MATRIX INVERSION SPEED UP WITH CUDA

BY

JORGE SORIANO PINEDO

ELECTRICAL AND COMPUTER ENGINEERING

Submitted in partial fulfillment of the
requirements for the degree of
Electrical engineering in ECE
in the Graduate College of the
Illinois Institute of Technology

Approved _____
Adviser

Chicago, Illinois
7/29/2011

To my family, for their support,
even if at times I didn't deserve it.

To my friends and coworkers,
for making hard things look easy.

To my grandpa Sam,
wherever you are.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF EXAMPLES

ABSTRACT

In this project several mathematic algorithms are developed to obtain a matrix inversion method - that combines CUDA's parallel architecture and MATLAB – which is actually faster than MATLAB's built in inverse matrix function. This matrix inversion method is intended to be used for image reconstruction as a faster alternative to iterative methods with a comparable quality. The algorithms developed in this project are Gauss-Jordan elimination, Cholesky decomposition, Gaussian elimination and matrix multiplication. Gauss-Seidel is also featured in the report, but only as an alternative method of finding the inverse, since it has not been developed in the project.

CHAPTER 1

INTRODUCTION

Image reconstruction from acquired data is a technique with many uses nowadays. It is deployed in a wide variety of areas, such as medicine, security surveillance or biometrics, among others.

Any image reconstruction method can be described with the following linear model:

$$g = H \cdot f \qquad \rightarrow \qquad f^* = H^{-1} \cdot g$$

And its direct solution can be found by inverting the *H* system matrix. The following images are examples obtained with the implementation of matrix inversion algorithm.

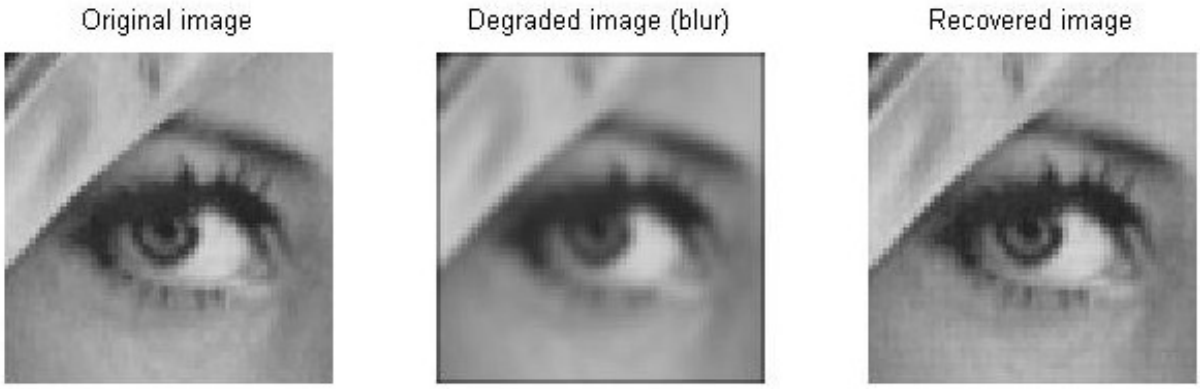

Figure 1.1.Original image to be reconstructed, degraded (blurred) image and recovered image using the inverse matrix reconstruction method developed in the project

The most extended way of recovering an image is through iterative methods. An iterative method is a mathematical procedure that generates a sequence of improving approximate solutions for a problem, with prefixed termination criteria. These methods

are often the only choice for nonlinear equations, but they are used for linear problems involving a huge number of variables where direct methods would be computationally very expensive.

On the other hand, direct methods attempt to solve the problem by direct inversion of H. In case that H is invertible this kind of method delivers an exact solution, but it is computationally expensive.

There are two main objectives for the elaboration of this project. The first one is to create a matrix inversion algorithm – thus, a direct method- accurate enough to be able to reconstruct a degraded image without visible artifacts in the recovered one. The second objective is to speed-up this matrix inversion, because for large sized matrix it is very time expensive.

In order to get a speed up for this method Compute Unified Device Architecture, or CUDA, is used. CUDA is a parallel architecture that permits massive simultaneous computing, useful for operations that are highly parallelizable. The purpose of using CUDA is to combine it with MATLAB using MEX files in order to execute faster the most timing consuming operation – matrix inversion - in MATLAB, and still have the results in MATLAB's environment.

CHAPTER 2

PARALLEL ARCHITECTURE

2.1 CUDA

The Compute Unified Device Architecture (CUDA) is NVIDIA's parallel computing architecture, and it enables powerful GPU hardware to C/C++, OpenCL and other programming interfaces. GPUs are capable of executing a huge number of threads at the same time with specific hardware for floating point arithmetic, 2D and 3D matrix cached access.

A GPU card that supports CUDA is a collection of multiprocessors where each of them has its own number of processors, and also its own fast shared memory, common to all the processors within. Inside every card all the multiprocessors share the card's global memory, which includes both constant and texture memory, and are cache-accessible from every processor.

From a software developer point of view, the CUDA model allows user defined functions named "kernels" that, when called, are executed simultaneously in N parallel threads as opposed to only once like traditional C functions. The number of threads to be executed in a thread-block (from now on, block) is decided by the developer, and the device is the one that will schedule the execution of each block. Likewise, the blocks will be put together into a grid, followed by scheduling execution of a grid on the collection of multiprocessors. (See figure X2). In other words, the programmer can define the threads in each block and the blocks in each grid, but further decisions are left to the device.

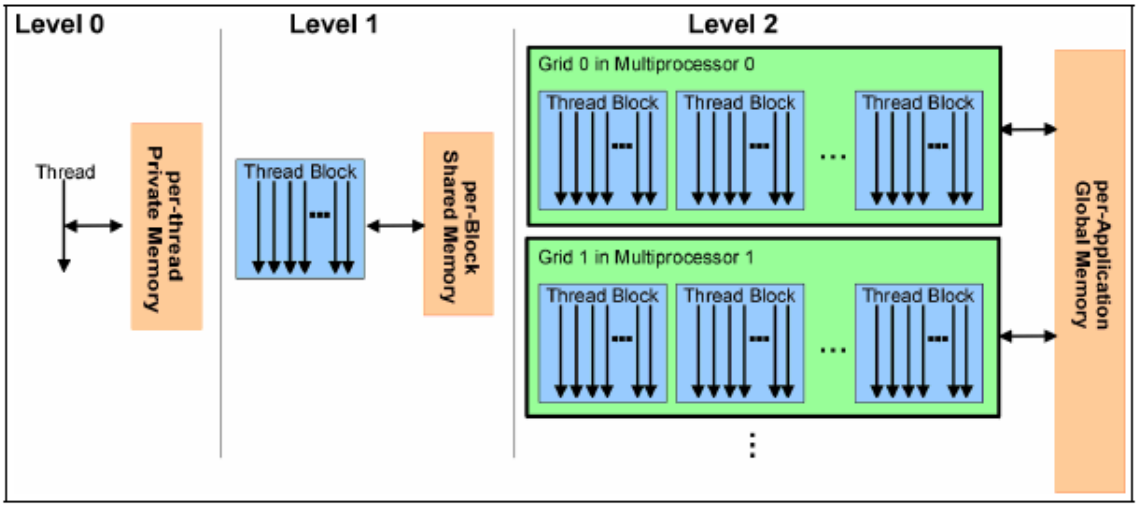

Figure 2.1. Grid, blocks and threads hierarchy.

However, there are more thread grouping than blocks and grids: the threads in blocks are sub-grouped in warps (groups of 32 threads). Each processor of the card can perform the same operation on each thread of a warp in a sequential fashion, so for optimal performance the programmer should avoid branching and get all the threads in a warp to execute the same instructions.

Each processor warp scheduler can switch content quickly and put a warp on hold during time consuming operations (like memory fetching). While these operations are taking place the scheduler will attempt to execute up to other 3 warps to fully utilize CUDA capabilities, it is important to submit a larger number of threads than the number of processors.

In the present report we will be referring to CUDA C/C++ programming interface and codes as CUDA without implying the rest of ways to use GPU computing for

simplicity. Also, all the codes explained in the report only will show CUDA C/C++ codes and structures.

2.2 CUDA kernels

Kernels, as stated in the point before, are CUDA user defined functions that run simultaneously in a group of threads in the GPU. The organization of the threads follows this structure:

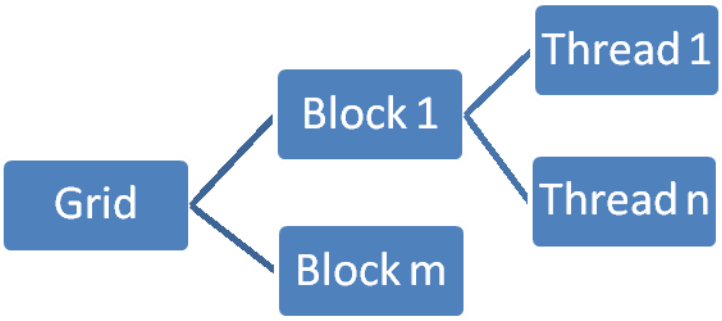

Figure 2.2. Organization of threads and blocks in a grid

A kernel can have a grid of a total of 65,356 blocks, and the blocks can have up to 512 or 1024 threads each (depending on the GPU card, in our case up to 512 only). The organization of blocks in the grid can be in one or two dimensions, and the organization of threads inside the blocks can be one, two or three-dimensional; since there can be as many different kernels as software developers, the organization is up to the person designing the kernel. There's no need to use all of the dimensions if not needed;

nevertheless, the maximum number of blocks and threads must not be surpassed in any case.

Whichever the organization, each thread has its unique thread identifier inside a block, and its block identifier inside the grid, so we can create a kernel in which every thread is unique and, if needed, can be assigned a different task than any other.

Because of this variable thread structure, we must setup the thread usage before calling a kernel. This setup must be done according to the needs of each kernel; if there's no set up the code won't compile correctly, but if the setup is wrong there won't be any compilation error and the code will be malfunctioning, leading to a crash while execution in the worst cases.

In order to setup a kernel, CUDA incorporates a new execution configuration syntax before the function arguments. The grid and block dimensions (thread count) are set between the <<<… >>> operators before the arguments of a kernel, which are summoned as if in a regular C function.

The following example illustrates the setup of a kernel named *myKernel* and its summon afterwards:

```
dim3 dimGrid( gridSizeX, gridSizeY);  //Grid dimensions
dim3 dimBlock( blockSizeX, blockSizeY, blockSizeZ); //Block dimensions
myKernel <<< dimGrid, dimBlock >>> ( argument1, argument2, ... ); // Call
```

2.3 CUDA function types

A CUDA C program is basically an extension of a C/C++ program with kernels. However, the compiler has to be able to differentiate between CUDA and C/C++ functions, and three different categories arise for that purpose:

Host functions: This are C/C++ functions called and executed in the CPU, and are preceded by the declaration specifier __host__. This kind of functions can have it omitted though, for they are the normal functions for a C program, and the compiler for CUDA interprets them like that if not stated otherwise. All the functions from point 2.3 belong to this kind.

Global functions: This kind of functions is called from the host but executed in the device, and are preceded by the specifier __global__. To put it simply, they are each and every function that a C/C++ code running in the CPU call to be executed in a GPU. CUDA kernels belong to this category.

Device functions: This last kind of functions are called and executed from the device (called from global functions). They are useful to keep global functions simple and modular, like any C/C++ program would be. Their specifier is __device__.

2.4 CUDA basic functions

There are many implemented CUDA C functions in its API, but there are three of them that are the basis to run any CUDA program; that is, allocating a variable in the device, copying the value of variables to and from the device, and releasing resources from the device.

Memory allocation:

Memory allocation should always be the first step in any program, and CUDA is not an exception. All the variables needed in a kernel must be allocated prior to its call

Table 2.1. CUDA memory allocation function.

cudaMalloc(void** ptr, int size) void** ptr: pointer to the variable allocated int size: number of bytes to be allocated for the variable pointed by ptr

Memory copy:

This function allows us to copy a variable to and from a device, and between device variables. It is useful to set the variables needed by a kernel, and to retrieve the outputs when the kernel has finished.

Table 2.2. CUDA memory copy function

cudaMemcpy(void * dst, const void * src, size_t count, enum cudaMemcpyKind kind) void *dst: Pointer to the variable where we will copy the value of src const void *src: Pointer to the variable from which we will copy the value to dst size_t count: Number of bytes to be copied enum cudaMemcpyKind kind: Copy from CPU to GPU, from GPU to CPU or from GPU to GPU.

Memory liberation

Once all the kernels in the code have been executed it's advisable to de-allocate the variables used.

Table 2.3. CUDA memory de-allocation function.

cudaFree(void *ptr) ptr: Pointer to the variable to release

2.5 Compiling a CUDA file

While a C/C++ file extension is .c or .cpp respectively, a CUDA file's extension is .cu. Hence, to compile a CUDA file you need a compiler that recognizes .cu file extension, and recognizes CUDA functions.

The nvcc compiler has been used for this purpose: it works great and the command line to use it is the same as gcc. To compile a file, it's only needed to type in a terminal window (in Linux OS):

nvcc -o <executable_name> <.cu file>

and the program will be ready to execute.

2.6 CUDA code example

The following example illustrates how the three categories of functions interact, how to set up and call a kernel, and the thread unique identification inside a kernel. This code calculates the minimum of two vectors element by element and returns a vector with those minimum values.

