LU DECOMPOSITION OF A MATRIX

The following code was taken from:
https://www.tutorialspoint.com/cplusplus-program-to-perform-lu-decomposition-of-any-matrix

The LU decomposition of a matrix produces a matrix as a product of its lower triangular matrix and an upper triangular matrix. The LU in LU Decomposition of a matrix stands for Lower Upper.
LU decomposition on a square matrix "A" of size n x n.

The code and its functionality:

The code was modified to fit a matrix with random numbers.

- **Headers:** The code includes necessary C++ libraries for input/output, vector handling, random number generation, and timing measurements. We use vectors since this code deals with a large input size.
- **The LU Decomposition Function:** The LUdecomposition function decomposes the square matrix 'A' into two triangular matrices, 'L' (lower triangular) and 'U' (upper triangular). The process is as follows:
    - Uses for loops to iterate through the elements of the input matrices.
    - **The Lower Triangular Matrix L:** In this loop, we get the lower triangular matrix. We check if j<i, and if so, L[j][i]=0. This ensures that the elements below the main diagonal of 'L' are zeroes. Else, L[j][i] is calculated based on the original matrix "A" and subtracts the product of the corresponding elements from "L" and "U".
    - **The Upper Triangular matrix U :** Here, if j<i, then U[i][j]=0, to ensure that the elements above the main diagonal are zeroes. Else, if j==i, then U[i][j] is set to 1 since the diagonal elements are always 1. Else, U[i][j] is calculated based on the original matrix "A" and subtracts the product of corresponding elements from "L" and "U".

  So, this function decomposes the input matrix into a lower triangle matrix and upper triangle matrix, while ensuring that some elements are appropriately set to 0 or 1 as part of the decomposition process.

- **The main function:** Defines the input matrix A (n x n) size. It initializes vector matrices A, L, and U. Here, it generates random values for matrix A since the matrix consists of large input data. It also measures the execution time by recording the start and end times before and after the LU decomposition function. It also prints the time of execution.

In summary, the code generates a random square matrix, performs LU decomposition, and measures the execution time.

The program's flow:

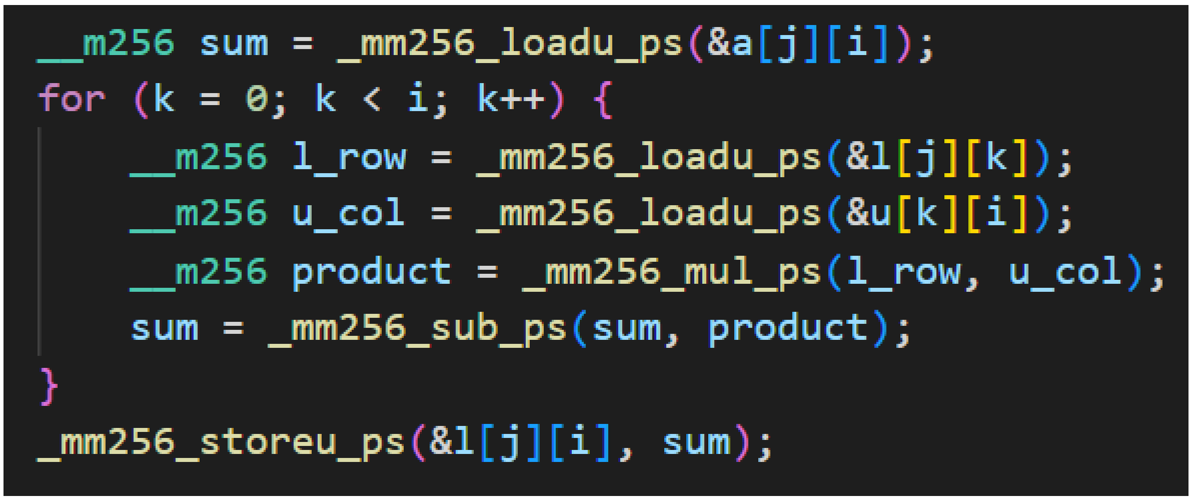
1. The program includes necessary libraries and declares the ‘LUdecomposition’ function.
2. The Main Function initializes variables, generates a random square matrix 'A' calls the LUdecomposition function,
3. The function decomposes ‘A'' into lower (‘L’) and upper (‘U’) triangular matrices,
4. Then the main function measures the execution time and displays the execution time.

Therefore the program starts in the main function, initializes data, forms an input data matrix of size n using random numbers, performs LU decomposition, and measures the execution time of the LU decomposition.

In the optimized code, Single Instruction, Multiple Data (SIMD) instructions, and intrinsic functions are used to accelerate the LU decomposition. SIMD allows for parallel processing of data, which can significantly improve the performance of matrix operations.

