Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science

# Block MLDU

*Accelerate Block MLDU*

Lucas Bekker
0657761

Supervisors:

dr. J.M.L. Joseph Maubach

v1.0

Eindhoven, December 2018

# Contents

# List of Acronyms

**TU/e** Technische Universiteit Eindhoven

**LU** contraction of the "Lower" and "Upper" matrices

**LDU** contraction of the "Lower", "Diagonal" and "Upper" matrices

**MLDU** contraction of the "Matrix", "Lower", "Diagonal" and "Upper" matrices

**COO** coordinate list

**CCS** compressed column storage

**VRAM** video random access memory

**RAM** random access memory

# Chapter 1

# Introduction

The goal of this Capita Selecta is to investigate the possibilities for a fast MATLAB based implementation of the Block MLDU algorithm.

High performance solvers are usually written in languages like C/C++ because they are compiled, avoiding the sometimes costly step of the interpreter. Languages like MATLAB are dynamically typed and interpreter based, which make them great for prototyping purposes.

The Block MLDU algorithm shows great possibilities, but the lack of a high performance implementation makes it prohibitive to use for large matrices. This is a problem because research projects sometimes encounter very large matrices that need to be solved with the Block MLDU algorithm.

This project tries to alleviate the problem by investigating the performance bottlenecks of a MATLAB based implementation and provide workarounds for them. This should result in a much faster implementation without requiring the extra work associated with a full C/C++ version.

The biggest advantage of this workflow is that the time required to implement a workaround for a bottleneck will be much lower in MATLAB than in C/C++, making it easier to try different approaches to the problems. The lessons learned at this stage will still be useful when the code does finally get ported to C/C++.

The biggest disadvantage to sticking to MATLAB is that the interpreter will always provide some overhead compared to C/C++. The hope is that this overhead will be small compared to the other bottlenecks encountered.

## 1.1 The Block MLDU algorithm

The Block MLDU algorithm is very comparable to the well known LU decomposition. The main difference between the Block MLDU algorithm and LU decomposition is the "block" nature of the splitting.

### 1.1.1 Example

A simple example will be provided to highlight the differences between LU and Block MLDU. Let matrix $A$ be $4 \times 4$:

$$A = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 2 & 1 & -1 & 0 \\ 0 & -1 & 0 & 1 \\ -2 & 0 & 1 & 3 \end{bmatrix} \qquad s = \begin{bmatrix} 2 & 1 & 1 \end{bmatrix}$$

As stated by row $s$, matrix $A$ will first be split using a $2 \times 2$ block, followed by two $1 \times 1$ blocks.

**Step one:**

$$\left[\begin{array}{cc|cc} D1 & D1 & U1 & U1 \\ D1 & D1 & U1 & U1 \\ \hline L1 & L1 & M1 & M1 \\ L1 & L1 & M1 & M1 \end{array}\right] \qquad L1 = \begin{bmatrix} 0 & -1 \\ -2 & 0 \end{bmatrix} \qquad D1 = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \qquad U1 = \begin{bmatrix} 1 & 0 \\ -1 & 0 \end{bmatrix}$$

$$Schur1 = L1D1^{-1}U1 = \begin{bmatrix} 3 & 0 \\ -2 & 0 \end{bmatrix} \qquad A - Schur1 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 2 & 1 & -1 & 0 \\ 0 & -1 & \mathbf{-3} & \mathbf{1} \\ -2 & 0 & \mathbf{3} & \mathbf{3} \end{bmatrix}$$

**Step two:**

$$\left[\begin{array}{cc|c|c} D1 & D1 & U1 & U1 \\ D1 & D1 & U1 & U1 \\ \hline L1 & L1 & D2 & U2 \\ \hline L1 & L1 & L2 & M2 \end{array}\right] \qquad L2 = \begin{bmatrix} 3 \end{bmatrix} \qquad D2 = \begin{bmatrix} -3 \end{bmatrix} \qquad U2 = \begin{bmatrix} 1 \end{bmatrix}$$

$$Schur2 = L2D2^{-1}U2 = \begin{bmatrix} -1 \end{bmatrix} \qquad A - Schur1 - Schur2 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 2 & 1 & -1 & 0 \\ 0 & -1 & -3 & 1 \\ -2 & 0 & 3 & \mathbf{4} \end{bmatrix}$$

**Step three:**

$$\left[\begin{array}{cc|c|c} D1 & D1 & U1 & U1 \\ D1 & D1 & U1 & U1 \\ \hline L1 & L1 & D2 & U2 \\ \hline L1 & L1 & L2 & D3 \end{array}\right]$$

**Result:**

$$L = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \mathbf{0} & \mathbf{-1} & 0 & 0 \\ \mathbf{-2} & \mathbf{0} & \mathbf{3} & 0 \end{bmatrix} \qquad D = \begin{bmatrix} \mathbf{1} & \mathbf{0} & 0 & 0 \\ \mathbf{2} & \mathbf{1} & 0 & 0 \\ 0 & 0 & \mathbf{-3} & 0 \\ 0 & 0 & 0 & \mathbf{4} \end{bmatrix} \qquad U = \begin{bmatrix} 0 & 0 & \mathbf{1} & \mathbf{0} \\ 0 & 0 & \mathbf{-1} & \mathbf{0} \\ 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A = (L + D)D^{-1}(D + U)$$

# Chapter 2

# Performance comparison

This report will use the MATLAB profiler to discuss the runtime performance of MATLAB code implementations of the Block MLDU algorithm. The runtime of a piece of code is dependent on both the efficiency of the code and the hardware on which it is executed. This means that the runtime performance results will most likely vary from computer to computer. In order to verify if the achieved results on any given computer are in line with expectations, one must also be able to compare the general MATLAB related performance of the computer.

