Homework Assignment 3: The LU factorization

Scientific Software / Technisch Wetenschappelijke Software

Mari Hove Gusdal

Practical info

Machine name: **Virton**

Other sources Did you use any sources other than the course material? Cite them here. Did you use generative AI? If so, how did you use it?

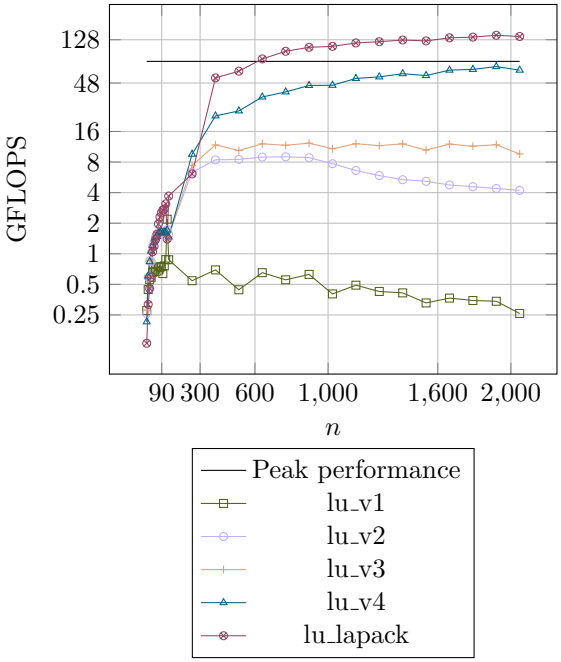
> I did use AI for some latex help(in regards to proper visualisations of the equations). And for some further explenation on the innerworkings of the LAPACK functions sgetrf and dgetrf.

Questions

Q1: What is the computational cost of the LU factorization?

> The computational cost of LU factorization is $flops = 2/3n^3 + O(n^2)$. The latter part coming from pivot-search, but will be dominated by the first part for large n.

F1: Plot the number of floating point operations per second in function of N for `lu_v1`, `lu_v2`, `lu_v3`, `lu_v4`, and `lu_lapack`. For the improvements '`lu_v2`→`lu_v3`', '`lu_v3`→`lu_v4`' and '`lu_v4`→`lu_lapack`', explain in one or two sentences how the performance changes and why. Keep it surface-level. If you suspect any anomalies in the performance plot, discuss them as well.



lu_v2 → lu_v3: We introduce blocking, i.e., factor the matrix in panels and update the trailing submatrix in block form, improving temporal locality. This lets us reuse data in L1/L2 cache while working on each block instead of repeatedly fetching it from RAM, so performance improves for sufficiently large matrices.

lu_v3 → lu_v4: Introduced BLAS functions (GEMM and TRSM). These are highly optimized functions for matrix multiplication and triangle solve, including multi-level blocking, packing for a contiguous layout and also introduces multithreading possibilities, shifting the work into Level 3 BLAS kernels.

lu_v4 → lu_lapack: Introduced LAPACK fortran routines: sgetrf_ and dgetrf_. These are LU specific routines that automatically allow for multithreaded BLAS, and spec-optimized methods. Main performance boost comes form no longer relying on lu_v2 for panel factorization, but instead using LAPACKs highly tuned methods with block-size chosen based on the machine's architecture.

Mark: The main benefit of blocking in lu_v3 and lu_v4 methods can be seen after $n \geq$ block_size (in my case: 128). Before this, lu_v2 will do the majority of the work, showing little difference between these methods for smaller matrices. Larger matrices have flops handled by level 3 BLAS operations, shifting LU factorisation from memory-bound (waiting for RAM) to compute-bound (doing FLOPS).

I included a line showing the peak performance of my machine (78.4 Gflops), but as can be seen: lu_lapack outperforms this. The peak performance is the single-core performance, however LAPACK methods allow for parallelization, meaning multi-threading of computations of sub parts of the matrices. The results in the graph were all executed with `OMP_NUM_THREADS=4`.

Q2: What change did you make to `lu_v1` to make it more efficient? Why does this change improve the performance?

To improve the performance of `lu_v1`, I swapped the order of the `i` and `k` loops. Since the matrices in `twsmatrix` are stored in column-major format, elements within a column lie next to each other in memory. Accessing memory in this order is much more efficient, because values that are adjacent in memory are likely to be loaded into the same cache line.

In the original version, the algorithm iterated over rows first, which meant jumping across memory with a large stride. This causes poor spatial locality: the CPU repeatedly reloads cache lines, and useful data is evicted before it can be reused. By iterating over columns first, the updated version accesses memory contiguously, improving cache utilisation and reducing cache misses. This comes from improved spatial locality, reducing TLB misses.

Contiguous access patterns also enable better vectorisation. The compiler or hardware can apply SIMD instructions only when data lies sequentially in memory. This allows more floating-point operations per cycle and therefore higher performance. Contiguous access also allows for prefetching, letting the CPU load predicted data into the cache early.

In total the improvements are visible in the plot. Lu_v1 peaks at `n=120`, before falling for every larger n due to poor memory locality. Meanwhile lu_v2 peaks for `n=768`, showcasing the reduced cache and TLB preassure, but still a limit in memory bandwidth.

