Masterpraktikum Scientific Computing – High Performance Computing

Gaurav Kukreja, Evangelos Drossos

Exercise Sheet 1

Exercise 3

a) The algorithm transforms the matrix of coefficients to reduced row echelon form.

It starts by first dividing the first row of the matrix by the value of the leading (non-zero) coefficient, so that the leading coefficient's value is set to equal one. Then, it adds a scalar multiple of the first row to all subsequent rows, so that the column of the leading coefficient contains only zeroes in all subsequent rows. It then proceeds by doing the same thing for all other rows in the matrix. The leading coefficient that gets selected for each row is an element on the main diagonal of the matrix. In the end, the matrix of coefficients A is transformed into a right triangular matrix with the additional property of its main diagonal containing only the value of 1 (reduced row echelon form).

Starting Situation: $Ax = b$ → Final System

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \cdot X = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} 1 & a'_{01} & a'_{02} \\ 0 & 1 & a'_{12} \\ 0 & 0 & 1 \end{pmatrix} \cdot X = \begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \end{pmatrix}$$

Finally, using backwards substitution, the solutions vector X is calculated:

$x_2 = b'_2$

$x_1 = b'_1 - a'_{12} * x_2$

$x_0 = b'_0 - a'_{02} * x_2 - a'_{01} * x_1$

b) Vectorization does not provide any significant benefits in the case of rank n=3 linear systems. The low number of loop iterations ensures that any speedup from vectorization remains marginal. This is especially true of inner loops, which are the main candidates for vectorization (though data dependencies pose some additional problems). The starting value for the iterator of the innermost loops is a function of the iterator i of the outermost loop (e.g. j=i+1 as a starting condition). This means that inner loops rarely need to perform more than 2 iterations. As a result, vectorization only leads to marginal improvements.

c) A number of optimizations have been made to achieve vectorization wherever possible. The loops for backwards substitution have been interchanged to avoid any

data dependencies. The for-loop that was responsible for the operations between rows has been split in two, and all vectorization candidates have been vectorized. Some minor changes have been made at the first step where the selected row gets divided by the leading non-zero coefficient (the element on the diagonal).

The matrix is now assumed to be stored in memory in a way that column elements can be accessed consecutively (as opposed to consecutive storage of row elements in the original implementation). This allows for easier memory access, especially for the backwards substitution step. There is one exception to this. The first step – division of elements in the currently selected line by the respective element in the diagonal – now has an inefficient memory access pattern.

This can be part of the reason why the 'optimized for vectorization' version of the code achieves lower performance than the original code. As already explained, the benefits in performance from vectorization are only marginal in the case of a system of rank n=3.

The #pragma ivdep instruction has been used in three occasions in the code to prevent the compiler from checking possible array dependencies that cause no problem. This is ensured through the conditions of the respective for-loops. Through some testing, and even though the returned performance values were somewhat unstable, it seems to be the case that force-vectorizing only the two loops that are responsible for row additions (which came into existence after the original loop was split into two) leads to better performance compared to the vectorization of all loops. This would mean that the attempted vectorization of the backwards substitution loop leads to lower performance. However, since the primary task of this exercise is vectorization, we decided to leave all #pragma ivdep instructions in the handed-in vector-optimized code.

Following performance metrics can be obtained through a sample run of each version of the code (rounded values for MFLOPS):

	Original	**Vector-optimized**
MFLOPS (timeofday)	344	269
Time elapsed	0.000346	0.000430

Following formula has been used to calculate FLOPs number:

$$flops = \sum_{i=1}^{n-1} i + n + \sum_{i=1}^{n-1} i * \left(\sum_{i=1}^{n} 2 * i + 2 \right) + \sum_{i=1}^{n-1} 2 * i$$

Where:

$$\sum_{i=i_0}^{n} i = \frac{(n + i_0)}{2} * (n - i_0 + 1)$$

This leads to the following formula:

$$flops = 0.5n^4 + 2n^2 - 1.5n$$