MATLAB provides a means to achieve this comparison by way of the `bench()` function. It runs a micro-benchmark with a variety of "typical" MATLAB workloads, namely:

- LU decomposition
- FFT
- ODE with `ode45`
- Sparse
- 2D
- 3D

The most important benchmarks for this algorithm are LU decomposition, FFT and Sparse, because they touch on hardware performance elements that are similar to the the Block MLDU algorithm. More information can be found by running: `doc bench`

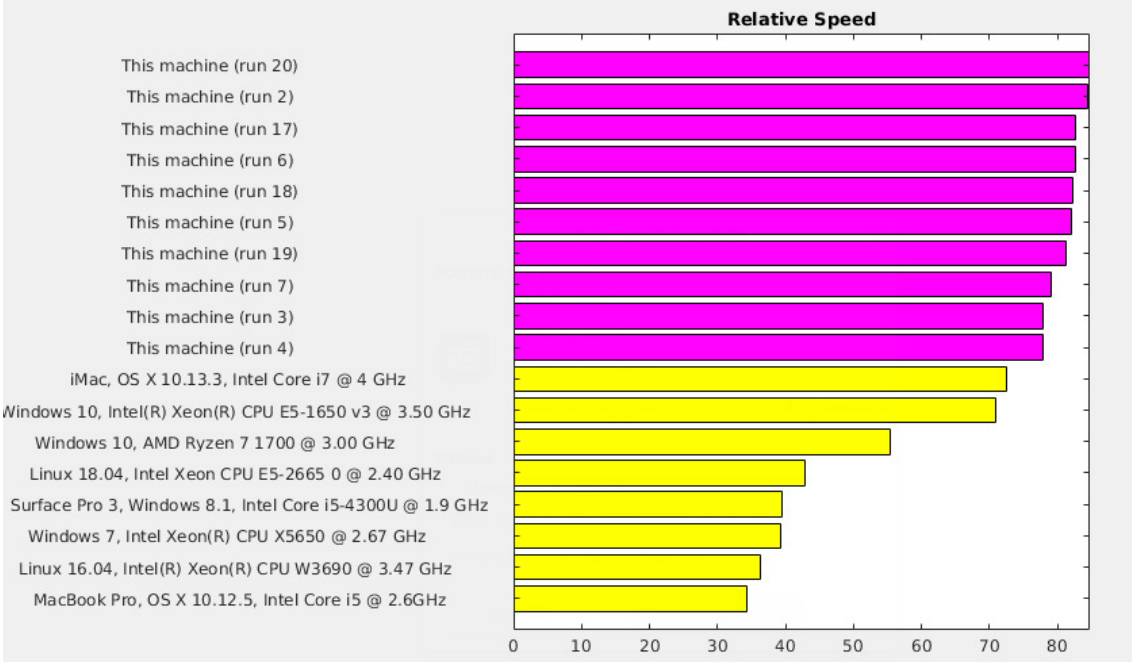
It should be noted that these benchmarks are far from an ideal way to compare different computers, but it can give a rough indication. It is recommended to run the benchmarks at least 10 times, as there can be some significant performance variance.

## 2.1 Test machine hardware specifications

A more detailed performance comparison between computers requires the hardware specifications. The test machine specifications are provided below:

- CPU: Intel Xeon E5-1650 v3 6 core 12 thread overclocked to 3.8 GHz
- Memory: 32 GB 4 channel DDR4 registered ECC running at 2133 MHz.
- Graphics: Two NVIDIA GTX Titan (Kepler) running in SLI

## 2.2 Test machine benchmark results




| Computer Type | LU | FFT | ODE | Sparse | 2-D | 3-D |
|---|---|---|---|---|---|---|
| This machine (run 20) | 0.0855 | 0.0741 | 0.0128 | 0.0763 | 0.4277 | 0.5287 |
| This machine (run 2) | 0.0878 | 0.0783 | 0.0146 | 0.0817 | 0.3929 | 0.4385 |
| This machine (run 17) | 0.0728 | 0.0927 | 0.0128 | 0.0777 | 0.3907 | 0.5657 |
| This machine (run 6) | 0.0824 | 0.0921 | 0.0128 | 0.0829 | 0.4147 | 0.4690 |
| This machine (run 18) | 0.0963 | 0.0815 | 0.0128 | 0.0789 | 0.4012 | 0.5139 |
| This machine (run 5) | 0.0894 | 0.0825 | 0.0127 | 0.0890 | 0.4353 | 0.4641 |
| This machine (run 19) | 0.0868 | 0.0988 | 0.0128 | 0.0763 | 0.4269 | 0.4743 |
| This machine (run 7) | 0.0949 | 0.0846 | 0.0133 | 0.0978 | 0.4410 | 0.4516 |
| This machine (run 3) | 0.0919 | 0.0975 | 0.0127 | 0.1077 | 0.4043 | 0.4269 |
| This machine (run 4) | 0.0938 | 0.0804 | 0.0133 | 0.1084 | 0.4356 | 0.4741 |
| iMac, OS X 10.13.3, Intel Core i7 @ 4 GHz | 0.0920 | 0.0871 | 0.0109 | 0.0969 | 0.8026 | 0.4441 |
| Windows 10, Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50 GHz | 0.0935 | 0.1420 | 0.0175 | 0.1088 | 0.3518 | 0.3011 |
| Windows 10, AMD Ryzen 7 1700 @ 3.00 GHz | 0.1827 | 0.1342 | 0.0188 | 0.2093 | 0.2641 | 0.2836 |
| Linux 18.04, Intel Xeon CPU E5-2665 0 @ 2.40 GHz | 0.1426 | 0.0951 | 0.0452 | 0.1786 | 0.8131 | 0.6906 |
| Surface Pro 3, Windows 8.1, Intel Core i5-4300U @ 1.9 GHz | 0.2381 | 0.2193 | 0.0241 | 0.1871 | 0.8136 | 0.5301 |
| Windows 7, Intel Xeon(R) CPU X5650 @ 2.67 GHz | 0.2119 | 0.1126 | 0.0290 | 0.1596 | 0.5868 | 1.6214 |
| Linux 16.04, Intel(R) Xeon(R) CPU W3690 @ 3.47 GHz | 0.1680 | 0.0885 | 0.0333 | 0.1417 | 1.3139 | 1.7373 |
| MacBook Pro, OS X 10.12.5, Intel Core i5 @ 2.6GHz | 0.1928 | 0.1737 | 0.0168 | 0.1361 | 1.9716 | 1.3791 |

Place the cursor near a computer name for system and version details. Before using this data to compare different versions of MATLAB, or to download an updated timing data file, see the help for the bench function by typing "help bench" at the MATLAB prompt.