Q3: Write down the formulas you derived for the blocked backpropagation to solve triangular systems of equations.

Given: $L \times X = B => X = L^{-1} \times B$

we solve the system:

$$\begin{bmatrix} L_{00}X_{00} \\ L_{10}X_{00} + L_{11}X_{10} \end{bmatrix} = \begin{bmatrix} B_{00} \\ B_{10} \end{bmatrix}. \tag{1}$$

From this, we derive:

$$\begin{aligned} X_{00} &= L_{00}^{-1} B_{00}, \\ L_{11} X_{10} &= B_{10} - L_{10} L_{00}^{-1} B_{00}, \\ X_{10} &= L_{11}^{-1} B_{10} - L_{11}^{-1} L_{10} L_{00}^{-1} B_{00}. \end{aligned} \tag{2}$$

Resulting in:

$$\begin{bmatrix} X_{00} \\ X_{10} \end{bmatrix} = \begin{bmatrix} L_{00}^{-1} B_{00} \\ L_{11}^{-1} B_{10} - L_{11}^{-1} L_{10} L_{00}^{-1} B_{00} \end{bmatrix} \tag{3}$$

Generalizing, the solution for each component is:

$$x_i = L_{ii}^{-1} \left(b_i - \sum_{k=0}^{i-1} L_{ik} x_k \right). \tag{4}$$

Q4: If $n = 512$ and $n_b = 64$, then approximately 10% of the floating point operations in the blocked LU factorization is spent on the LU factorization of the small blocks. What percentage of the runtime of `lu_v4` is spent of the small blocks if those parameters are used? Explain.

The remaining 90% of the floating point operations are performed by Level–3 BLAS routines (`trsm`/`gemm`), which run at a significantly higher GFLOP/s rate than the unblocked LU in the small panels.

$$Symbols: F = number of flops, R = floprate[gflop/s], t = time[s]. \tag{5}$$

$$t_s = \frac{F_s}{R_s}, \qquad t_\ell = \frac{F_\ell}{R_\ell},$$

so the fraction of the total runtime spent in the small blocks is

$$\text{fraction} = \frac{t_s}{t_s + t_\ell} = \frac{F_s/R_s}{F_s/R_s + F_\ell/R_\ell}.$$

Since $F_s = 0.1F$ and $F_\ell = 0.9F$, and assuming $R_\ell \approx 6R_s$ (a conservative estimate for BLAS–3 vs. unblocked LU),

$$\text{fraction} = \frac{0.1}{0.1 + 0.9/6} \approx 0.40.$$

Despite the small blocks only taking up about 10% of the FLOPS, their much lower FLOP rate compared to BLAS means they will take up easily atleast 40% of the runtime. Making the choice of blocksize crucial for good performance.

Q5: Describe your testing methodology. What did you test and why?

I first wanted to test that my implementations actually returned the correct output, as this is the essential part of the functions. For this I made the assumption that the already implemented naive methods from the course team, is correct. From this correct matrix, I then checked the result of each of the lu-functions against the result of `lu_v1`. Returning false if the matrix was different, or true for the same result.

I made a helper function to calculate the residual of the LU factorization, to see how close the reconstructed matrix is to the original matrix. This was done by calculating $R = PA0 - LU$, and then calculating the norm of R divided by the norm of $A0$. If this value was below a certain threshold (depending on float or double matrix), I considered the test to be passed. This was to check that the LU factorisation, actually computes back into the original matrix.

I also wanted to test the secureness of my implementation. Checking if my implementation handles incorrect sized matrices, double vs ints vs floats. In addition, I also wanted to test how it handles non-square matrices. And again check if these resulted in the same output for each method.

Finally I checked if the functions could handle zero-sized matrices.

Under the assumption that all of my tests were implemented correctly, every function passed every test.

Now I did have to make 2 small changes in order to get them all to pass.

The first was to add a `if (!ipiv.size() == min_val){throw new exception}`, despite including an assertion below this. This was to be able to catch the error, as assertions automatically throw `std::abort()`, and are not catch-able. For performance, I would remove this part, and keep only the assertion. However to be able to run every function without stop, I had to include it.

The 2nd change I made was to add a `if (m==0 || n==0) return;` to `lu_lapack`. This is due to `dgetrf` and `sgetrf` not being able to handle a `lda` of 0.

Q6: What changes did you make to support `float` matrices?

In order to support both float matricies and double matricies, i included a check for wether the Type T was of type float or double. For lu_v4: If double i mapped the matrix multiplication to cblas_dgemm and triangular solve cblas_dtrsm. Or if float i mapped the MM to cblas_sgemm and triangular solve to cblas_strsm. For lu_lapack: If float I mapped to the function sgetrf_, or if double i mapped to the function dgetrf_.

These functions have been specifically optimized to handle double and float matricies respectively, and will therefore give the best performance as possible regardless of T type.

I additionally made a small change in my tests to handle both float and double matricies. Floats have a lower precision degree than doubles, and i could therefore not use the same tolarance for error in my computation of lu_residual. Floats have atmost 1e-7 rounding errors, meanwhile doubles have 1e-15. Therefore i mapped the tolarance to 1e-5 if floats, and 1e-12 for doubles.