Table 2.4. CUDA code example

```
#include "cuda.h"                    // needed to use CUDA C/C++
#define BLOCK_SIZE 10

__device__ float min( float a, float b )
 {
        return (a<b)? a:b;
}
__global__ void findMinKernel( float *input1, float *input2, float *output )
{
        tx = threadIdx.x;                // Thread identifier inside the block
        bx = blockIdx.x;                 // Thread's block identifier inside the grid
        x = tx + BLOCK_SIZE * bx;

        a = input1[x]; b = input2[x];
        output[x] = min(a, b);
}

int main( int argc, char *argv[] ){
        float input1[ 2 * BLOCK_SIZE ];              // First input
        float input2[ 2 * BLOCK_SIZE ];              // Second input
        float output[ 2 * BLOCK_SIZE ];              // Output

        // give the inputs some values
        float *input1_dev;                   // First input in the device
        float *input2_dev;                   // Second input in the device
        float *output;                       // Output in the device
```

```
        // Variables' size to be allocated
        int size = 2 * BLOCK_SIZE * sizeof( float );

        // Allocation of the variables in the device
        cudaMalloc( (void**) &input1_dev, size );
        cudaMalloc( (void**) &input2_dev, size );
        cudaMalloc( (void**) &output_dev, size );

        // Copy the inputs from host to device
        cudaMemcpy( input1_dev, input1, size, cudaMemcpyHostToDevice );
        cudaMemcpy( input2_dev, input2, size, cudaMemcpyHostToDevice );

        // Set up the kernel
        dim3 dimGrid( 2 );                      // The same as dimGrid(2,1)
        dim3 dimBlock( BLOCK_SIZE );            // The same as dimBlock(BLOCK_SIZE, 1, 1)

        // Launching the kernel
        findMinKernel <<< dimGrid, dimBlock >>> ( input1_dev, input2_dev, output_dev );

        // Once the kernel has finished, we have to retrieve the output
        cudaMemcpy( output, output_dev, size, cudaMemcpyDeviceToHost );

        // Free CUDA variables if not needed anymore
        cudaFree( input1_dev );
        cudaFree( input2_dev );
        cudaFree( output_dev );

        // Do whatever we want with the output we got back
        ( ... )

        // exit the program
        return 0;
}
```

CHAPTER 3

MATLAB'S MEX FILES

3.1 Description of MEX files

MEX-files (Matlab Executable files) give the possibility to interface C/C++ or FORTRAN subroutines to MATLAB, and call them directly from MATLAB as if they were built-in functions. They also provide functionality to transfer data between MEX-files and MATLAB, and the ability to call MATLAB functions from C/C++ or FORTRAN code.

The two reasons any software developer finds to write a MEX-file are the following:

- The ability to call large existing C/C++ or FORTRAN routines directly from MATLAB without having to rewrite them as MATLAB files.
- To achieve speed. Bottlenecks can be rewritten as a MEX-file for efficiency

That last reason is the motivation to use MEX-files. Since CUDA programming can be done in a C/C++ language, and this language is compatible with MEX-files, we can create an application that combines CUDA, C/C++ and MEX-files at the same time to optimize MATLAB's most timing consuming operations.

3.2 MEX-files entry point

A MEX-file code written in C/C++ keeps almost the same structure as a typical C/C++, except for the main function. For MATLAB to be able to recognize the function

and execute it properly an entry point must be set, so it takes the place of the main function.

In a generic MATLAB function such as:

```
[output1, output2, … ] = function (input1, input2, …)
```

All the inputs are written in the right side of the function, and the outputs on the left. In MEX-files' entry point that structure is maintained to be consistent. The entry point syntax is as follows:

```
void mexFunction( int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
```

where:

nlhs: Number of arguments on the Left Hand Side

plhs: Pointers to the arguments on the Left Hand Side

nrhs: Number of arguments on the Right Hand Side

prhs: Pointers to the arguments on the Right Hand Side

With this structure, very similar to the one used in a C/C++ main, if we want to know the number of outputs we will check nlhs value and if we want to access to the second input we will have to access prhs[1], for instance.

3.3 Basic MEX functions

There are many MEX functions available to interface C/C++ with MATLAB, even to the extent of using a MATLAB function directly from the MEX function. Here

are only listed the two most important MEX functions: accessing to and creating MATLAB's variables.

MATLAB variables access:

It is mandatory to access MATLAB variables in order to do any computation with them, which is the whole point of creating any MEX file. The two functions that allow us to do so are the following:

Table 3.1. MATLAB's MEX functions to access variables

`void* mxGetData( *ptr )`	`double* mxGetPr( *ptr )`
`float *var1;` `var1 = (float*) mxGetData( prhs[0] );`	`double *var2;` `var2 = mxGetPr( prhs[1] );`

MATLAB variables creation:

It is also mandatory to create the output MATLAB variables to store the values computed within the MEX file. Again, here are two functions that serve that purpose:

Table 3.2. MATLAB's MEX functions to create variables

```
mxArray *mxCreateDoubleMatrix( mwSize m, mwSize n, mxComplexity ComplexFlag );

double *output1;
output1 = mxCreateDoubleMatrix( 3, 4, mxREAL );
```

```
mxArray *mxCreateNumericArray( mwSize ndim, const mwSize *dims,
    mxClassID classid, mxComplexity ComplexFlag );
float *output2;
int dims[2]; dims[0] = 5; dims[1] = 3;
```

```
output2 = mxCreateNumericMatrix( 2, dims, mxSINGLE_CLASS,
    mxREAL);
```

3.4 Compiling a MEX file

A MEX code file that uses C/C++ has .c extension, but it cannot be compiled with a typical C/C++ compiler such as gcc; instead, the mex compiler is needed. It is included within MATLAB, and can be found in most of UNIX distributions, if not all. The command syntax to compile a MEX file is the same as with any other compiler:

```
mex <filename>
```

and it can be called from inside MATLAB´s environment or from a system terminal. The output produced is a .mex file that can be called from MATLAB as with any other buil-in or m function.

3.5 MEX code example

This example returns the double of the input.

Table 3.3. MEX code example.

```
#include "mex.h"

void mexFunction( int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
  double *input,*output;
  int M, N;

  // Input checking: only one input admitted. If that's not the case,
  // print an error message and exit the program!
  if (nrhs != 1)
    mexErrMsgTxt( "There must be only one input" );

  // If any other input checking has to be made it MUST be done
  // before continuing with the code

  // Get the sizes of the input
  M = mxGetM( prhs[0] );        // M's value is the number of rows of the input
  N = mxGetN( prhs[0] );        // N's value is the number of cols of the input
  // Get the pointer to the input from MATLAB to operate with it
  input = mxGetPr( prhs[0] );

  // Create the MATLAB output
  plhs[0] = mxCreateDoubleMatrix( mxGetM(prhs[0]), mxGetN(prhs[0]), mxREAL);

  // Get the pointer to the output of MATLAB to operate with it
  output = mxGetPr( plhs[0] );

// Equivalent to a C/C++ from now on
  for ( int i = 0; i < M*N; i++ )
    output[i] = 2*input[i];

// We can end, the values are already stored in MATLAB's variables
  mexPrintf( "Done!\n");          // Equivalent to printf();

}
```

CHAPTER 4

COMBINING MEX AND CUDA FILES

4.1 Code structure

The only thing needed to merge both CUDA and MEX codes is to exchange the main function from the CUDA code for the entry point of the MEX file: having a main and a MEX entry point would render Matlab incapable of detecting the entry point of the file correctly.

The usual structure of the mexFunction would be the following:

- Get all the inputs needed from MATLAB.
- Allocate and copy all the variables needed for CUDA.
- Execute the kernels needed.
- Copy the results back to MATLAB variables.
- Free CUDA variables

The combined code should be saved as a .cu file because it is going to be compiled from a .cu file to a final .mex file for Matlab to use it.

4.2 Compilation of a merged code

To compile a merged code, two steps are needed:

1. Compile the .cu file with nvcc and only create its object, while including Matlab's libraries.

2. Compile the .o file (object) with the mex command while including CUDA's libraries

In order to do so in an automatic way from inside Matlab's environment, the function cudaToMex was created. Its code is the following:

Table 4.1. Code to compile a MEX file containing CUDA functions.

```
function result = cudaToMex( name )
%CUDATOMEX Compiles a .cu file and creates a .mex file in matlab
%   cudaToMex( name ), where name has NO extension
%
%   Example: if your file was multiply.cu, then type
%                cudaToMex( multipy );

system( sprintf( 'nvcc -I"%s/extern/include" -c %s.cu  -Xcompiler -fPIC', matlabroot, name) );

system( sprintf( 'mex -cxx %s.o -L /usr/local/cuda/lib64 -lcudart -lcublas -lcufft –L
                                /home/tesla0/cuLIB/include', name ) );
system( sprintf( 'rm %s.o', name ) );
end
```

The first instruction creates the object file, the second one creates the mex file, and the last one removes the object file, since we don't need it anymore.

4.3 Considerations

There are some minor issues that must be taken into account when programming a merged code:

- The input variables from MATLAB are constant and in no way must be modified, or else the program will crash during execution.
- While C/C++ reads and stores variables row – wise, MATLAB handles its variables column – wise (see figure x). To avoid any confusion when accessing variables, the following macro has been coded:

```
#define IDC2D(i, j, ld) (((j)*(ld))+(i))
```

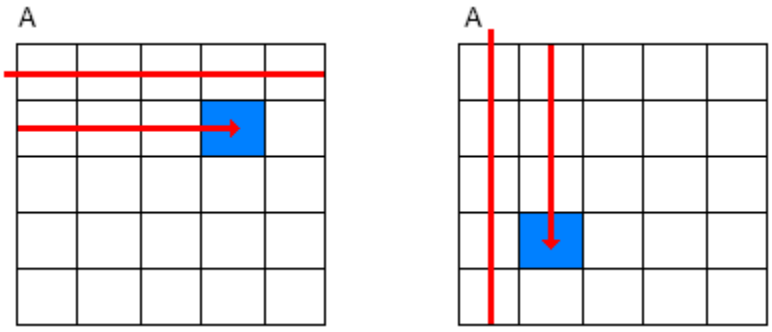

Figure 4.1. C/C++ and MATLAB's way of storing variables.

CHAPTER 5

MATRIX INVERSION

5.1 Image reconstruction as a linear inverse problem

The reconstruction of any image from acquired data is an inverse problem itself. Being f[x,y] an unknown image we want to obtain, h[x,y] the channel that unknown signal travels through and n[x,y] the additive noise the channel adds (uncorrelated to f[x,y]), we can express the image g[x,y] that is obtained by any measure system as:

$$\boldsymbol{g[x,y] = h[x,y] * f[x,y] + n[x,y]} \quad (1)$$

where the * operator denotes linear convolution. $\Delta \cdot \boldsymbol{Id}$

It is possible to express this same model in a matrix notation such as:

$$\boldsymbol{g = H \cdot f + n} \quad (2)$$

where capital letters express matrix and minus letters express vertical vectors. The direct approach to reconstruct the image f[x,y] would be one of the following methods:

$$\boldsymbol{f_{recovered} = H^{-1} \cdot g} \quad (3)$$

$$\boldsymbol{f_{recovered} = (H \cdot H^{T})^{-1} \cdot H^{T} \cdot g} \quad (4)$$

$$\boldsymbol{f_{recovered} = (H \cdot H^{T} + \Delta \cdot Id)^{-1} \cdot H^{T} \cdot g} \quad (5)$$

The first method is not possible in most of cases (because $\boldsymbol{H}$ can be not a square matrix) and, even if applicable, it is not a good approach because it does not consider the noise introduced by the channel.

As for the later ones, both of them need to invert large matrices to recover the original image, and these matrices are big enough to make finding its inverse very time consuming, even for small images such as 64x64 pixels (see table x). Hence the need of a speed-up to confront this problem, and the need to combine CUDA and MATLAB.

Table 5.1. Inversion time of large matrices in MATLAB

Image Size	H matrix size	Inversion time (MATLAB)
64 x 64	4096 x 4096	24.29 s
80 x 80	6400 x 6400	1 min 27.46 s
96 x 96	9216 x 9216	4 min 19.31 s
128 x 128	16384 x 16384	24 min 56.89 s

However, we must bear in mind that when a direct algorithm cannot find the exact solution it has to approximate it, which may cause visible reconstruction artifacts in the recovered image. Iterative algorithms approach the correct solution using multiple iteration steps, which allows obtaining a better reconstruction at the cost of a higher computation time. The objective of this implementation is to get a significant speed-up while maintaining the results obtained with iterative methods.

All the executions have been done in a NVIDIA TESLA C1060 card, with a 4.0 CUDA Driver and 4.0 CUDA Runtime versions.

5.1.1. Algorithms taken into consideration

There are many possible approaches to calculate the inverse of a matrix, and it's up to the programmer to come up with its own methods. The ones implemented in this project are the following:

1. Use the Gauss – Jordan elimination method to find the inverse matrix directly.

$$[AI] \rightarrow \ A^{-1}[AI] \rightarrow [IA^{-1}] \tag{6}$$

2. Decompose the input matrix with Cholesky decomposition, find the inverse of the Cholesky matrix Q taking advantage that it´s a triangular matrix and finally calculate the inverse of A with matrix multiplication:

$$A = Q \cdot Q^T \rightarrow A^{-1} = (Q \cdot Q^T)^{-1} = (Q^T)^{-1} \cdot Q^{-1} \tag{7}$$

In the following points of the report each of them will be explained, compared to MATLAB's built in functions and given an analysis about their performance, accuracy and possible improvements.

The Gauss-Seidel algorithm has also been included in the final comparison between methods to invert a matrix, but since it has not been developed during the elaboration of this project it there's no explanation of it. Its whole code, as well as for the rest of algorithms, can be found in the appendix of the project.

5.2 Algorithms developed

5.2.1 Gauss-Jordan elimination

The Gauss-Jordan elimination method is a variation of Gaussian elimination. On one hand, Gaussian elimination places zeros below each pivot (diagonal element) in the matrix by performing elementary row operations, starting from the top left element and working downwards. On the other hand, Gaussian-Jordan goes a step further: it places zeros above and below each pivot, leaving the matrix in a reduced row echelon form. Every matrix has a reduced row echelon form, and Gauss-Jordan method is guaranteed to find it.

A matrix is in reduced row echelon form if it meets these two features:

- All rows with at least one element different from zero are above any rows of all zeros
- The leading coefficient of a row – that is, the first number different to zero from the left – is 1 and always strictly to the right of the leading coefficient of the row above it.

Since we are going to deal with invertible matrices – thus, square matrices – all rows will have a leading coefficient different from zero.