As can bee seen, the performance of the test machine is generally very high. The second thing to note is that the performance varies quite significantly, sometimes as much as about 33%. That is why this benchmark was run 20 times, even though it only shows the results from the 10 best runs.

# Chapter 3

# Initial implementation

```
function [ L, D, U ] = MLDU_Simple( A, s )

% Creating indices structure
j = 0;
[m,n] = size(A);
i_j = struct('i',[],'j',[]);
index = struct('M',i_j,'L',i_j,'D',i_j,'U',i_j);

% Preallocate output matrices
L = zeros(m,n);
D = zeros(m,n);
U = zeros(m,n);

% Loop over the blocks
for i = 1:length(s)

    update_index_nested(i);

    L(index.L.i,index.L.j) = A(index.L.i,index.L.j);
    D(index.D.i,index.D.j) = A(index.D.i,index.D.j);
    U(index.U.i,index.U.j) = A(index.U.i,index.U.j);

    schur_complement =  A(index.L.i,index.L.j)* ...
                       (A(index.D.i,index.D.j)\ ...
                        A(index.U.i,index.U.j));

    A(index.M.i,index.M.j) = A(index.M.i,index.M.j) - schur_complement;

end

% Nested index function
function [ ] = update_index_nested(i)

    index.D.i = (1:s(i)) + j;
    index.D.j = index.D.i;
    index.L.i = (index.D.i(end) + 1):m;
    index.L.j = index.D.j;
    index.U.i = index.D.i;
    index.U.j = (index.D.j(end) + 1):n;
    index.M.i = index.L.i;
    index.M.j = index.U.j;

    j = s(i) + j;

end

end
```

The block MLDU algorithm was implemented using MATLAB. It represents a typical first attempt without many optimizations. It uses pre-allocation for the output matrices and a nested function for the splitting of the matrix.

## 3.1 Verification and performance analysis

A simple test was required to verify the correctness of the implementation of the algorithm and analyze its performance. The following code was used:

```
function [ E ] = Test_Function_1( n, MLDU_Function )
% This test function constructs the A matrix and subsequently calculates
% the block MLDU factorization and the Frobenius norm of the error.

% Parameters
N = n^2;

% Generate matrix
A = Matrix_A(n);

% Generate blocks
s = 2*ones(N/2,1);

% Calculate Block MLDU factorization
[L,D,U] = MLDU_Function(A,s);

% Calculate error norm
E = norm(A - (L + D)*(D\(D + U)),'fro');

end
```

```
function [A] = Matrix_A(n)
% test matrix as provided by Jos Maubach

A = kron(eye(n), ...
         gallery('tridiag',n,-1,2,-1)) + ...
         kron(gallery('tridiag',n,-1,2,-1), eye(n));

end
```

As an example, matrix A and s are provided below ($n = 4$):

$$A = \left[\begin{array}{cccc|cccc|cccc|cccc}
4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
-1 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 4 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4
\end{array}\right]$$

$$s = \begin{bmatrix} 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \end{bmatrix}$$

Running the test with $n = 100$ creates a $10.000 \times 10.000$ matrix. Calculating the Frobenius norm of the error results in:

```
>> [ E ] = Test_Function_1( 100, @MLDU_Simple )

E =

1.1409e-13
```

The resulting error is small enough to be confident in the correct operation of the implementation, at least for this test case.

Running the profiler with the same input results in the following table:
The first thing to note is that the runtime is about 30 seconds for this test case, of which roughly 20 seconds are spend by the `MLDU_Simple` function. A $10.000 \times 10.000$ matrix is not really small, but it's not large either. As such, 20 seconds to provide the factorization is a rather poor result.

| Function Name | Calls | **Total Time** | Self Time* | Total Time Plot (dark band = self time) |
|---|---|---|---|---|
| Test_Function_1 | 1 | 32.791 s | 11.604 s | |
| MLDU_Simple | 1 | 21.134 s | 20.859 s | |
| MLDU_Simple>update_index_nested | 5000 | 0.275 s | 0.275 s | |
| Matrix_A | 1 | 0.054 s | 0.010 s | |
| gallery | 2 | 0.041 s | 0.014 s | |
| elmat/private/tridiag | 2 | 0.016 s | 0.010 s | |
| elmat/private/argclass | 2 | 0.010 s | 0.005 s | |
| spdiags | 2 | 0.006 s | 0.006 s | |
| ismember | 8 | 0.005 s | 0.001 s | |
| ismember>ismemberR2012a | 8 | 0.004 s | 0.002 s | |
| kron | 2 | 0.004 s | 0.004 s | |
| ismember>ismemberBuiltinTypes | 8 | 0.002 s | 0.002 s | |

To get a better understanding of what is going on, the following profiler information was pulled up:

**Function listing**

| time | Calls | line | |
|---|---|---|---|
| | | 1 | function [ E ] = Test_Function_1( n, MLDU_Function ) |
| | | 2 | % This test function constructs the A matrix and subsequently calculates |
| | | 3 | % the block MLDU factorization and the Frobenius norm of the error. |
| | | 4 | |
| | | 5 | % Parameters |
| < 0.001 | 1 | 6 | N = n^2; |
| | | 7 | |
| | | 8 | % Generate matrix |
| 0.059 | 1 | 9 | A = Matrix_A(n); |
| | | 10 | |
| | | 11 | % Generate blocks |
| < 0.001 | 1 | 12 | s = 2*ones(N/2,1); |
| | | 13 | |
| | | 14 | % Calculate Block MLDU factorization |
| 21.137 | 1 | 15 | [L,D,U] = MLDU_Function(A,s); |
| | | 16 | |
| | | 17 | % Calculate error norm |
| 11.567 | 1 | 18 | E = norm(A - (L + D)*(D\(D + U)),'fro'); |
| | | 19 | |
| 0.029 | 1 | 20 | end |

About 12 seconds of the runtime are spent calculating the Frobenius norm of the error, which can clearly be omitted from the performance analysis. Zooming in on the profiler information of the `MLDU_Simple.m` function yields:

## 3.2 Discussion of the results

The results mostly speak for themselves. A huge part of the runtime of the implementation is spent on line 27, where the results of the Schur complement update gets subtracted from the matrix. Even though there are some calculations involved in this line, they could not account for this excessive runtime.