SIMD is used to accelerate LU decomposition in the following ways:

1. **SIMD Usage** - SIMD instructions to process multiple data elements in parallel, specifically leveraging the AVX2 instruction set. The code includes the ‘<immintrin.h>’ header, which provides access to SIMD instructions. It also uses the '#pragma GCC target("avx2")' directive to target the AVX2 instruction set, an advanced SIMD technology.
2. **Data Loading and Storage** - Utilizes 256-bit SIMD registers (__m256) to load and store eight single-precision floating-point values at once.
3. **Matrix Operations** - In the non-optimized version, we perform matrix multiplication element-wise in a loop. In the optimized version, we load eight elements from the ‘l’ and ‘u’ matrices and perform multiplication and subtraction operations in parallel. Matrix multiplication is accelerated by performing multiplication and subtraction operations in parallel using _mm256_mul_ps and _mm256_sub_ps instructions. The division is optimized by broadcasting the divisor value to a SIMD register and dividing all elements simultaneously with _mm256_div_ps.



```
__m256 sum = _mm256_loadu_ps(&a[j][i]);
for (k = 0; k < i; k++) {
    __m256 l_row = _mm256_loadu_ps(&l[j][k]);
    __m256 u_col = _mm256_loadu_ps(&u[k][i]);
    __m256 product = _mm256_mul_ps(l_row, u_col);
    sum = _mm256_sub_ps(sum, product);
}
_mm256_storeu_ps(&l[j][i], sum);
```

4. **Loop Unrolling** - Manually unrolls loops, allowing for multiple data elements to be processed within each loop iteration, minimizing loop overhead, which means that instead of processing one element at a time, you process multiple elements simultaneously within each loop iteration.
5. **Parallelism** - Achieves parallelism by processing multiple data elements concurrently, reducing the number of iterations required for LU Decomposition.
6. **Final Acceleration** - The combination of these SIMD optimizations accelerates LU decomposition, resulting in a substantial speedup compared to non-SIMD implementations.

Compilation steps and Flags:

Compile the provided SIMD-accelerated code for LU decomposition, using a C++ compiler that supports AVX2 and SIMD instructions. Here are the compilation steps and flags needed for the code:
Compiler: I used a C++ compiler that supports AVX2, such as GCC with VSCode.
Compile Command: Used the following compilation command:

```
g++ -O3 -mavx2 -o optimized optimized.cpp
```

Flags used:

- **'-O3'**: This flag enables aggressive optimization. It's crucial for achieving the best performance.
- **'-mavx2'**: This flag enables AVX2 instructions, which are essential for the SIMD acceleration in the code.
- **'-o optimized'**: This flag specifies the output executable name.

Run the compiled program - Did it using the following command:

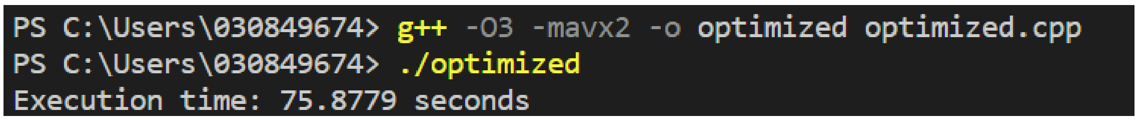
```
./optimized
```

This executes the LU decomposition code with SIMD acceleration.

<u>Proof of achieved speedup:</u>

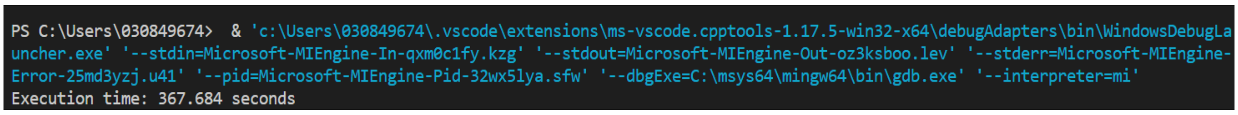
The SIMD-optimized code demonstrates a significant **average speedup of 4.63x** in LU Decomposition compared to the non-optimized version. Execution time is reduced, and matrix calculations are executed with higher efficiency, which was way better than expected.

Execution time for the optimized LU Decomposition code:



```
PS C:\Users\030849674> g++ -O3 -mavx2 -o optimized optimized.cpp
PS C:\Users\030849674> ./optimized
Execution time: 75.8779 seconds
```

Execution time for the non-optimized LU Decomposition code:



```
PS C:\Users\030849674>  & 'c:\Users\030849674\.vscode\extensions\ms-vscode.cpptools-1.17.5-win32-x64\debugAdapters\bin\WindowsDebugLa
uncher.exe' '--stdin=Microsoft-MIEngine-In-qxm0c1fy.kzg' '--stdout=Microsoft-MIEngine-Out-oz3ksboo.lev' '--stderr=Microsoft-MIEngine-
Error-25md3yzj.u41' '--pid=Microsoft-MIEngine-Pid-32wx5lya.sfw' '--dbgExe=C:\msys64\mingw64\bin\gdb.exe' '--interpreter=mi'
Execution time: 367.684 seconds
```