An example of matrices in row echelon and reduced row echelon form would be:

Row echelon form: $\begin{bmatrix} 1 & a_{12} & a_{13} & b_1 \\ 0 & 5 & a_{23} & b_2 \\ 0 & 0 & 2 & b_3 \end{bmatrix}$

Reduced row echelon form: $\begin{bmatrix} 1 & a_{12} & a_{13} & a_{14} & b_1 \\ 0 & 1 & a_{23} & a_{24} & b_2 \\ 0 & 0 & 0 & 1 & b_3 \end{bmatrix}$

Gauss-Jordan can be applied on a square matrix to find its inverse. In order to do so, the square matrix must be augmented with the identity matrix of the same dimensions, and then apply the following matrix operators:

$$[AI] \rightarrow A^{-1}[AI] \rightarrow [IA^{-1}] \tag{8}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 0 \\ 0 & 1 & 2 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 & 1 & 0 & 0 \\ 4 & 5 & 0 & 0 & 1 & 0 \\ 0 & 1 & 2 & 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & \frac{5}{6} & \frac{-1}{6} & \frac{-5}{2} \\ 0 & 1 & 0 & \frac{-4}{3} & \frac{1}{3} & 2 \\ 0 & 0 & 1 & \frac{2}{3} & \frac{-1}{6} & \frac{-1}{2} \end{bmatrix}$$

This application only finds the inverse if the matrix is non-singular, that is, if and only if the identity matrix can be obtained using only elementary row operations. Otherwise, the matrix is non-invertible. Also, it must be mentioned that this method can be used to solve a linear equation by augmenting the equation matrix with the result matrix instead of the identity matrix. Nevertheless, it is not in the scope of this project the implementation of that feature.

5.2.1.1 CUDA implementation

The Gauss-Jordan elimination method consists in 3 operations that are used recursively from the top row to the bottom of the matrix. These operations are:

1. Switch rows if the pivot of the actual row is zero.

   a. Copying the corresponding column of the matrix to the CPU to check if the pivot value is 0 or not.
   b. In the case it is zero, find the first next value of the column that is not.
   c. Switch the previous pivot row with the one matching the index of the new nonzero pivot value.
2. Normalize the pivot row so the pivot value is 1.
3. Create zeros above and below the pivot row with elementary row operations.

There is an explanation of the need to copy the column to CPU for each row: checking if the pivot value is 0 or not is much faster on a CPU than in a GPU, especially for big-sized matrices. Also, it would only need one thread to operate on a GPU, wasting a lot of time in something not parallelizable at all.

The parallelized parts of this algorithm are the switching rows process, the normalization of a row and the update of the matrix when creating zeros. Hence, three kernels have been developed to run this algorithm in CUDA:

Kernel 1: Given two indexes *j* and *k*, and a matrix A, it switches the values of the two rows in a matrix so that

$$A_{ki} = A_{ji} \quad , \quad A_{ji} = A_{ki} \ \forall i$$

Table 5.2. Gauss-Jordan elimination row switching kernel code.

```
__global__ void switchRows( float *matrix,float *result, int index,  int rowToSwitch, int lda ) {
```

```
    int y = threadIdx.y + LINEAR_BLOCK_SIZE * blockIdx.y;
    float tmp_m, tmp_r;

    if ( y < lda ){
        tmp_m = matrix[ IDC2D( index, y, lda ) ];
        matrix[ IDC2D( index, y, lda ) ] = matrix[ IDC2D( rowToSwitch, y, lda ) ];
        matrix[ IDC2D( rowToSwitch, y, lda ) ] = tmp_m;
        tmp_r = result[ IDC2D( index, y, lda ) ];
        result[ IDC2D( index, y, lda ) ] = result[ IDC2D( rowToSwitch, y, lda ) ];
        result[ IDC2D( rowToSwitch, y, lda ) ] = tmp_r;
    }
}
```

Here is an example of row switching with and augmented 3x3 elements matrix. It is supposed that the first row has already been computed.

Example 5.1. Example of the row switching method in Gauss-Jordan elimination

The first row has already been computed, leaving the matrix:

$$\begin{bmatrix} 1 & 3 & 2 & 1 & 0 & 0 \\ 0 & 0 & 5 & 3 & 1 & 0 \\ 0 & 7 & 9 & 2 & 0 & 1 \end{bmatrix}$$

Copy the 2nd column and find the first non-zero element in the CPU.

$$\begin{bmatrix} 3 \\ 0 \\ 7 \end{bmatrix}$$

The first nonzero element is 7, so the 2nd and 3rd rows must be switched.

$$\begin{bmatrix} 1 & 3 & 2 & 1 & 0 & 0 \\ 0 & 7 & 9 & 2 & 0 & 1 \\ 0 & 0 & 5 & 3 & 1 & 0 \end{bmatrix}$$

Kernel 2: Given an index *k* and a matrix A, it normalizes the k-th row by the pivot value of the row.

Table 5.3. Gauss-Jordan elimination pivot row normalization kernel code.

```
__global__ void normalizePivotRow(  float *matrix, float *result, int index, int lda ) {
    // Position of each thread inside the block
       int ty = threadIdx.y;
    // Position of each thread inside the matrix
       int y = ty + LINEAR_BLOCK_SIZE * blockIdx.y;
    // Pivot value of the row
       __shared__ float pivotValue;

       if ( y < lda ) {
          if ( ty == 0 )    // First thread of each block loads pivotValue
             pivotValue = matrix[ IDC2D( index, index, lda) ];
          __syncthreads();

       // Every thread divides the element of its position by pivotValue
          matrix[ IDC2D( index, y, lda )] /= pivotValue;
          result[ IDC2D( index, y, lda )] /= pivotValue;
      }
}
```

Here is an example of the row normalization process.

Example 5.2. Example of the row normalization process in Gauss-Jordan elimination

$$\begin{bmatrix} 1 & 3 & 2 & 1 & 0 & 0 \\ 0 & 7 & 9 & 2 & 0 & 1 \\ 0 & 0 & 5 & 3 & 1 & 0 \end{bmatrix} \rightarrow \textit{Row normalization} \rightarrow \begin{bmatrix} 1 & 3 & 2 & 1 & 0 & 0 \\ 0 & 1 & 9/7 & 2/7 & 0 & 1/7 \\ 0 & 0 & 5 & 3 & 1 & 0 \end{bmatrix}$$

Kernel 3: Given a matrix A, it creates zeros above and below the pivot row by subtracting said row conveniently to all the rest of the matrix's rows.

Table 5.4. Gauss-Jordan elimination zero-maker kernel code.

```
__global__ void linearMge(  float *matrix, float *result, int index, int lda ) {

    int ty = threadIdx.y;
    int x =  blockIdx.x;
    int y = ty + blockIdx.y * LINEAR_BLOCK_SIZE;
    __shared__ float multColumn[ LINEAR_BLOCK_SIZE ];
    __shared__ float matrixPivotValue;
    __shared__ float matrixRow[ LINEAR_BLOCK_SIZE ];
    __shared__ float resultPivotValue;
    __shared__ float resultRow[ LINEAR_BLOCK_SIZE];
    float newMatrixValue;  float newResultValue;

    if ( y < lda ) {
       // Each block loads the value of the pivot Row to be substracted
        if ( ty == 0 ){
          matrixPivotValue = matrix[ IDC2D( index, x, lda )];
          resultPivotValue = result[ IDC2D( index, x, lda )];
        }

        multColumn[ ty ] = matrix[ IDC2D( y, index, lda )];
        matrixRow[ ty ] = matrix[ IDC2D( y, x, lda )];
        resultRow[ ty ] = result[ IDC2D( y, x, lda )];
        __syncthreads();

        if ( y!= index ) {
            newMatrixValue = matrixRow[ty] - multColumn[ty] * matrixPivotValue;
            newResultValue = resultRow[ty] - multColumn[ty] * resultPivotValue;
            // Copy to the matrix
             matrix[ IDC2D( y, x, lda) ] = newMatrixValue;
             result[ IDC2D( y, x, lda) ] = newResultValue;
          }
    }
}
```

Here is an example of the execution of the kernel for the second row:

Example 5.3. Example of Gauss-Jordan elimination 3rd kernel execution

$$\begin{bmatrix}1 & 3 & 2 & 1 & 0 & 0\\ 0 & 1 & 3 & 2 & 0 & 1\\ 0 & 2 & 5 & 3 & 1 & 0\end{bmatrix} \rightarrow Multiplier\ Column \begin{bmatrix}-3\\ 0\\ -2\end{bmatrix} \rightarrow \begin{bmatrix}1 & 0 & -7 & -5 & 0 & -3\\ 0 & 1 & 3 & 2 & 0 & 1\\ 0 & 0 & -1 & -1 & 1 & -2\end{bmatrix}$$

As it can be inferred from the code, there are two matrices instead of an augmented matrix: one for the input matrix and the other one for the result of the algorithm (and whose result is initially the identity matrix). It is programmed this way for simplicity as to keep track of the values to be changed and to be able to use the macro IDC2D (see point 4.3 for more information).

The complete code for this implementation can be found in the appendix of this project.

5.2.1.2 Performance evaluation

The time performance of both Gauss-Jordan method (implemented in CUDA) and MATLAB's inversion function can be seen in next figure, and the individual processes of Gauss-Jordan in the later table:

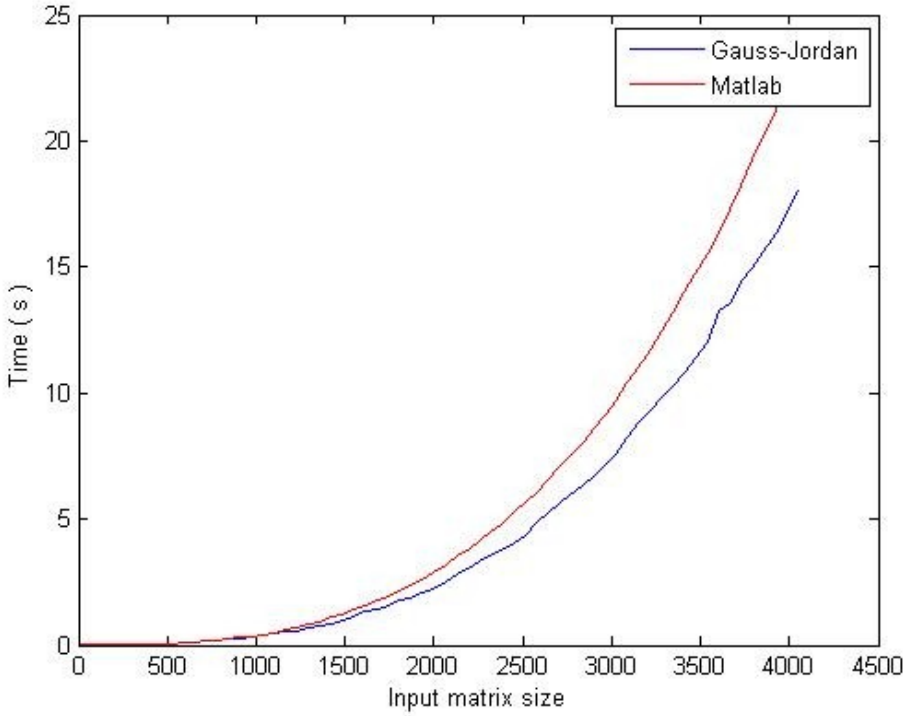

Figure 5.1. Execution time comparison between Gauss-Jordan elimination and MATLAB's inverse function

Table 5.5. Mean execution times and occupancy of the different parts of the Gauss-Jordan elimination algorithm.

Operation	Mean execution time	Occupancy
Column copy back to host	13.04 us	-
Switching rows (kernel 1)	102.35 us	1
Normalize the pivot row (kernel 2)	78.33 us	1
Update the matrix to create zeros (kernel 3)	4.607 ms	1

Assuming that all the operations are done for all the rows of the matrix - which is not necessarily true for the row switching kernel - the bottleneck of the algorithm is the third kernel. This kernel, which subtracts the pivot row to all of the rest of the matrix, spends approximately the 95.96% of the time spent by the algorithm in the CUDA card.

The occupancy listed for each kernel is a parameter that evaluates the efficiency of the use of the CUDA card resources (mainly based in register and threads per block usage). This parameter takes values between 0 and 1, being 1 the best usage possible. An occupancy of 1 for all the kernels means that there are not bad-dimensioned kernels – regarding threads per block, warps, and registers available for each block. However, occupancy of 1 does not imply in any way that the kernel executes fast nor slow.

The main problem of this third kernel is that it was designed to work with linear, vertical blocks to exploit that all the values in the same column have the same pivot row value. However, each thread has to access to the input and result matrices twice in a not-coalesced way (which makes memory accesses much slower than they should be).

As a conclusion, it can be said that this implementation is unsuccessful. The execution time is not as low as expected, and the fact that CUDA can only compute the inverse with single float variables makes MATLAB's built in function the better choice. However, this doesn't mean that this method is a bad mean to find an inverse – nor to solve a linear system if slightly modified; this only implies that, if there is a way, we haven't found it yet.

5.2.2. Cholesky decomposition

The Cholesky decomposition was named after its discoverer, Andre-Louis Cholesky, and it consists in the decomposition of a hermitian, positive-definite matrix into the product of a lower triangular matrix Q and its conjugate transpose

$$A = Q \cdot Q^{H} \tag{9}$$

When it is applicable, this decomposition is twice as efficient as the LU decomposition for solving systems of linear equations. Also, this decomposition is unique: given an input matrix A that is, in fact, an hermitian and positive-definite matrix, there is one and only one lower triangular matrix Q with strictly positive diagonal entries. To obtain the Cholesky decomposition of any matrix, we simply have to equal both sides of the equation to obtain:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} q_{11} & 0 & 0 \\ q_{21} & q_{22} & 0 \\ q_{31} & q_{32} & q_{33} \end{bmatrix} \cdot \begin{bmatrix} q_{11}^{*} & q_{21}^{*} & q_{31}^{*} \\ 0 & q_{22}^{*} & q_{32}^{*} \\ 0 & 0 & q_{33}^{*} \end{bmatrix} \tag{10}$$

$$q_{ii} = \sqrt{\left(a_{ii} - \sum_{k=1}^{i-1} q_{ik}^{2}\right)} \tag{11}$$

$$q_{ji} = \frac{\left(a_{ji} - \sum_{k=1}^{i-1} q_{jk} q_{ik}\right)}{q_{ii}} \text{ for } j > i \tag{12}$$

All the values under the square root are always positive because A is symmetric and positive definite, and all the elements of Q are real.