Calculating the Schur complement happens at lines 23, 24 and 25, which takes about 1.5 seconds. Most of that time is not actually spent doing any calculations, but rather accessing the elements of matrix `U`. This is to be expected, as the sparse storage format of MATLAB is catered towards accessing columns rather than rows. Further proof of this can be found at line 21, where separating the elements of `U` from `A` actually takes more time than calculating the Schur complement update.

The point is that most of the runtime of line 27 will most likely be spent on the memory operations associated with inserting columns and rows of data into an existing sparse matrix. There are a couple of obvious possibilities that might be tried to circumvent this problem:

- Switch from sparse to full matrix storage.
- Store rows in a transposed fashion.
- Use a matrix storage format that doesn't require as many memory operations to insert rows and columns.

**Function listing**

| time | Calls | line | |
|---|---|---|---|
| | | 1 | function [ L, D, U ] = MLDU_Simple( A, s ) |
| | | 2 | |
| | | 3 | % Creating indices structure |
| < 0.001 | 1 | 4 | j = 0; |
| < 0.001 | 1 | 5 | [m,n] = size(A); |
| < 0.001 | 1 | 6 | i j = struct('i',[],'j',[]); |
| < 0.001 | 1 | 7 | index = struct('M',i_j,'L',i_j,'D',i_j,'U',i_j); |
| | | 8 | |
| | | 9 | % Preallocate output matrices |
| < 0.001 | 1 | 10 | L = zeros(m,n); |
| < 0.001 | 1 | 11 | D = zeros(m,n); |
| < 0.001 | 1 | 12 | U = zeros(m,n); |
| | | 13 | |
| | | 14 | % Loop over the blocks |
| < 0.001 | 1 | 15 | for i = 1:length(s) |
| | | 16 | |
| 0.355 | 5000 | 17 | update_index_nested(i); |
| | | 18 | |
| 0.449 | 5000 | 19 | L(index.L.i,index.L.j) = A(index.L.i,index.L.j); |
| 0.060 | 5000 | 20 | D(index.D.i,index.D.j) = A(index.D.i,index.D.j); |
| 1.734 | 5000 | 21 | U(index.U.i,index.U.j) = A(index.U.i,index.U.j); |
| | | 22 | |
| 1.452 | 5000 | 23 | schur_complement = A(index.L.i,index.L.j)* ... |
| | 5000 | 24 | (A(index.D.i,index.D.j)\ ... |
| | 5000 | 25 | A(index.U.i,index.U.j)); |
| | | 26 | |
| 17.081 | 5000 | 27 | A(index.M.i,index.M.j) = A(index.M.i,index.M.j) - schur_complement; |
| | | 28 | |
| 0.002 | 5000 | 29 | end |
| | | 30 | |
| | | 31 | % Nested index function |
| | | 32 | function [ ] = update_index_nested(i) |
| | | 33 | |
| | | 34 | index.D.i = (1:s(i)) + j; |
| | | 35 | index.D.j = index.D.i; |
| | | 36 | index.L.i = (index.D.i(end) + 1):m; |
| | | 37 | index.L.j = index.D.j; |
| | | 38 | index.U.i = index.D.i; |
| | | 39 | index.U.j = (index.D.j(end) + 1):n; |
| | | 40 | index.M.i = index.L.i; |
| | | 41 | index.M.j = index.U.j; |
| | | 42 | |
| | | 43 | j = s(i) + j; |
| | | 44 | |
| | | 45 | end |
| | | 46 | |
| < 0.001 | 1 | 47 | end |

The first possibility may seem silly, but is not as stupid as may at first appear. Fill-in is a real nemesis of algorithms like Block MLDU, and can make a sparse matrix very full in some instances. Switching to full matrix storage in these situations makes perfect sense and will provide a very healthy performance improvement.

Storing the rows as transposed columns can have tangible benefits, but it has certain disadvantages as well. The most obvious downside is the extra memory consumption, as the matrix will probably need to be stored twice. The second problem is that the block nature of the algorithm makes a lot of the work required to adapt the code that much harder.
It should be noted that this possible improvement should definitely be investigated further, as it could be a huge advantage in situations where the input matrix is symmetric.

Switching to a different matrix storage format will probably provide the most benefits, especially if the storage format is designed from scratch and tailored to the needs of the Block MLDU algorithm. The second implementation of the algorithm will focus on this possibility.

# Chapter 4

# Reduced memory consumption

The initial implementation has a couple of problems, one of which is it's memory consumption. This excessive memory consumption is mostly caused by the implementation of the memory preallocation for the matrices `L`, `D` and `U`. Using `zeros(m,n)` results in full matrices for `L`, `D` and `U`, while they are often far from full.

The painful part is the fact that matrices `L`, `D` and `U` grow dynamically with each iteration of the `for`-loop. Adding to the problem is that fill-in makes it impossible to know a priory by just how much they grow each iteration. Memory preallocation is a real pig in these type of situations and resorting to a full `zeros` matrix is often the only way that doesn't involve very complicated or convoluted code.

Dynamic growth of a matrix in a `for`-loop is evil, mostly because matrices require a contiguous memory block. Adding data to a matrix involves allocating an entirely new contiguous block of memory for the matrix, copying the old values from the previous block to the new, then releasing the old block for potential reuse. These memory operations are time consuming and should be avoided in `for`-loops. The most common way to deal with this problem is to use memory preallocation, but the other possibility is to use a data structure that drops the contiguous memory range requirement.

