Yann KERVERDO

Task 2 Report

Here is my report on Task 2. All times are in milliseconds.

To begin, I used GCC version 10.1.0. Here are the finals results:

Table 1: GCC Benchmark - gcc 10.1.0

	O0	**O2-fno-tree-vectorize**	**O3-ftree-vectorize**	**Ofast-ftree-vectorize**
1024x1024	6765	2589	2790	2418
2048x2048	81177	27204	27033	38399
4096x4096	974268	429919	261521	542840

The Gaussian elimination part is the dominant one (transformation into an upper triangular matrix) and has a complexity of $O(n^3)$. The pivot search is $O(n^2)$. In our case, since nrhs = n, the back substitution is also $O(n^3)$. Therefore, the total complexity is $O(n^3)$, which explains why the computation time increases so much when the matrix size is doubled.

With the *-O2 -fno-tree-vectorize* option, the code is reorganized to speed up loops and remove unnecessary instructions. This option enables most optimizations, but it does not vectorize the operations. We can see that the execution time is greatly reduced, although the improvement is smaller for matrices of size 4096×4096

With the original code, the execution times for *-O3* were:

Table 2 : -O3 with the original code

O3-ftree-vectorize	4434	50194	640890

We notice that the computation time is longer compared to *-O2* without vectorization. In my opinion, this could be because the compiler vectorizes automatically, but it doesn’t know that the data is aligned or that there is no aliasing. It therefore has to generate more cautious SIMD instructions to handle unaligned memory access and potential pointer dependencies. These precautions add memory overhead and reduce cache efficiency.

I generated the vectorization report using the *-fopt-info-vec-all* option, and here is what the compiler did:

```
for (int j = 0; j < n; j++) {
    double temp = matrix[row1 * n + j];
    matrix[row1 * n + j] = matrix[row2 * n + j];
    matrix[row2 * n + j] = temp;
}
```

This loop was vectorized by GCC using 16-byte vectors. Each iteration of j accesses contiguous and independent elements in memory. There are no dependencies between iterations, so the compiler can execute the operations simultaneously using SIMD

```
for (int k=i+1;k<n;k++) {
    a[j*n+k] -= factor*a[i*n+k];
}
for (int k=0;k<nrhs;k++) {
    b[j*nrhs+k] -= factor*b[i*nrhs+k];
}
```

These loops were vectorized by GCC using 16-byte vectors. Each iteration of k accesses contiguous and independent elements in memory. There are no dependencies between iterations, so the compiler can execute the operations simultaneously using SIMD

```
for (int k=j+1;k<n;k++) {
 b[j*nrhs+i] -= a[j*n+k]*b[k*nrhs+i];
}
```

Not vectorized by GCC: WAW (Write after Write), also called an output dependency. Each iteration writes to the same variable *b[j*nrhs + i]* while reading multiple values to compute sum. In general, this prevents vectorization. Then I applied the following changes:

```
#pragma omp simd
for (int k=j+1;k<n;k++) {
    sum += a[j*n+k]*b[k*nrhs+i];
}
b[j*nrhs+i] -= sum;
```

Normally, it can be vectorized when the compiler recognizes it as a sum reduction. However, a new problem arises: the column-wise access is not contiguous in memory, so vectorization is again impossible. Fixing this would require a complete restructuring of the code to use column-major order, but then the triangularization would lose its vectorization on b. When *nrhs* is small, this could greatly improve execution speed, but here *nrhs* = *n*, so the benefit would be limited. I still tried the following approach:

```
double sum = 0.0;
double tmp_col[n];
for (int k=j+1;k<n;k++){
    tmp_col[k] = b[k*nrhs+i];
}
#pragma omp simd
for (int k=j+1;k<n;k++) {
    sum += a[j*n+k]*tmp_col[k];
}
b[j*nrhs+i] -= sum;
```

Copying into a buffer first allows the following loop to be vectorized, but the copy itself cannot be vectorized. As a result, a lot of time is lost overall because we add a new non-vectorizable loop

```
for (int j = i+1; j < n; j++) {
    double val = fabs(a[j*n + i]);
    if (val > max_val) {
        max_val = val;
        swap_row = j;
    }
}
```

Conditional statements are allowed in a loop if they can be replaced by masks, but in this case, it is not possible. I tried this method:

```
double max_val = fabs(a[i*n + i]);
#pragma omp simd
for (int j = i+1; j < n; j++) {
    double val = fabs(a[j*n + i]);
    max_val = fmax(max_val, val);
}
for (int j=i+1; j < n; j++) {
    if (fabs(a[j*n + i]) == max_val) {
        swap_row = j;
        break;
    }
}
```

The *if* could be done in a separate sequential loop afterward, but it's unclear if this would actually save time. Another problem is the write-after-write and non-contiguous memory access, making vectorization too complicated

Finally, the main loop cannot be vectorized due to too many dependencies and nested loops. I was not able to make any other loops vectorizable.

However, I added several changes to improve vectorization and memory alignment:

- *restrict* on pointers a and b: This tells the compiler that no other variable points to the same memory. It reduces aliasing checks, allowing the compiler to reorder instructions and vectorize loops more freely, enabling more aggressive optimizations.
- *__builtin_assume_aligned* on a and b: This tells the compiler that the pointers are aligned to 32 bytes (suitable for AVX/AVX2 SIMD instructions). Previously, the compiler had to generate slower code compatible with unaligned memory accesses. Now, loops accessing a and b can use fast, aligned vector loads and stores, reducing micro-operations and cache/memory penalties. It is essential that the pointers are actually aligned in memory.
- *_mm_malloc* in main.c: This is used to allocate memory aligned to 32 bytes, which ensures that *__builtin_assume_aligned* is valid.
- *#pragma omp simd* for the k loops: This allows the compiler to treat these loops as SIMD loops and enable vectorization, even though the loops were already vectorized

The compiler vectorizes the same loops but we no longer lose time as before. This results in a version that runs as fast as *-O2* for the two smaller matrices and faster for the 4096×4096 matrix.

We notice that with the *-Ofast* option, execution is slower than with *-O3* and *-O2*. According to the report, the same loops are vectorized. However, *-Ofast* enables *-ffast-math*, allowing optimizations that are not valid for all standard-compliant programs, which seems to slow down the execution.

Here are the results using GCC 11.4.0:

Table 3: GCC Benchmark - gcc 11.4.0

	O0	**O2-fno-tree-vectorize**	**O3-ftree-vectorize**	**Ofast-ftree-vectorize**
1024x1024	7768	2634	3285	3539
2048x2048	59127	27081	51579	51899
4096x4096	973747	471019	753977	687900

We observe results quite similar to version 10.1.0 for the *-O0* and *-O2* options without vectorization. However, the options with vectorization are much less efficient than in the previous version and are even slower than *-O2*. I'm not sure how to explain this, since according to the reports, the same loops are optimized and this version is supposed to be more recent. It's possible that version 10.1.0 is simply better suited to the CPU being used.

Here are the results using GCC 8.4.0:

Table 4: GCC Benchmark - gcc 8.4.0

	O0	**O2-fno-tree-vectorize**	**O3-ftree-vectorize**	**Ofast-ftree-vectorize**
1024x1024	8301	3375	2954	3888
2048x2048	147702	43184	34311	56351
4096x4096	1086809	709211	772146	639618

Overall, we observe results that are much slower than with version 10.1.0. This is likely due to the age of the compiler. This time, there is a slight improvement when vectorizing with *-O3* except for the largest matrices. The *-Ofast* option allows faster execution on these large matrices but slows down performance on the medium and small ones. The same loops have been vectorized.