5.2.2.1 CUDA implementation

As it can be observed in the previous point, the Cholesky decomposition is a recursive algorithm. To obtain the diagonal values q_{ii} all the previous column values *qji* must be calculated, and vice versa, to get the new column values all the previous values (diagonal and column ones) must be known.

That doesn't mean there is no way of exploiting any parallelism the algorithm has. The calculus of each qji element is independent from all the elements in its column excepting the diagonal elements. Furthermore, the summations required to calculate the coefficients can be done in several separated steps (uploading a coefficient of the matrix with the known values so far), so it's perfectly possible to update a coefficient of the Q matrix whenever some other values that affect said coefficient are final.

Four kernels are needed to deal with this algorithm:

Kernel 1 (topleft): Calculates the qii and qij elements of a diagonal block of the matrix. These values are final, and need no further computation.

Table 5.6. Cholesky decomposition topleft kernel code.

```
__global__ void topleft( float *matrix, int boffset, int mat_size ) {
        int tx = threadIdx.x;
        int ty = threadIdx.y;
         __shared__ float topleft[BLOCK_SIZE][BLOCK_SIZE];

        // Load the proper diagonal block
        topleft[ty][tx] = matrix[IDC2D(ty + BLOCK_SIZE*boffset,
                tx + BLOCK_SIZE*boffset ,mat_size)];
        __syncthreads();
```

```
        float fac;
         for( int k = 0; k < BLOCK_SIZE; k++ ) {
                  __syncthreads();

                 fac = rsqrtf( topleft[k][k] );
                 __syncthreads();

                 if ( ( ty == k ) && ( tx >= k ) )
                                  topleft[tx][ty] = (topleft[tx][ty])*fac;
                 __syncthreads();
                 if ((ty>=tx)&&(tx>k))
                  topleft[ty][tx] = topleft[ty][tx] - topleft[tx][k] * topleft[ty][k];

        } // For loop end

        __syncthreads();
        if ( ty >= tx )   // Update of the lower triangle of the block
                 matrix[ IDC2D(ty+BLOCK_SIZE*boffset,tx+BLOCK_SIZE*boffset,mat_size)] =
                                          topleft[ty][tx];

        __syncthreads();
}
```

Kernel 2 (strip): Calculates all the values of the strip of blocks below the last block computed by the topleft kernel, taking profit of its results. As with the previous kernel, these updated values are final.

Table 5.7. Cholesky decomposition strip kernel code.

```
__global__ void strip (float *matrix, int blockoffset, int mat_size ){

        // blockoffset labels the topleft position
        int boffx = blockIdx.x + blockoffset + 1; // working position
        int tx = threadIdx.x;
        int ty = threadIdx.y;
```

```
        __shared__ float topleft[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float workingmat[BLOCK_SIZE][BLOCK_SIZE];

        topleft[ty][tx] = matrix[ IDC2D(ty + blockoffset * BLOCK_SIZE, tx +
                blockoffset * BLOCK_SIZE, mat_size) ];

        // Read the working block (already transposed)
        workingmat[tx][ty] = matrix[ IDC2D( ty+boffx*BLOCK_SIZE,
                tx+blockoffset*BLOCK_SIZE, mat_size ) ];
        __syncthreads();

        // Forward-substitution for the new strip-elements
        if( ty == 0 ) {
                for ( int k = 0; k < BLOCK_SIZE; k++ ) {
                        float dotprod=0.f;
                        for ( int m = 0; m < k; m++ )
                                dotprod += topleft[k][m] * workingmat[m][tx];
                        workingmat[k][tx] = (workingmat[k][tx] - dotprod)/topleft[k][k];
                } // for loop end
        }// if clause end

   __syncthreads();

// we have the result transposed, we undo it when copying to the matrix
  matrix[ IDC2D( ty+boffx*BLOCK_SIZE, tx+blockoffset*BLOCK_SIZE, mat_size) ] =
        workingmat[tx][ty];
   __syncthreads();
}
```

Kernel 3 (diagupdate): Updates the values (summations) of the elements in the diagonal blocks with the elements found by the strip kernel.

Table 5.8. Cholesky decomposition diagonal update kernel code.

```
__global__ void diagupdate ( float *matrix, int blockoffset, int mat_size ) {
        int boffx = blockIdx.x+blockoffset+1;
        int tx = threadIdx.x;
        int ty = threadIdx.y;
```

```
        __shared__ float left[BLOCK_SIZE][BLOCK_SIZE];
        left[ty][tx] =
             matrix[ IDC2D(ty+boffx*BLOCK_SIZE, tx+blockoffset*BLOCK_SIZE, mat_size)];
        __syncthreads();

        float matrixprod=0.f;
        if ( ty >= tx ) {
                for ( int kk = 0; kk < BLOCK_SIZE; kk++ )
                        matrixprod+=left[ty][kk]*left[tx][kk];
                matrix[ IDC2D( ty+boffx*BLOCK_SIZE, tx+boffx*BLOCK_SIZE, mat_size ) ]-=
                        matrixprod;
        }
        __syncthreads();
}
```

Kernel 4 (loupdate): Updates all the rest of values of the matrix with the results obtained by the strip kernel.

Table 5.9. Cholesky decomposition low update kernel code.

```
_global__ void loupdate(  float *matrix, int blockoffset, int mat_size, int mat_blocks ) {
        int tx = threadIdx.x;
        int ty = threadIdx.y;

        int boffy = blockIdx.y+blockoffset+1;
        int boffx = boffy+1;

        // Left matrix is the matrix of the strip
        __shared__ float left[BLOCK_SIZE][BLOCK_SIZE];

        // Upt matrix
        __shared__ float upt[BLOCK_SIZE][BLOCK_SIZE];

        // Reading the data
        int tmpx,tmpy,tmpb;
        tmpy = boffy * BLOCK_SIZE;
        tmpb = blockoffset*BLOCK_SIZE;

        upt[ty][tx] = matrix[ IDC2D( ty + tmpy, tx + tmpb, mat_size) ];
```

```
        for (;boffx<mat_blocks;boffx++){

                tmpx = boffx * BLOCK_SIZE;
                left[ty][tx] = matrix[ IDC2D( ty + tmpx, tx + tmpb, mat_size) ];
                __syncthreads();

                float matrixprod=0.f;
                for (int kk=0;kk<BLOCK_SIZE;kk++)
                        matrixprod+=left[ty][kk]*upt[tx][kk];
                __syncthreads();

                matrix[ IDC2D( ty + tmpx, tx + tmpy, mat_size) ]-=matrixprod;
        }

}
```

In order to do the decomposition the matrix is first divided in square blocks of 16 x 16 elements each. Afterwards, these four steps are repeated from the top-left corner to the bottom-right corner, as long as there are blocks left.

- Compute the diagonal block with the *topleft* kernel.
- Compute the values below the previous block with the *strip* kernel.
- Update the diagonal blocks with the *diagupdate* kernel.
- Update the rest of the matrix blocks with the *loupdate* kernel.

Here is an example of how the decomposition of a 5x5 blocks matrix (80x80 elements) would be computed:

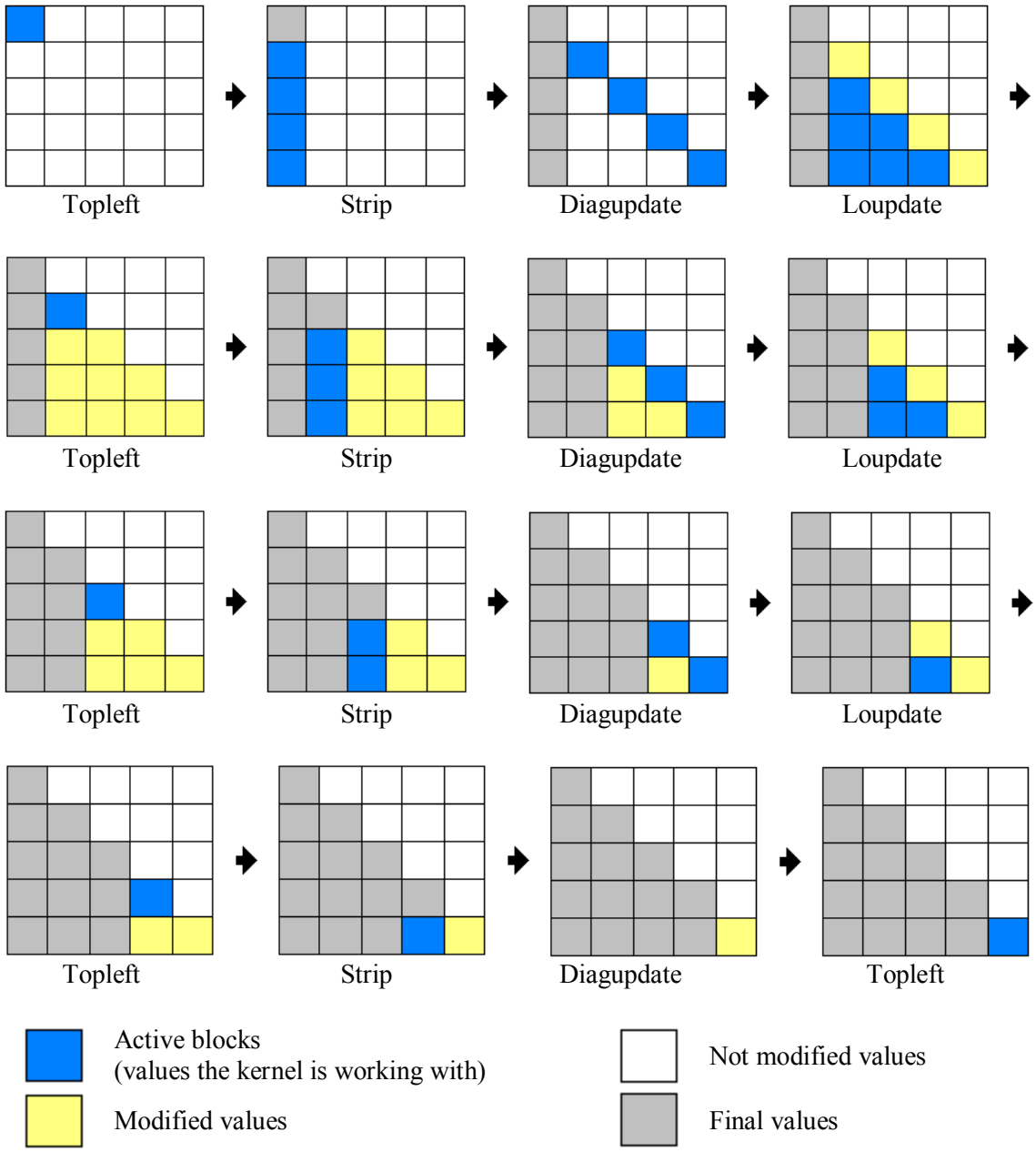

Figure 5.2. Example of kernel interaction to compute the Cholesky decomposition

The whole code and implementation of the method is included in the appendix of this report.

5.2.2.2 Performance evaluation

MATLAB has its own Cholesky decomposition built-in function that can be used for this same purpose, instead of a CUDA implementation of the method. The execution times for both methods are shown below:

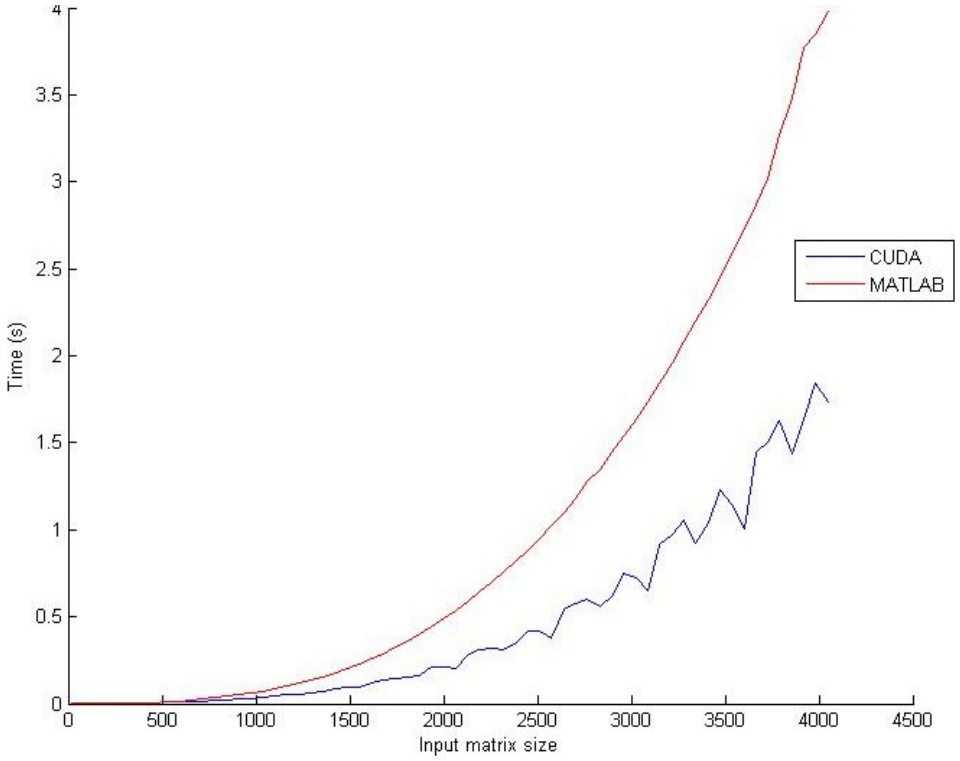

Figure 5.3. Execution times of both MATLAB's and CUDA Cholesky decomposition functions.