With an eye on making the implementation more memory efficient for a variety or reasons, an effort was undertaken to tackle the problem. As stated earlier, preallocation is difficult to achieve. With this in mind, the plan was to NOT use memory preallocation, but instead write the code in such a way that the negative effects of not using memory preallocation are largely avoided. MATLAB has a data structure known as a "cell array",[1] the most obvious differentiating qualities between it and a matrix data structure are:

- Can contain different kind of elements. A collection of an "int", a matrix and a string can all be stored in the same cell array.
- No contiguous memory range required. The separate entries of the cell array may require a contiguous memory range, but the collection of the entries might be stored separately.

The most obvious quality that cell arrays and matrices share is their ability to be indexed, something that a "struct" lacks.[2] The improved implementation uses cell arrays to store the additions of the matrices `L`, `D` and `U`, with each element of the cell array containing a COO representation of the `for`-loop additions.[3]

[1] *cell documentation*. Mathworks. 2018. URL: `https://nl.mathworks.com/help/matlab/ref/cell.html`.

[2] *struct documentation*. Mathworks. 2018. URL: `https://nl.mathworks.com/help/matlab/ref/struct.html`.

[3] *Sparse matrix*. Wikipedia. 2018. URL: `https://en.wikipedia.org/wiki/Sparse_matrix#Coordinate_list_(COO)`.

## 4.1 The code

```
function [ L, D, U ] = MLDU_Simple_cell( A, s )

% Creating indices structure
j = 0;
[m,n] = size(A);
s_l = length(s);
i_j = struct('i',[],'j',[]);
index = struct('M',i_j,'L',i_j,'D',i_j,'U',i_j);

% Preallocate output matrices
L_cell = cell(size(s));
D_cell = L_cell; U_cell = L_cell;

% Loop over the blocks
for i = 1:s_l

    j = update_index_nested(s(i),j,m,n);

    L_cell{i} = fill_cells_local(A,L_cell{i},index.L.i,index.L.j);
    D_cell{i} = fill_cells_local(A,D_cell{i},index.D.i,index.D.j);
    U_cell{i} = fill_cells_local(A,U_cell{i},index.U.i,index.U.j);

    shift_index_nested(i,s_l);

    schur_complement =  A(index.L.i,index.L.j)* ...
                       (A(index.D.i,index.D.j)\ ...
                        A(index.U.i,index.U.j));

    A(index.M.i,index.M.j) = A(index.M.i,index.M.j) - schur_complement;

end

L = collaps_cell_local(L_cell,m,n);
D = collaps_cell_local(D_cell,m,n);
U = collaps_cell_local(U_cell,m,n);

% Nested index update function
function [ j ] = update_index_nested( s, j, m, n )

    index.D.i = (1:s) + j;
    index.D.j = index.D.i;
    index.L.i = (index.D.i(end) + 1):m;
    index.L.j = index.D.j;
    index.U.i = index.D.i;
    index.U.j = (index.D.j(end) + 1):n;
    index.M.i = index.L.i;
    index.M.j = index.U.j;

    j = s + j;

end

% Nested index shift function.
function [ ] = shift_index_nested( i, s_end )

    D_cell{i}([1,2],:) =  D_cell{i}([1,2],:) + ...
                          [(index.D.i(1) - 1);(index.D.j(1) - 1)];

    % Only shift the index when not at the last itteration.
    if i ≠ s_end

        L_cell{i}([1,2],:) =  L_cell{i}([1,2],:) + ...
                              [(index.L.i(1) - 1);(index.L.j(1) - 1)];
```

```
64
65          U_cell{i}([1,2],:) =  U_cell{i}([1,2],:) + ...
66                                [(index.U.i(1) - 1);(index.U.j(1) - 1)];
67
68      end
69
70  end
71
72  end
73
74  % Local function that fills the cell array.
75  function [ A_cell ] = fill_cells_local( A, A_cell, index_i, index_j )
76
77      % Insert COO representation of selected A entries.
78      [A_cell(1,:),A_cell(2,:),A_cell(3,:)] = find(A(index_i,index_j));
79
80  end
81
82  % Local function that converts the cell array to a regular sparse matrices.
83  function [ A ] = collaps_cell_local( A_cell, m, n )
84
85      % Strange matlab syntax that just works in this particalur case...
86      A_col = [A_cell{:}];
87
88      % Convert COO to CCS (MATLAB) sparse.
89      A = sparse(A_col(1,:),A_col(2,:),A_col(3,:),m,n);
90
91  end
```

## 4.2 Discussion of the code

This implementation creates one new nested function, two new local functions and increases the number of lines from 47 to 91, so the reduced memory consumption does come at the cost of code complexity.

## 4.3 Profiler results

The profiler results below are generated by running:

```
[ E ] = Test_Function_1( 100, @MLDU_Simple_cell )
```

| time | Calls | line | |
|---|---|---|---|
| | | 1 | function [ E ] = Test_Function_1( n, MLDU_Function ) |
| | | 2 | % This test function constructs the A matrix and subsequently calculates |
| | | 3 | % the block MLDU factorization and the Frobenius norm of the error. |
| | | 4 | |
| | | 5 | % Parameters |
| < 0.001 | 1 | 6 | N = n^2; |
| | | 7 | |
| | | 8 | % Generate matrix |
| 0.014 | 1 | 9 | A = Matrix_A(n); |
| | | 10 | |
| | | 11 | % Generate blocks |
| < 0.001 | 1 | 12 | s = 2*ones(N/2,1); |
| | | 13 | |
| | | 14 | % Calculate Block MLDU factorization |
| 16.450 | 1 | 15 | [L,D,U] = MLDU_Function(A,s); |
| | | 16 | |
| | | 17 | % Calculate error norm |
| 0.985 | 1 | 18 | E = norm(A - (L + D)*(D\(D + U)),'fro'); |
| | | 19 | |
| 0.003 | 1 | 20 | end |

This is exactly the same as the previous situation. The error is also acceptable:

```
E =

1.1287e-13
```

Apart from the targeted result that the memory consumption has decreased, a couple of other noteworthy positive changes occurred as well.

