 4 * N) / 21; ++i) {
    A[x] = 0;
}
```

Snippet 3

For the third code snippet it is questioned if it holds any dependencies. And it does hold dependencies indeed.

```
L1  for(int i = 1; i < N; i++) {
L2      for(int j = 1; j < M; j++) {
L3          for(int k = 1; k < L; k++) {
S1              a[i+1][j][k-1] = a[i][j][k] + 5;
            }
        }
    }
```

Formally written the dependency is described as the following: $S_1[1,1,2]\ \delta\ S_1[2,1,1]$

This means that S_1 is true dependence on itself, in detail: e.g. the element a[2,1,1] is dependent on the element a[1,1,2]. Accordingly the distance vector corresponds to: (1, 0, -1); and the direction vector to: (<, =, >).

The distance vector is calculated by the offset of the indexes of the dependent elements and the direction vector by the signs of the elements of the distance vector.