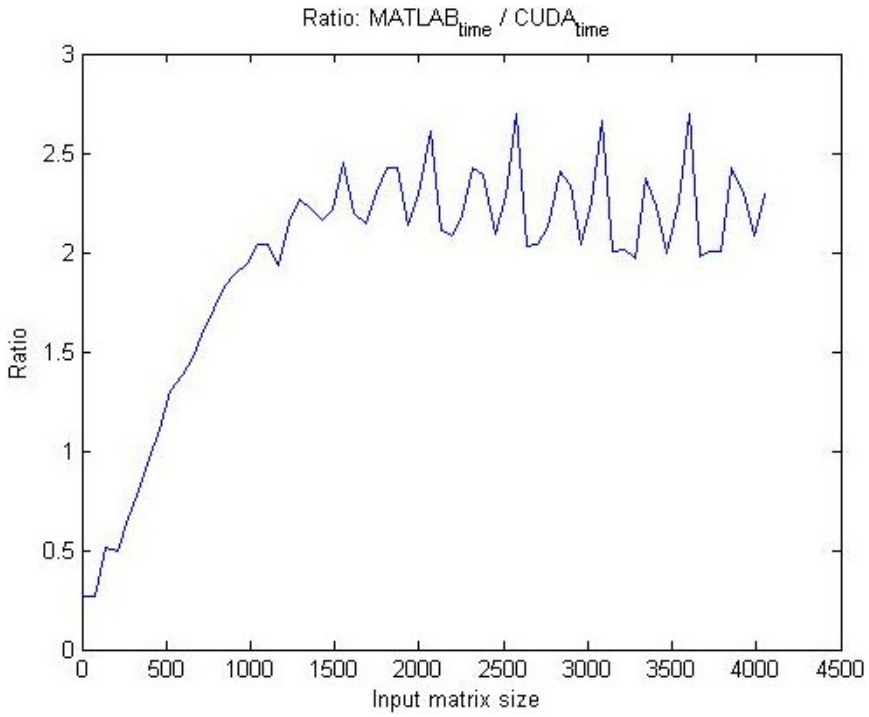

Figure 5.4. Ratio between MATLAB's and CUDA´s Cholesky decomposition functions execution times

There is some improvement from MATLAB's execution times, but the speed-up only goes up to twice as fast in most cases. In order to analyze this performance, the code has been run in CUDA profiler. The mean execution time and occupancy of each kernel that the profiler returns are the following:

Table 5.10. Mean execution times and occupancy of CUDA Cholesky decomposition kernels

Kernel	Mean execution time	Occupancy
Topleft	18.45 us	0.25
Strip	175.51 us	1
Diagupdate	54.78 us	1
Loupdate	6.316 ms	1

The two relevant values of this table are these: topleft kernel's occupancy and loupdate kernel's mean execution time. Let's analyze the reasons behind both of them.

The topleft kernel is an adaptation of the typical C/C++ code in the sense that it's a recursive function. In the first iteration it uses one thread to calculate one coefficient, and then synchronizes all threads; in the second one it uses two more threads for two more coefficients, and then synchronization again; and so it goes on until the end of the block. This means that a little less than the half of the threads launched in the block aren't used at all – the exact number is 120 out of 256, a 46.88 %of them. The code could be rewritten to use only as many threads as needed, but the low occupancy of this kernel isn't the major problem the algorithm has, as it can be seen in the next figure.

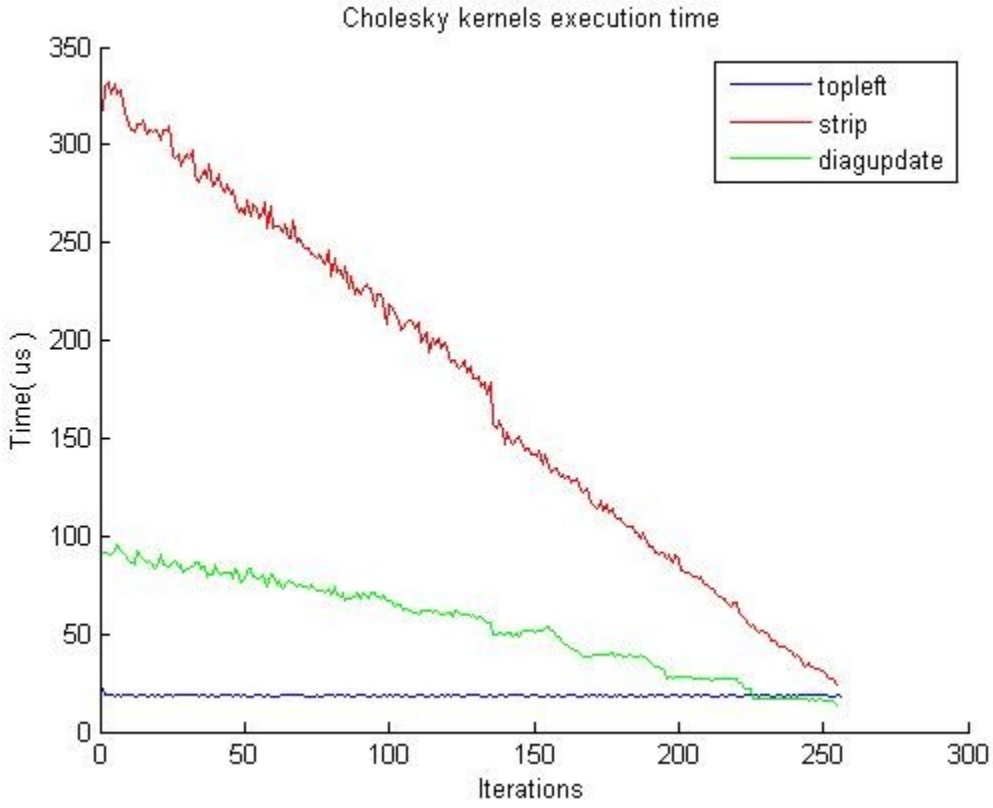

Figure 5.5. Execution times comparison between topleft, strip and diagupdate kernels of Cholesky decomposition.

Even if we were to optimize the execution of the topleft kernel it is, by far, the least time consuming kernel in most of the iterations of the algorithm, so the global time of the algorithm wouldn't be greatly reduced.

We can also see the strip and diagupdate execution times in the previous graph. Each iteration both of them work with the same amount of blocks, but the diagupdate kernel works faster. This is caused by the fact that diagupdate has more parallelism: it can calculate all the new values of the diagonal block at the same time, while the strip kernel can't and has to follow a similar pattern of that used in the topleft kernel.

Regarding the loupdate kernel's execution time, it spends way more time that it seems from its mean.

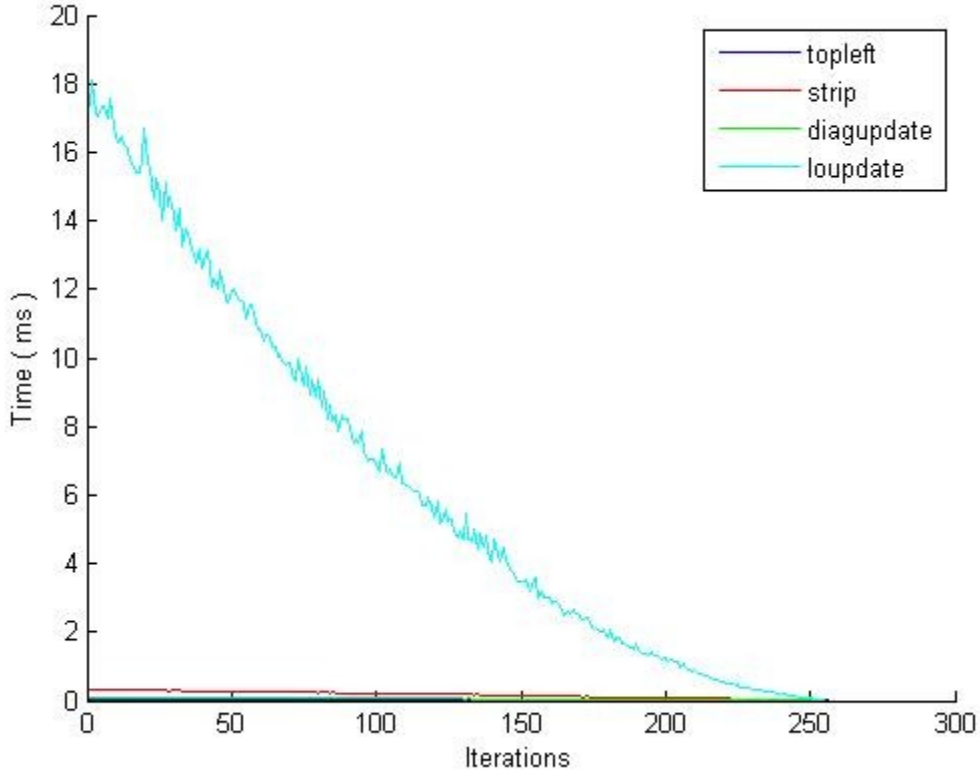

Figure 5.6. Execution time comparison between all kernels of Cholesky decomposition.

As it appears clearly in the previous graph, the fourth kernel is the real bottleneck of the algorithm. This is caused because the kernel updates the lower blocks in a recursive way instead of being all simultaneous:

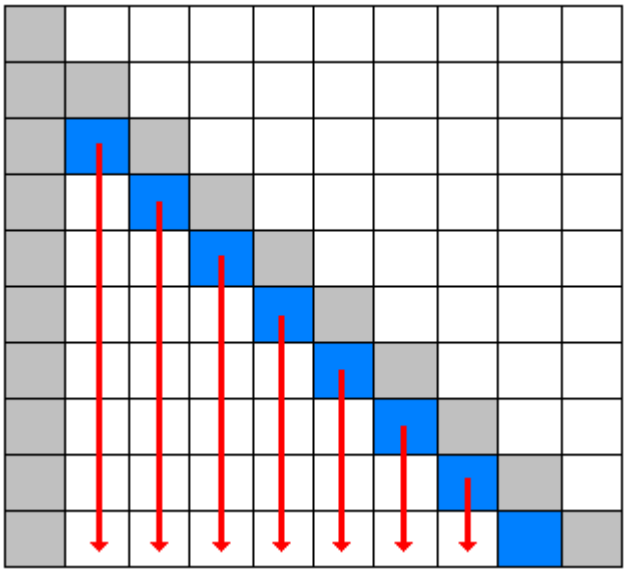

Figure 5.7. Cholesky decomposition's loupdate kernel recursive block update. (Blue blocks indicate the first blocks updated)

To put it into a mathematic notation, if the matrix dimension is N blocks, and the time of updating a single block is T_{single} seconds, the total amount of time T_{update} to do an update of all the matrix's blocks is

$$T_{update} = T_{single} \sum_{i=1}^{N-2} i \tag{13}$$

As for the total time spent during the whole algorithm, it could be approximated by:

$$T_{total\ loupdate} = \sum_{j=1}^{N-2} T_{update}^{j} = T_{single} \sum_{j=1}^{N-2} \sum_{i=1}^{N-1-j} i \tag{14}$$

which grows exponentially as the input matrix increases. Of course, this assumption is done considering that all the blocks spend the same time updating one

block, which might be not true, but it gives an overall view about the execution time of this kernel.

The ideal way to handle this update would be launching as many thread-blocks in the grid as blocks to be updated, but a CUDA limitation would be met: there mustn't be more than 65535 blocks in a grid. For example, a 4096 x 4096 square matrix will have 32896 blocks to be updated in the first iteration, but if we are going to work with larger matrixes, such as 8192 x 8192, a total of 131.328 blocks would be needed at the same time.

A way to handle this setback would be having control from the CPU (outside the kernel) of which blocks have been updated and which have not, and run the kernel with the proper block adjustments as many times as needed until complete the update.

Regarding the accuracy of the method, for a 4096 x 4096 the mean relative error between both CUDA and MATLAB's implementation is 5.71 e-6, which is perfectly acceptable. We must bear in mind that CUDA can only work with single float variable format.

5.2.3 Gaussial Elimination (GETS)

Gaussian elimination is an algorithm for solving systems of linear equations, which can be used to calculate determinants, matrix inverses (Gauss-Jordan elimination) and to find the rank of a matrix. This algorithm gets a matrix into its row echelon form with elemental row operations, which is explained in point 5.2.1. As a reminder, a matrix in row echelon form must meet:

- All rows with at least one element different from zero are above any row of all zeros.
- The leading coefficient of a row is always strictly to the right of the leading coefficient of the row above it.

Again, since we are going to deal with invertible matrices all rows must have a leading coefficient different from zero. If that requirement is not met, the matrix is not invertible.

This algorithm is based in Gaussian elimination, but it takes profit of the fact that the input matrix is a lower triangular matrix - the output from the Cholesky decomposition. Hence, it has been named GETS (Gaussian Elimination Triangular Solver). The GETS algorithm is in a point between Gaussian Elimination and Gauss-Jordan elimination:

- Zeros are only made in the lower triangle of the matrix like in Gaussian elimination, but it takes advantage of the fact that the zeros in the upper triangle of the matrix are already done.

- The input matrix is augmented with an identity of the same size, and the whole matrix is put into reduced row echelon form, like in Gauss-Jordan.

At a first glance it looks like a particular case of Gauss-Jordan elimination, but its CUDA implementation gets a notorious speed up from the implementation seen in point 5.2.1. of this report. This algorithm would also solve triangular linear systems with slight modifications, but doing so is not in the scope of the project.

5.2.3.1 CUDA implementation

GETS' CUDA implementation is a lot simpler and faster than that of Gauss-Jordan. This is due to two main facts: whether the input matrix is invertible or not, and the thread usage to compute the result.

If the input matrix A is invertible its determinant will be different from zero, and so will be the determinant of its Cholesky decomposition Q matrix:

$$\det(A) = \det(Q) \cdot \det(Q^T) = \det(Q)^2$$

Furthermore, it is known that Q is a lower triangular matrix, which means that its determinant is the multiplication of all its diagonal entries; even if there was only one entry equal to zero its determinant would be zero, which cannot be. Therefore, there will not be any diagonal entry equal to zero.

The importance of these values is that they will be the pivot values of the algorithm. Since in the upper triangle of the matrix there is nothing but zeros, the elementary operations done with previous rows will affect neither the pivot values nor values below the next rows. In the next figure there's an example of this independence:

the matrix entries in red are not modified after doing the subtraction of the first row conveniently.