One of the positive aspect is that calculating the error has a vastly reduced runtime, from about 10 seconds to just under one second. This is caused by the fact that matrices `L`, `D` and `U` are now sparse, making the calculation a lot faster.
The runtime required to save the matrices `L`, `D` and `U` has actually decreased. The initial implementation spends 2.243 seconds at lines 19 through 21 while the cell array implementation takes a total of 1.187 seconds at lines 19, 20, 21 and 33, 34, 35.

Lastly and maybe the most significant side effect of the cell array implementation is the runtime reduction of line 26, from about 17 seconds to about 13 seconds, a reduction of about 25%. This is probably caused by the reduced memory consumption itself, allowing the cache of the CPU to operate more effectively.

| time | Calls | line | |
|---|---|---|---|
| | | 1 | function [ L, D, U ] = MLDU_Simple_cell( A, s ) |
| | | 2 | |
| | | 3 | % Creating indices structure |
| < 0.001 | 1 | 4 | j = 0; |
| < 0.001 | 1 | 5 | [m,n] = size(A); |
| < 0.001 | 1 | 6 | s_l = length(s); |
| < 0.001 | 1 | 7 | i_j = struct('i',[],'j',[]); |
| < 0.001 | 1 | 8 | index = struct('M',i_j,'L',i_j,'D',i_j,'U',i_j); |
| | | 9 | |
| | | 10 | % Preallocate output matrices |
| < 0.001 | 1 | 11 | L_cell = cell(size(s)); |
| < 0.001 | 1 | 12 | D_cell = L_cell; U_cell = L_cell; |
| | | 13 | |
| | | 14 | % Loop over the blocks |
| < 0.001 | 1 | 15 | for i = 1:s_l |
| | | 16 | |
| 0.364 | 5000 | 17 | j = update_index_nested(s(i),j,m,n); |
| | | 18 | |
| 0.404 | 5000 | 19 | L_cell{i} = fill_cells_local(A,L_cell{i},index.L.i,index.L.j); |
| 0.108 | 5000 | 20 | D_cell{i} = fill_cells_local(A,D_cell{i},index.D.i,index.D.j); |
| 0.553 | 5000 | 21 | U_cell{i} = fill_cells_local(A,U_cell{i},index.U.i,index.U.j); |
| | | 22 | |
| 0.286 | 5000 | 23 | shift_index_nested(i,s_l); |
| | | 24 | |
| 1.499 | 5000 | 25 | schur_complement = A(index.L.i,index.L.j)* ... |
| | 5000 | 26 | (A(index.D.i,index.D.j)\ ... |
| | 5000 | 27 | A(index.U.i,index.U.j)); |
| | | 28 | |
| 13.109 | 5000 | 29 | A(index.M.i,index.M.j) = A(index.M.i,index.M.j) - schur_complement; |
| | | 30 | |
| 0.002 | 5000 | 31 | end |
| | | 32 | |
| 0.031 | 1 | 33 | L = collaps_cell_local(L_cell,m,n); |
| 0.003 | 1 | 34 | D = collaps_cell_local(D_cell,m,n); |
| 0.088 | 1 | 35 | U = collaps_cell_local(U_cell,m,n); |
| | | 36 | |

# Chapter 5

# Increased memory bandwidth

Line 29 of the function `MLDU_Simple_cell` is the obvious bottleneck of the implementation. This line gets called every iteration of the `for` loop and adds data to the matrix `A`. This addition of data requires many memory operations, which makes it slow. The performance of the main memory subsystem of a computer is comprised of two aspects, bandwidth and latency. Bandwidth is a measure of maximum throughput and latency is a measure of the access time.

This chapter tries to shed some light on which of the two main memory performance indicators is to blame for the poor performance of line 29.

## 5.1 GPU

The GPU is interesting in this scenario because dedicated GPU's contains VRAM. Comparing VRAM to standard RAM, which makes up the main memory of a computer, reveals that VRAM has a much higher bandwidth than RAM. The downside is that VRAM has a somewhat higher latency than RAM.

Implementing the block MLDU algorithm on the GPU would make it possible to identify whether latency or bandwidth is to blame for the poor runtime of line 29. A bandwidth bottleneck would result in a faster execution of line 29 and a latency bottleneck would cause it to be slightly slower then on the CPU.

## 5.2 MATLAB Sparse gpuArray limitations

MATLAB facilitates GPGPU programming, but has limitations, especially when it comes to sparse matrices. The most notable limitations for the block MLDU algorithm are the inability to index in sparse gpuArrays and the fact that the "\" operator is not feature complete.

These limitations make it very unappealing to write a "simple" MATLAB implementation of the block MLDU algorithm on the GPU. The overall performance will almost certainly be slower than the CPU implementation, because all the hacks required to get it functional will swamp any floating point performance benefits that the GPU has over the CPU.

Regardless, it will provide insight into the latency versus bandwidth question and as such is still valuable to this project.

### 5.2.1 The code

```
function [ L, D, U ] = MLDU_Simple_GPU( A, s )

% Creating indices structure
j = 0;
[m,n] = size(A);
s_l = length(s);
i_j = struct('i',[],'j',[]);
index = struct('M',i_j,'L',i_j,'D',i_j,'U',i_j);

% Convert A to gpuArray.
A = gpuArray(A);

% Loop over the blocks to perform the factorization.
for i = 1:(s_l - 1)

    j = update_index_nested(s(i),j,m,n);

    [index_m_L,index_n_L] = GPUArray_select_local(index.L.i,index.L.j,m,n);
    [index_m_D,index_n_D] = GPUArray_select_local(index.D.i,index.D.j,m,n);
    [index_m_U,index_n_U] = GPUArray_select_local(index.U.i,index.U.j,m,n);

    D_inv = Calculate_D_inverse_local(A,index_m_D,index_n_D,m,n);

    schur_complement = (index_m_L*A*index_n_L)*D_inv*(index_m_U*A*index_n_U);

    A = A - schur_complement;

end

% Reset for the next loop.
j = 0;
A = gather(A);

% Preallocate output matrices
L_cell = cell(size(s));
D_cell = L_cell; U_cell = L_cell;

% Loop over the blocks to extract L, D, U.
for i = 1:s_l

    j = update_index_nested(s(i),j,m,n);

    L_cell{i} = fill_cells_local(A,L_cell{i},index.L.i,index.L.j);
    D_cell{i} = fill_cells_local(A,D_cell{i},index.D.i,index.D.j);
    U_cell{i} = fill_cells_local(A,U_cell{i},index.U.i,index.U.j);

    shift_index_nested(i,s_l);

end

L = collaps_cell_local(L_cell,m,n);
D = collaps_cell_local(D_cell,m,n);
U = collaps_cell_local(U_cell,m,n);

% Nested index update function
function [ j ] = update_index_nested( s_i, j, m, n )

    index.D.i = (1:s_i) + j;
    index.D.j = index.D.i;
    index.L.i = (index.D.i(end) + 1):m;
    index.L.j = index.D.j;
    index.U.i = index.D.i;
    index.U.j = (index.D.j(end) + 1):n;
    index.M.i = index.L.i;
```

```
65      index.M.j = index.U.j;
66
67      j = s_i + j;
68
69  end
70
71  % Nested index shift function.
72  function [ ] = shift_index_nested( i, s_end )
73
74      D_cell{i}([1,2],:) =  D_cell{i}([1,2],:) + ...
75                            [(index.D.i(1) - 1);(index.D.j(1) - 1)];
76
77      % Only shift the index when not at the last itteration.
78      if i ≠ s_end
79
80          L_cell{i}([1,2],:) =  L_cell{i}([1,2],:) + ...
81                                [(index.L.i(1) - 1);(index.L.j(1) - 1)];
82
83          U_cell{i}([1,2],:) =  U_cell{i}([1,2],:) + ...
84                                [(index.U.i(1) - 1);(index.U.j(1) - 1)];
85
86      end
87
88  end
89
90  end
91
92  % Local function that fills the cell array.
93  function [ M_cell ] = fill_cells_local( M, M_cell, index_i, index_j )
94
95      % Insert COO representation of selected A entries.
96      [M_cell(1,:),M_cell(2,:),M_cell(3,:)] = find(M(index_i,index_j));
97
98  end
99
100 % Local function that converts the cell array to a regular sparse matrices.
101 function [ M ] = collaps_cell_local( M_cell, m, n )
102
103     % Strange matlab syntax that just works in this particalur case...
104     M_col = [M_cell{:}];
105
106     % Convert COO to CCS (MATLAB) sparse.
107     M = sparse(M_col(1,:),M_col(2,:),M_col(3,:),m,n);
108
109 end
110
111 % Local function to facilitate slicing a sparse gpuArray.
112 function [ index_m, index_n ] = GPUArray_select_local( i, j, m, n )
113
114 index_m = gpuArray(sparse(i,i,1,m,m));
115 index_n = gpuArray(sparse(j,j,1,n,n));
116
117 end
118
119 function [ D_inv ] = Calculate_D_inverse_local( M, index_i, index_j, m, n )
120
121 [i,j,v] = find(index_i*M*index_j);
122
123 [i,j,v] = gather(i,j,v);
124
125 M_cpu = sparse((i - i(1) + 1),(j - j(1) + 1),v);
126
127 [x,y,z] = find(inv(M_cpu));
128
129 D_inv = gpuArray(sparse((x + i(1) - 1),(y + j(1) - 1),z,m,n));
130
131 end
```

### 5.2.2 Profiler results

The profiler results below are generated by running:

```
[ E ] = Test_Function_1( 40, @MLDU_Simple_GPU )
```

The total runtime of this reduced size test was about 16 seconds, which is way slower than the `MLDU_Simple_cell` implementation, which takes about 0.4 seconds. The error of `MLDU_Simple_GPU` was again reasonable at $2.9072e-14$.

| time | Calls | line | |
|---|---|---|---|
| | | 1 | function [ L, D, U ] = MLDU_Simple_GPU( A, s ) |
| | | 2 | |
| | | 3 | % Creating indices structure |
| < 0.001 | 1 | 4 | j = 0; |
| < 0.001 | 1 | 5 | [m,n] = size(A); |
| < 0.001 | 1 | 6 | s_l = length(s); |
| < 0.001 | 1 | 7 | i_j = struct('i',[],'j',[]); |
| < 0.001 | 1 | 8 | index = struct('M',i_j,'L',i_j,'D',i_j,'U',i_j); |
| | | 9 | |
| | | 10 | % Convert A to gpuArray. |
| 1.965 | 1 | 11 | A = gpuArray(A); |
| | | 12 | |
| | | 13 | % Loop over the blocks to perform the factorization. |
| < 0.001 | 1 | 14 | for i = 1:(s_l - 1) |
| | | 15 | |
| 0.030 | 799 | 16 | j = update_index_nested(s(i),j,m,n); |
| | | 17 | |
| 1.903 | 799 | 18 | [index_m_L,index_n_L] = GPUArray_select_local(index.L.i,index.L.j,m,n); |
| 2.209 | 799 | 19 | [index_m_D,index_n_D] = GPUArray_select_local(index.D.i,index.D.j,m,n); |
| 1.899 | 799 | 20 | [index_m_U,index_n_U] = GPUArray_select_local(index.U.i,index.U.j,m,n); |
| | | 21 | |
| 4.024 | 799 | 22 | D_inv = Calculate_D_inverse_local(A,index_m_D,index_n_D,m,n); |
| | | 23 | |
| 3.342 | 799 | 24 | schur_complement = (index_m_L*A*index_n_L)*D_inv*(index_m_U*A*index_n_U); |
| | | 25 | |
| 0.405 | 799 | 26 | A = A - schur_complement; |
| | | 27 | |
| < 0.001 | 799 | 28 | end |
| | | 29 | |
| | | 30 | % Reset for the next loop. |
| < 0.001 | 1 | 31 | j = 0; |
| 0.005 | 1 | 32 | A = gather(A); |
| | | 33 | |
| | | 34 | % Preallocate output matrices |
| < 0.001 | 1 | 35 | L_cell = cell(size(s)); |
| < 0.001 | 1 | 36 | D_cell = L_cell; U_cell = L_cell; |
| | | 37 | |
| | | 38 | % Loop over the blocks to extract L, D, U. |
| < 0.001 | 1 | 39 | for i = 1:s_l |
| | | 40 | |
| 0.018 | 800 | 41 | j = update_index_nested(s(i),j,m,n); |
| | | 42 | |
| 0.022 | 800 | 43 | L_cell{i} = fill_cells_local(A,L_cell{i},index.L.i,index.L.j); |
| 0.012 | 800 | 44 | D_cell{i} = fill_cells_local(A,D_cell{i},index.D.i,index.D.j); |
| 0.036 | 800 | 45 | U_cell{i} = fill_cells_local(A,U_cell{i},index.U.i,index.U.j); |
| | | 46 | |
| 0.027 | 800 | 47 | shift_index_nested(i,s_l); |
| | | 48 | |
| < 0.001 | 800 | 49 | end |
| | | 50 | |
| 0.003 | 1 | 51 | L = collaps_cell_local(L_cell,m,n); |
| < 0.001 | 1 | 52 | D = collaps_cell_local(D_cell,m,n); |
| 0.005 | 1 | 53 | U = collaps_cell_local(U_cell,m,n); |