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 2 & 1 & 0 & 0 & 1 & 0 \\ 6 & 2 & 5 & 0 & 0 & 1 \end{bmatrix} \rightarrow Multiplier\ Column \begin{bmatrix} 0 \\ -2 \\ -6 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & -2 & 1 & 0 \\ 0 & 2 & 5 & -6 & 0 & 1 \end{bmatrix}$$

We can conclude, then, that no partial pivoting kernel will be needed, getting a slight speed up.

The second crucial fact in the implementation is the thread usage. There's no need to launch as many threads as elements in the augmented matrix because most of them will remain idle. The way of use them will be launching only as many as elements in the columns of the matrix that are to be modified. In other words, in the first iteration only one column of both input and result matrices will be modified, so only a "column" of threads will be used; in the second iteration, two of them; and so on. The fact that the matrix entries above the diagonal will remain zero has also been taken into account, as shown in figure x.

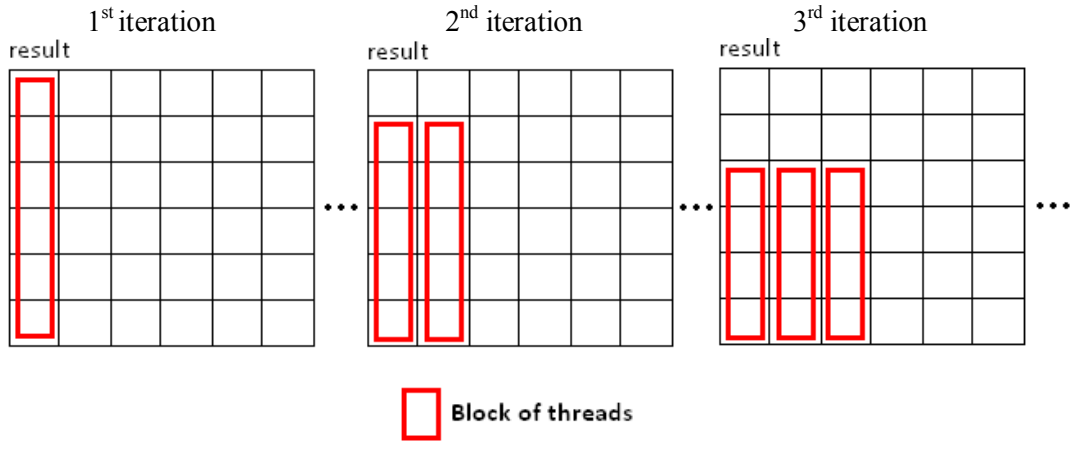

Figure 5.8. Thread management and usage in GETS algorithm

With this set up in mind, only one kernel is needed to do the whole inverse of the Cholesky matrix input. The kernel is called recursively from the CPU as many times as rows the matrix has (once for each row).

Kernel 1 (getsKernel): Given an input matrix, a result matrix and an index k, calculates row elementary operations with the k-th row to create zeros in the k-th column of the input matrix.

Table 5.11. Gaussian elimination (GETS) kernel code.

```
__global__ void getsKernel( float *inputMatrix, float *resultMatrix, int index, int lda ){
        int ty = threadIdx.y;
        int x =  blockIdx.x;
        int y = ty + blockIdx.y * LINEAR_BLOCK_SIZE + index;

        __shared__ float inputMatrixCol[ LINEAR_BLOCK_SIZE ];
        __shared__ float resultMatrixPivotRowValue;
        __shared__ float inputMatrixPivotRowValue;

        if ( y < lda ) {
                if ( ty == 0 ) {
                        resultMatrixPivotRowValue = resultMatrix[ IDC2D( index, x, lda ) ];
                        inputMatrixPivotRowValue = inputMatrix[ IDC2D(index, index, lda ) ];
                }
                inputMatrixCol[ ty ] = inputMatrix[ IDC2D( y, index, lda ) ];
                __syncthreads();

                if ( y!= index )
                        resultMatrix[IDC2D(y,x,lda)]-= inputMatrixCol[ ty ]*
                                resultMatrixPivotRowValue/ inputMatrixPivotRowValue;
                else
                        resultMatrix[ IDC2D( y, x, lda) ] /= inputMatrixPivotRowValue;
        }
}
```

5.2.3.2 Performance evaluation

There is no function in MATLAB that performs the same exact operations that GETS does, but there's one that gives the same results: the inverse of a lower triangular matrix. Here are the execution times of both methods:

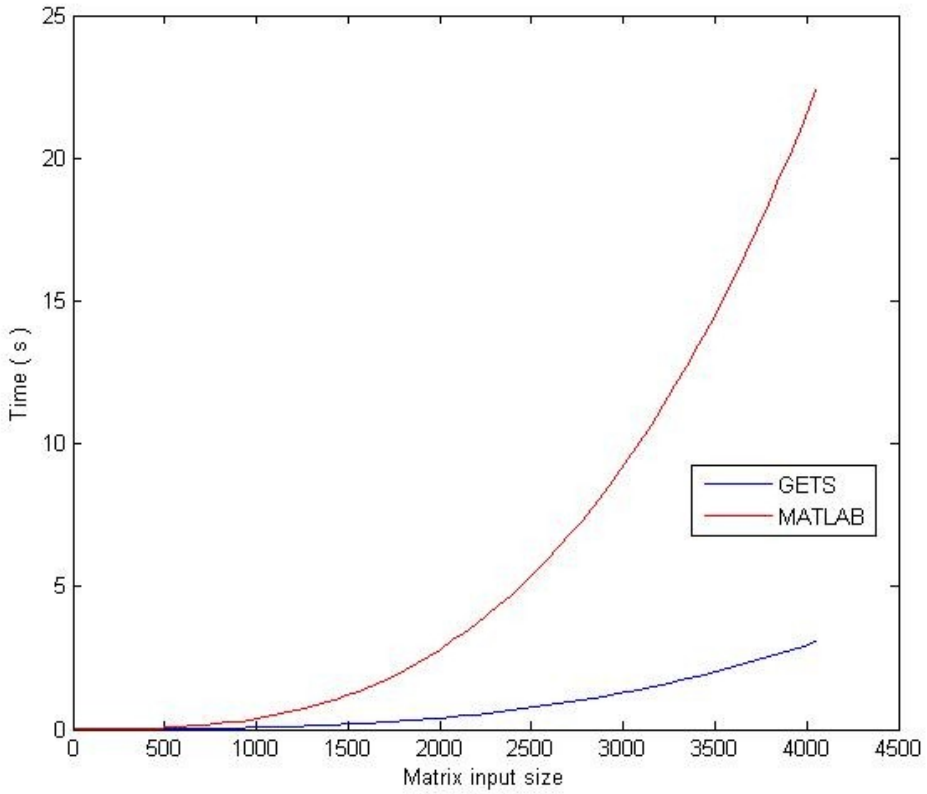

Figure 5.9. Execution time comparison between GETS and MATLAB's function to get an inverse of a triangular matrix

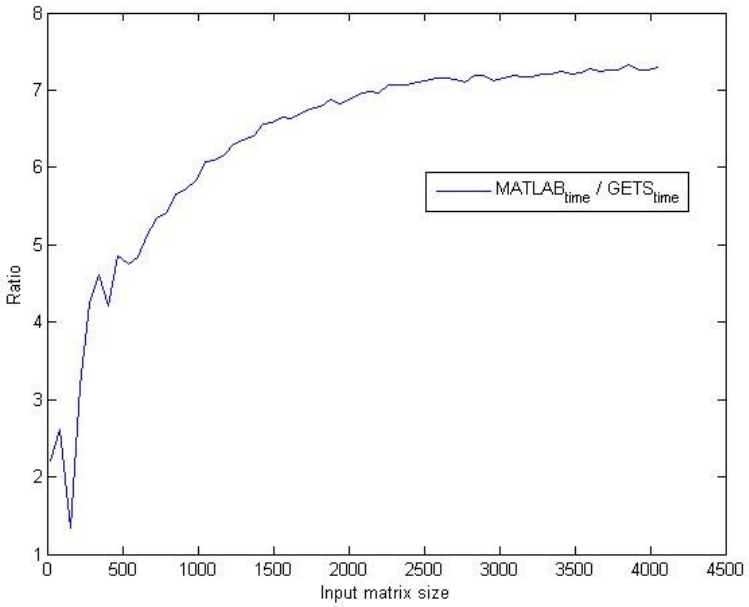

Figure 5.10. Ratio of execution times between MATLAB's and GETS' functions to get an inverse of a triangular matrix.

There's a huge improvement from Gauss-Jordan elimination's performance: for matrix sizes larger than 1000x1000 the algorithm goes at least 6 times faster than MATLAB.

The code has been run in CUDA's profiler as well for a large matrix size (4096 x 4096) and the results obtained are the following:

Table 5.12. Mean execution time and occupancyof GETS and Gauss-Jordan elimination kernels.

Kernel	Mean execution time	Mean occupancy
getsKernel	0.8068 ms	0.9726
LinearMGE (from Gauss-Jordan)	4.607 ms	1

This new algorithm is in mean 5.5 times faster than the one from Gauss-Jordan. However, its occupancy is not always 1, as the mean occupancy is not exactly 1. Let's analyze the reason behind it.

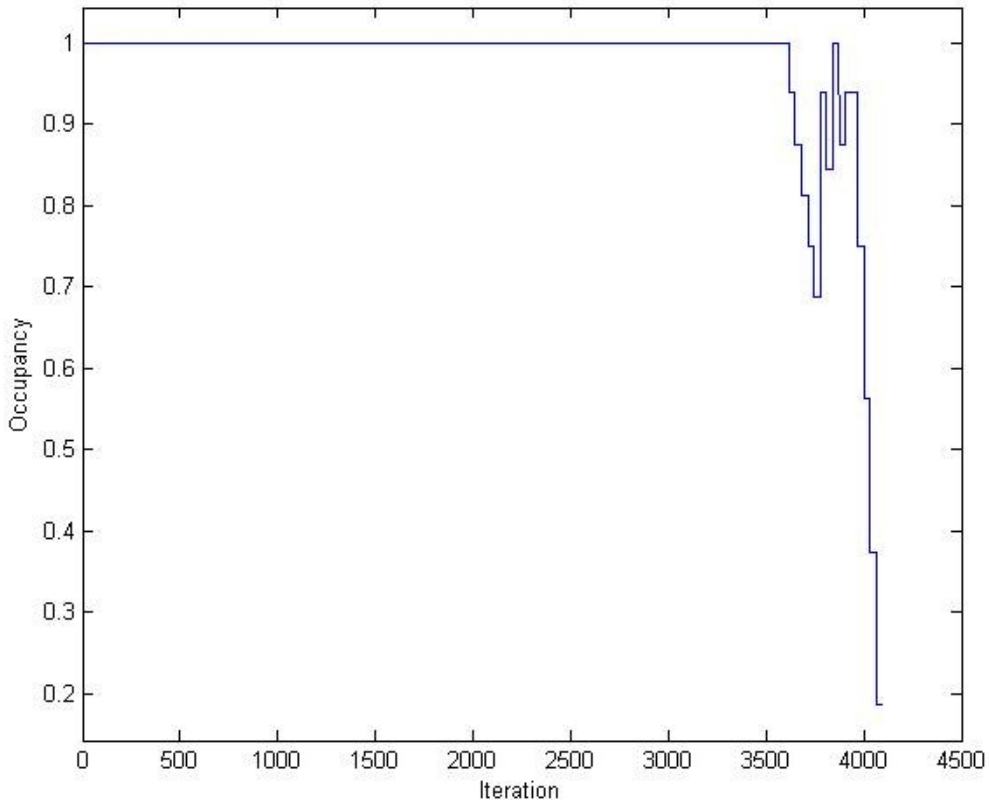

Figure 5.11. GETS algorithm occupancy evolution throughout execution.

As it can be seen, the first downfall is when we reach past the 3500-th iteration approximately. This may be caused because of the thread-block size: this code has been designed to work with linear blocks. This implies that for large matrix sizes there is more than one block needed for column because the maximum size of a block is 512 threads, but the usage of the resources is good.

However, when there are only a few iterations left there are a lot of columns that must be updated, but the elements in those columns are fewer than the maximum of the block - that is, 512 threads. In the worst case, the last iteration, there will be one block for each element to be updated, leading to occupancy of 0.188. That means 511 threads remaining idle while one working, being the same for each of the matrix's columns.

A way to handle that bad occupancy in the latter iterations of the algorithm would be an adaptive control of the threads used from the CPU, since the kernel is launched recursively. This thread control may lead to a minor speed up in later iterations, but there won't be a really significant speed up in the whole time because:

- The latter iterations are less than 1/8 from the total, and
- The most time consuming iterations (the ones where most threads are launched) are around the middle of the matrix, where the occupancy is already 1.

5.2.4 Matrix multiplication

Matrix multiplication is a mathematic operation that involves two input matrices and gives a third one as a result.

$$A \cdot B = C$$

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mN} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{N1} & b_{N2} & \cdots & b_{Nn} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{pmatrix}$$

$$where\ c_{ij} = \sum_{k=1}^{N} a_{ik} \cdot b_{kj} \quad for\ i = 1..m\ , j = 1..m$$

To be able to multiply both matrices the inner dimensions of A and B must agree. The result matrix C has the same rows as A and the same columns as B.

5.2.4.1 CUDA implementation

Two approaches have been considered in this point. The first one is an exact port from a C/C++ code taking advantage of the simultaneous threads, and the second one - extracted from CUDA SDK – takes advantage of coalesced memory accesses and shared memory of each block of threads.

The first approach is pretty basic: as many threads as entries the result matrix has are launched. Each thread has to load the entire row of matrix A and the entire column of matrix B according to their position in the result matrix, and then sum their element by element multiplication.