The most interesting timing result for this discussion is that of line 26 (above), which has a value of 0.405. Line 29 of `MLDU_Simple_cell` executes the same operation, but takes only 0.164 seconds.

The profiler results of `[ E ] = Test_Function_1( 40, @MLDU_Simple_cell )` are provided below:

```
  time    Calls   line
                    1 function [ L, D, U ] = MLDU_Simple_cell( A, s )
                    2
                    3 % Creating indices structure
                1   4 j = 0;
< 0.001         1   5 [m,n] = size(A);
                1   6 s_l = length(s);
< 0.001         1   7 i_j = struct('i',[],'j',[]);
< 0.001         1   8 index = struct('M',i_j,'L',i_j,'D',i_j,'U',i_j);
                    9
                   10 % Preallocate output matrices
< 0.001         1  11 L_cell = cell(size(s));
                1  12 D_cell = L_cell; U_cell = L_cell;
                   13
                   14 % Loop over the blocks
< 0.001         1  15 for i = 1:s_l
                   16
  0.030       800  17     j = update_index_nested(s(i),j,m,n);
                   18
  0.027       800  19     L_cell{i} = fill_cells_local(A,L_cell{i},index.L.i,index.L.j);
  0.015       800  20     D_cell{i} = fill_cells_local(A,D_cell{i},index.D.i,index.D.j);
  0.028       800  21     U_cell{i} = fill_cells_local(A,U_cell{i},index.U.i,index.U.j);
                   22
  0.031       800  23     shift_index_nested(i,s_l);
                   24
  0.062       800  25     schur_complement =  A(index.L.i,index.L.j)* ...
              800  26                        (A(index.D.i,index.D.j)\ ...
              800  27                         A(index.U.i,index.U.j));
                   28
  0.164       800  29     A(index.M.i,index.M.j) = A(index.M.i,index.M.j) - schur_complement;
                   30
< 0.001       800  31 end
                   32
  0.002         1  33 L = collaps_cell_local(L_cell,m,n);
< 0.001         1  34 D = collaps_cell_local(D_cell,m,n);
  0.004         1  35 U = collaps_cell_local(U_cell,m,n);
                   36
```

The clear result from `MLDU_Simple_GPU` is that it does not constitute an improvement. This may very well be down to the limiting aspects of MATLAB sparse gpuArrays, but further investigation would be required to know for certain. The second lesson from `MLDU_Simple_GPU` is that the operation at line 29 or 26 is latency bound.

# Chapter 6

# Conclusion

The goal of this Capita Selecta is to investigate the possibilities for a fast MATLAB based implementation of the Block MLDU algorithm. The most distinguishing drawback of using MATLAB for high performance code is its interpreter overhead.

The implementations provided in this report are not particularly fast, but they do indicate one thing. Because almost all of the runtime of the implementation is spent on a single MATLAB statement, it can be concluded that the interpreter overhead of MATLAB is not to blame for the disappointing runtime.

Finding that the MATLAB interpreter is not the performance bottleneck, incentivized further research into a MATLAB implementation of the Block MLDU algorithm. This research has shown that the bottleneck is the storing operation of the Schur complement update into the matrix. Fill-in is the ultimate culprit of this problem, because it requires freeing extra memory for the additional non-zero elements. The memory operations required are DRAM latency bound and as such are very hard to accelerate.

It should be noted that an incomplete version of the Block MLDU algorithm would not suffer the aforementioned problems, and as such could easily be implemented in a high performance manner using a similar approach as provided in this report.

The conclusion of this Capita Selecta is that the default sparse matrix storage format (CCS) of MATLAB is sub optimal for this algorithm. Large performance gains should be possible if a matrix storage format was selected that circumvents the costly memory freeing operations associated with the Schur complement update.

One possible alternative to CCS would be COO, as it is more flexible. It's main advantage in this situation is the fact that the order in which the matrix entries are stored, does not alter the location of the entries in the matrix. This should circumvent the need for all the memory freeing operations that are currently frustration the performance.

The main challenge of the COO approach is searching through the elements of the matrix during the matrix splitting step of the algorithm. The lack of ordering in the COO format causes these operations to be much slower than in the CCS situation, so finding a way around this problem will again provide great performance improvements.

# Bibliography


[1] *cell documentation*. Mathworks. 2018. URL: https://nl.mathworks.com/help/matlab/ref/cell.html.

[2] *Sparse matrix*. Wikipedia. 2018. URL: https://en.wikipedia.org/wiki/Sparse_matrix#Coordinate_list_(COO).

[3] *struct documentation*. Mathworks. 2018. URL: https://nl.mathworks.com/help/matlab/ref/struct.html.