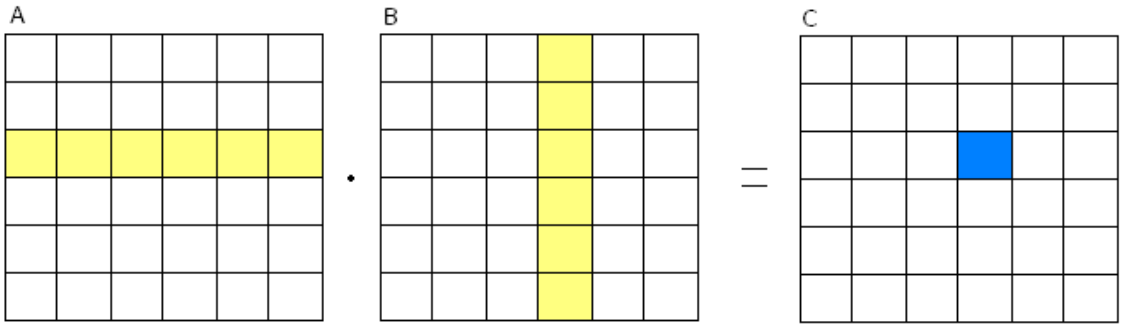

Figure 5.12. Loaded values by a single generic thread in a square matrix multiplication.

Multiplication kernel (1st approach)

Table 5.13. Multiplication kernel code. Direct approach.

```
__global__ void multKernel(  float *input1d, float *input2d, float *outputd,
        int M1, int N1, int M2, int N2) {

        int col = threadIdx.x + blockIdx.x * BLOCK_WIDTH;
        int row = threadIdx.y + blockIdx.y * BLOCK_HEIGHT;
        float outputValue = 0;

        if ( col < N2 && row < M1 ) {
                for ( int k = 0; k < N1; k++ )
                        outputValue += input1d[ row + k*M1 ] *  input2d[ k + col*M2 ];
                outputd[ row + col*N2 ] = outputValue;
        }
}
```

The previous approach has a fatal flaw: for large sized matrices a single thread has to load a lot of values at the same time as the rest of threads, deriving in memory accessing latency issues. Furthermore, most of the loaded values by each thread are needed by a lot of them, so accesses to the input matrices can - and should - be minimized.

That's the goal of the CUDA SDK implementation. Instead of letting each thread load all its needed values, it takes advantage that nearby entries of the result matrix need nearby entries of both input matrices. Hence, all the threads in a block load two values each (one from each input matrix) into the shared memory of the block. This has three main effects:

- The bigger the block, the lesser the number of values each thread has to load, permitting faster calculus of each final value.
- All the values are in the shared memory of the block, permitting much faster access to the values of the threads that need them.
- There are less simultaneous accesses to memory. In fact, the number is reduced by the number of threads each block has (256 in our case, since we are working with 16x16 blocks).

The loading of the values, however, is not done in one sweep. Each block loads as many values as threads in the block, calculates a partial result of the multiplication, and heads for the next needed value. This process is shown in the next figure (assuming 2x2 block size for simplicity):

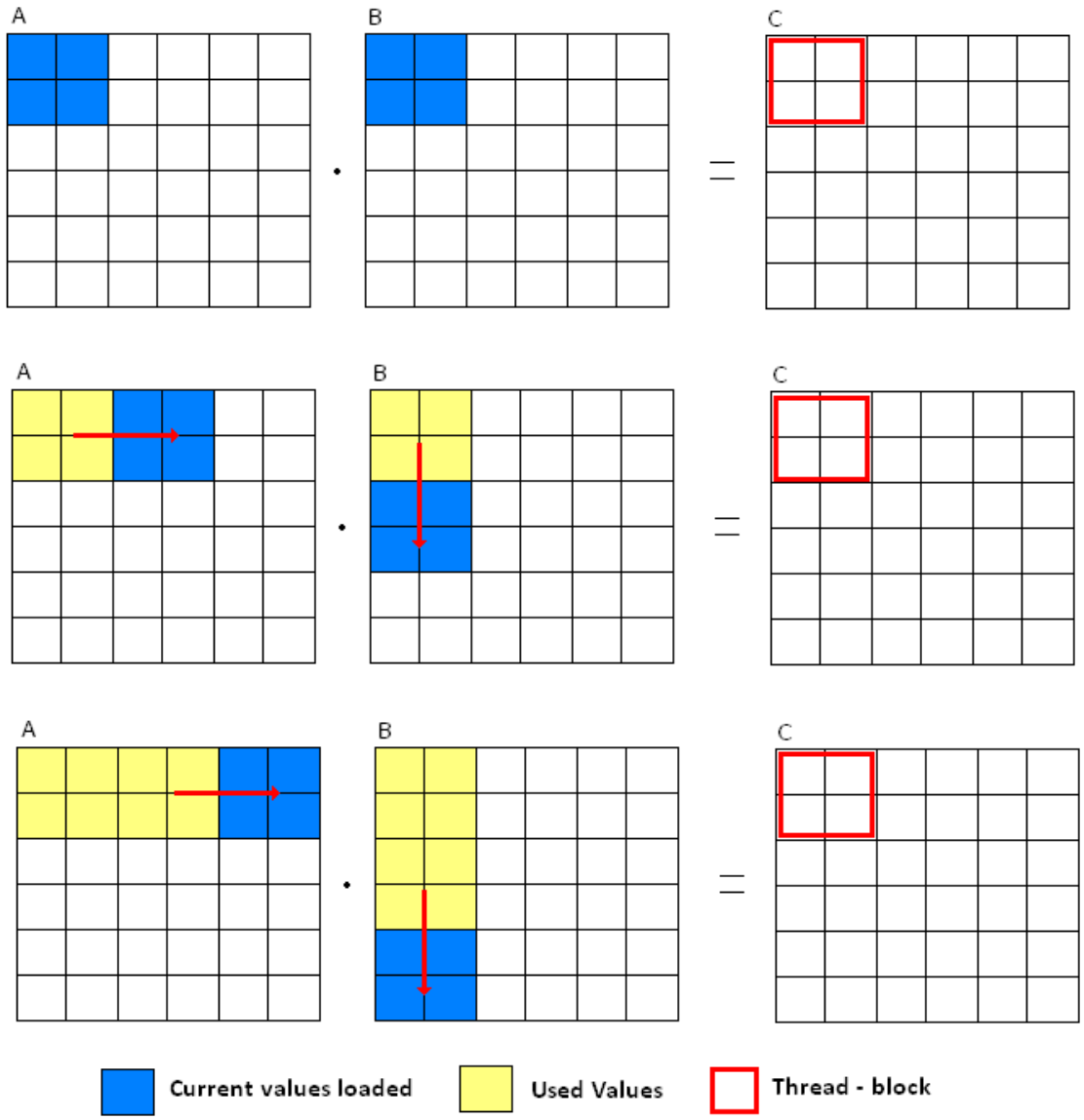

Figure 5.13. CUDA SDK multiplication code's way to load values and perform matrix multiplication.

Multiplication kernel code (2nd approach)

Table 5.14. Multiplication kernel code. CUDA SDK approach.

```
__global__ void matrixMul( float* C, float* A, float* B, int wA, int wB){

    // Block and thread index
    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * wB;

    // Csub is used to store the element of the block sub-matrix that is computed by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B required to compute the block sub-matrix
    for (int a = aBegin, b = bBegin; a <= aEnd; a+= aStep, b += bStep) {

        // Declaration of the shared memory array As used to store the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        // Declaration of the shared memory array Bs used to store the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from device memory to shared memory;
        //each thread loads one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

       // Synchronize to make sure the matrices are loaded
```

```
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];

        // Synchronize to make sure that the preceding computation is done before
        //loading two new sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to device memory; each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

5.2.4.2 Performance evaluation

In this point we evaluate the execution times of MATLAB's matrix multiplication and the two approaches explained previously.

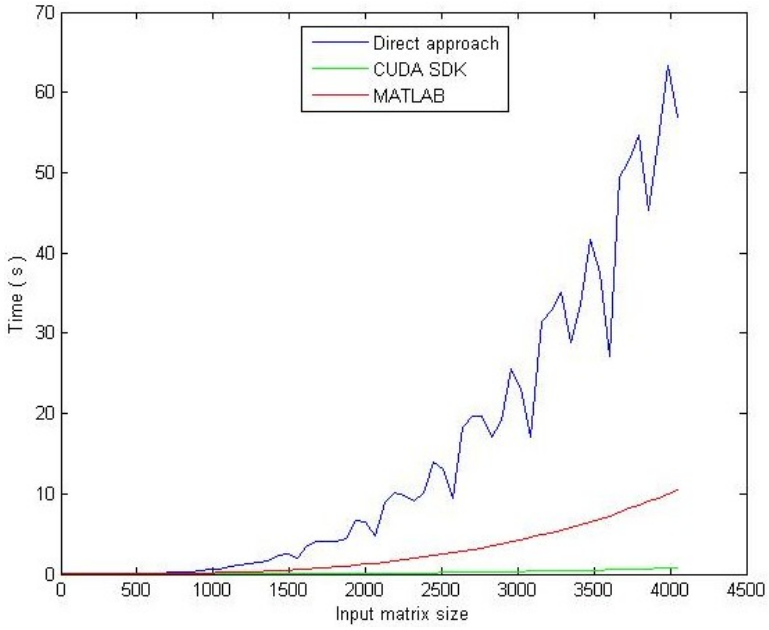

Figure 5.14. Execution time comparison between matrix multiplications algorithms

It can clearly be seen that the direct implementation is nowhere near optimal: it spends way too much time in a simple operation because of the flaws commented before. However, the CUDA SDK approach really improves MATLAB's times significantly.

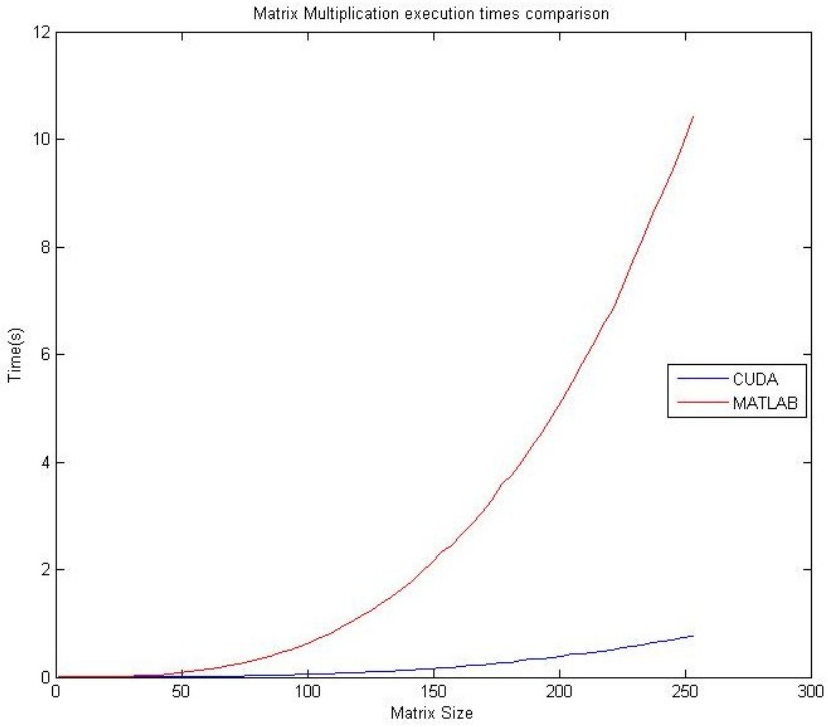

Figure 5.15.Execution times comparison between CUDA SDK and MATLAB's matrix multiplication functions.

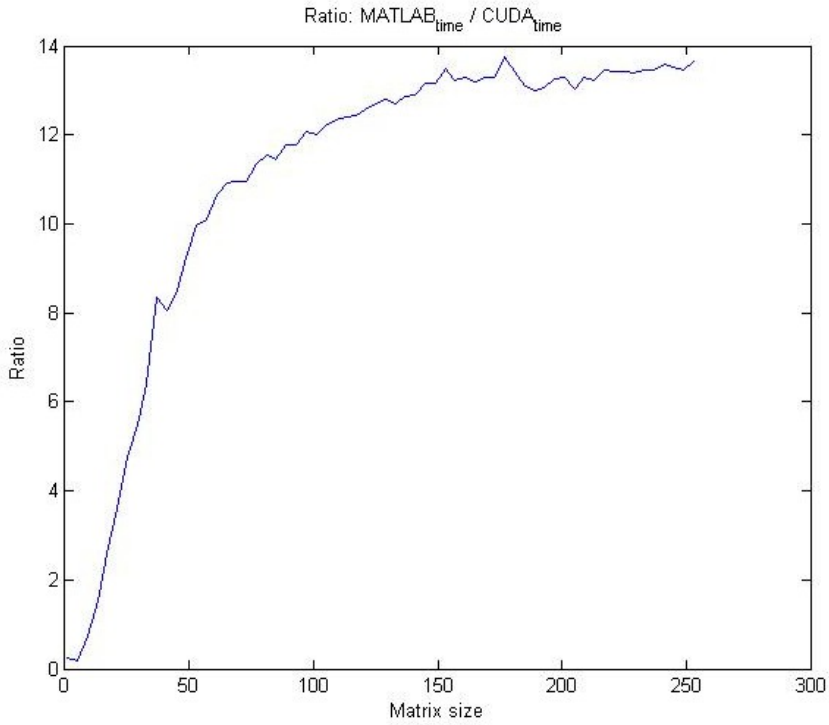

Figure 5.16. Ratio of execution times between MATLAB's and CUDA SDK matrix multiplication methods.

This speed up comes with a great accuracy too, being the absolute mean difference between both methods 5·10^-5.

5.3. Gauss-Seidel overview

The Gauss-Seidel method is an iterative method used to solve a linear system of equations. It can be applied to any matrix with non-zero elements on the diagonals, but its convergence is only guaranteed if the matrix is symmetric and positive definite. Since this algorithm can solve a linear system of equations, if the result is the identity it can find the system matrix inverse.

This is not a direct method like Gaussian Elimination, but iterative: it finds the solution by computing an initial solution value and refining iteration after iteration. It has two advantages:

- It may be computationally more efficient than Gaussian elimination.
- The round-off error can be controlled.

The execution times of this algorithm are the following:

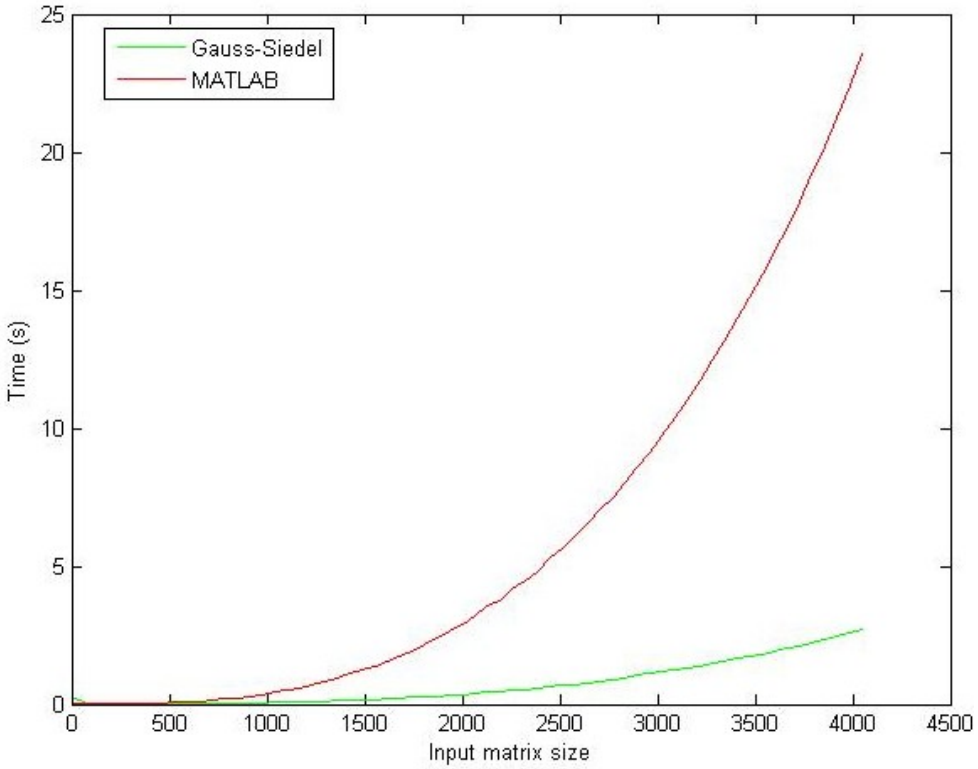

Figure 5.17. Comparison between Gauss-Seidel method and MATLAB's inversion execution times.

This algorithm has been found in NVIDIA forums [6]. It hasn't been developed during the elaboration of the project and it's only featured for academic purposes.

5.4 Summarization and comparison

This point is a summarization of the results obtained along this chapter. Four methods to invert a matrix are compared: MATLAB's built-in function; Gauss-Jordan elimination; the Cholesky, GETS and matrix multiplication combination; and Gauss-Seidel.

Here is a graph with the execution times of the four methods mentioned:

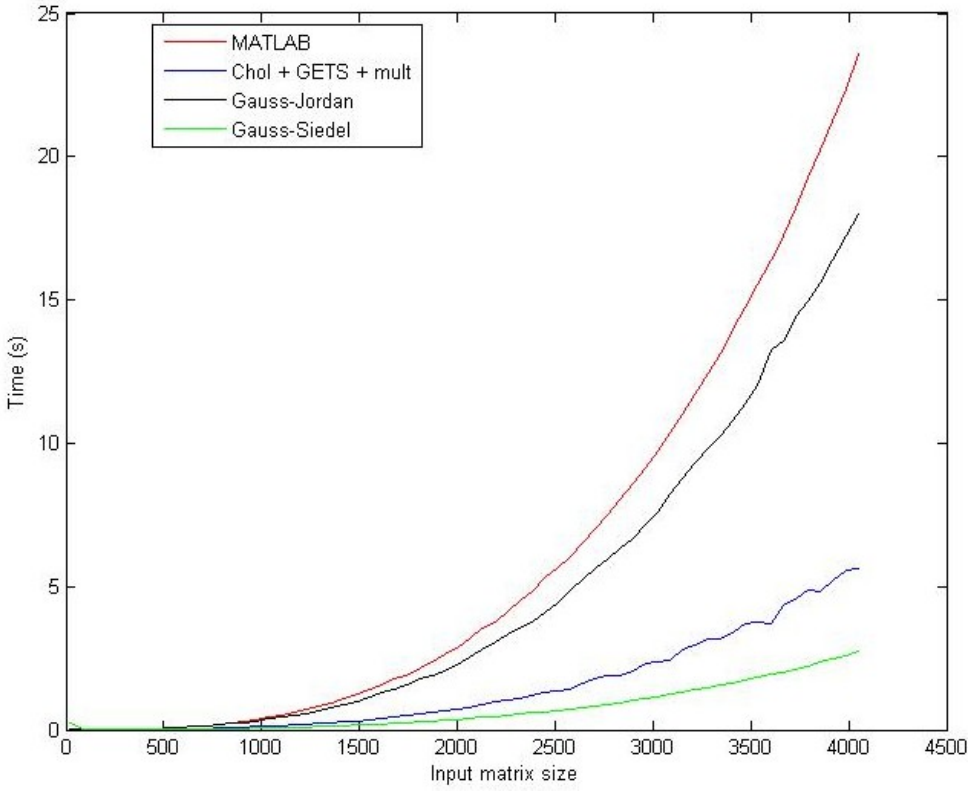

Figure 5.18. Matrix inversion execution times comparison between the two methods developed, Gauss-Seidel and MATLAB's built-in function

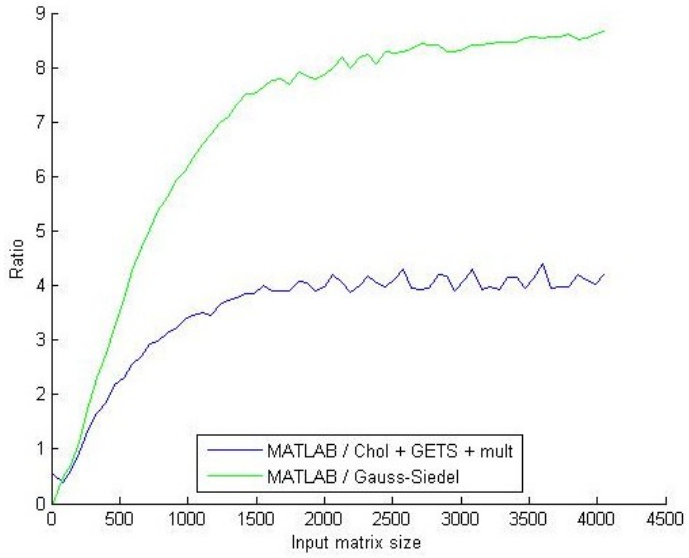

Figure 5.19. Ratio of matrix inversion execution times comparison between the best of the two methods developed, Gauss-Seidel and MATLAB's built-in function

The Gauss-Seidel method is the fastest algorithm of the four, improving MATLAB's speed up to 8 times. The Colesky, GETS and matrix multiplication combination speed it up 4 times, which is not a bad result, but it could be better. The Gauss-Jordan, however, spends too much time, and it's not a worthy choice.

The Cholesky, GETS and matrix multiplication combination code, even with non-optimal results, can be adapted for other purposes:

- The matrix multiplication code can be used in many applications that require large-sized matrix multiplications, with a great performance improvement.
- With slight modifications the GETS algorithm can be used to solve triangular systems instead of back-substitution.

The downfall of this way of calculating a matrix inverse is the Cholesky decomposition. As said in its analysis, its bottleneck comes from the sequential update

instead of a simultaneous update of the matrix. Maybe a reconsideration of that kernel could speed up the whole operation.

CHAPTER 6

PRACTICAL IMPLEMENTATION

In this chapter of the project we will apply the programs developed to attack an iterative image reconstruction problem from a different approach (channel matrix inversion) and achieve better execution time performance with the results of comparable quality.

6.1 System matrix creation

Being g[x,y] the image obtained by any measure system (see Chapter 5):

$$\boldsymbol{g}[\mathbf{x},\mathbf{y}] = \boldsymbol{h}[\mathbf{x},\mathbf{y}] * \boldsymbol{f}[\mathbf{x},\mathbf{y}] + \boldsymbol{n}[\mathbf{x},\mathbf{y}]$$

it is possible to express this same model in a matrix notation such as:

$$\boldsymbol{g} = \boldsymbol{H} \cdot \boldsymbol{f} + \boldsymbol{n}$$

where capital letters express matrix and minus letters express vertical vectors. The ***g***, ***f***, and ***n*** vectors are done just by putting all the pixels of the images consecutively in a vector, from left to right and from top to bottom.

The system matrix ***H*** is created by having in the i-th row the contributions of all the ***f*** vector pixels to the i-th pixel of the g vector.

To create the matrix automatically in MATLAB the following code has been developed:

Table 6.1. Code to create the *H* system matrix.

```
function Hout = filter2matrix( h, rowsf, colsf, rowsg, colsg)
%FILTER2MATRIX Converts a filter f to a system mattrix H
```

```
%   H = filter2matrix( h, Mf, Nf, Mg, Ng)
%       h: the initial filter
%       rowsi: the number of rows of the image (i = f, g )
%       colsi: the number of columns of the image (i = f, g )
%       Reminder:      g = H*f + n

% Find the central element of the h mattrix
    [rowsh, colsh] = size( h );
    rowc = ceil(rowsh/2);        %central row
    colc = ceil(colsh/2);     %central column

%Creation of the needed blocks
    a(1:colsg,1:colsf,1:rowsh) = 0; % 3D, cada i indica un cuadrado
diferente

% Give the blocks values
    for i=1:rowsh               % i controls h rowsa
        for j=1:colsh           % j controls h columns
            for k=1:colsg       % k controls the current block's row
                pos = j-colc+k;
                if ((pos>0)&&(pos<(colsf+1)))   % If we are inside each
"block"
                    a(k,pos,i) = h(i,j);
                end
            end
        end
    end

% Final matrix mount
    Hout(1:(rowsg*colsg),1:(rowsf*colsg)) = 0;  % Inicialization
outside of the loop

    for j=1:rowsh           % j controls a's index
        for k=1:rowsg       % k controls H block rows
            pos = j-rowc+k;
            if ((pos>0)&&(pos<(rowsf+1)))   % if we are inside H
                Hout((1 + (k-1)*colsg):(k*colsg), (1 + (colsf*(pos-
1))):(colsf*pos)) = a(:,:,j);
            end

        end
    end

end
```

6.2 System matrix inversion method

As said previously in Chapter 5 of this report, the recovered image f can be computed as:

$$f_{recovered} = (H \cdot H^T)^{-1} \cdot H^T \cdot g$$

$$f_{recovered} = (H \cdot H^T + \Delta \cdot Id)^{-1} \cdot H^T \cdot g$$

The process of recovering f is the following:

1. Degraded image *g* and filter *h* conversion to their matrix form.
2. Multiply H by its transposed.
3. Add lambda times the identity matrix if needed.
4. Invert the resulting matrix of points 2 and 3.
5. Multiply the result from 4 by H transposed.
6. Multiply the result by g.

The following codes are MATLAB's .m files to do the reconstruction in only one call.

Table 6.2. Implementation in a .m file of the system matrix inversion method.

```
function vector = im2vector( image )
%IM2VECTOR Just a smoke screen for reshape
    [M, N] = size( image );
    vector = reshape( image', M*N, 1 );
end

function image = vector2im( vector )
%VECTOR2IM Just a smoke screen for reshape
    M = sqrt( length( vector ));
    image = reshape( vector, M, M )';
end
```

```
function f = reconstruction( g, h, lambda )
   %ITERATIVERECONSTRUCTION recovers a blurred and noisy image
   % System: G = H·F + N
   %       G, F y N vectors, H system matrix

       G = single( im2vector( g ) );
       [M, N] = size( g );

   // Function that creates the system matrix H.
   // Explained in the previous section
       H = single( filter2matrix( h, M, N, M, N ) );

   % Ways to isolate the reconstructed image f*:
   %   1. f* = (H·H')^(-1)· H' · g
   %   2. f* = (H·H' + lambda·I)^(-1) · H' · g

       HHT = matrixMul( H, H' );

   % Method 2.  f* = (H·H' + lambda·I)^(-1) · H'· g
       if ( lambda ~= 0 )
           HHT = HHT + lambda * eye( length(H));
       end

   % Identical for both methods
       HHT1 = gpu_inversion( HHT );  // developed CUDA matrix inversion
       f = matrixMul( HHT1, H' );    // developed matrix multiplication
       f = f * G;

       f = vector2im( f );    // Reshape the image

end
```

6.3 Results

In this section the results obtained with the matrix inversion method. The original image used for this test is the following:

Figure 6.1. Original image to be reconstructed by using the inverse matrix reconstruction method

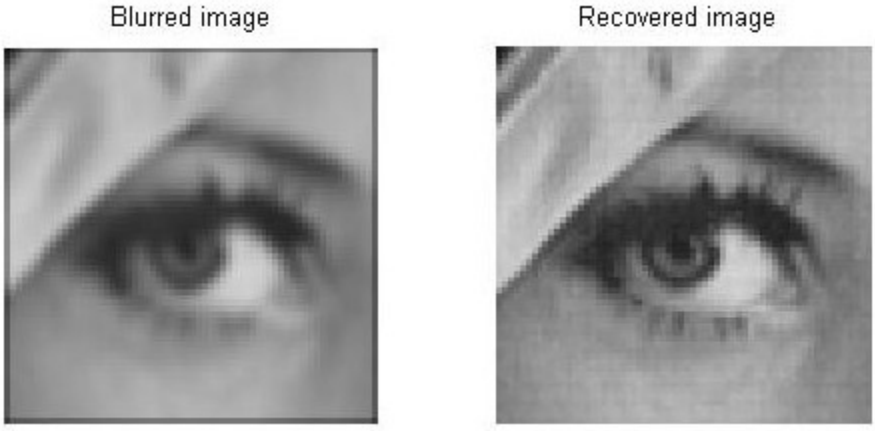

Figure 6.2. Blurred image without noise and its recovered image using the inverse matrix reconstruction method.

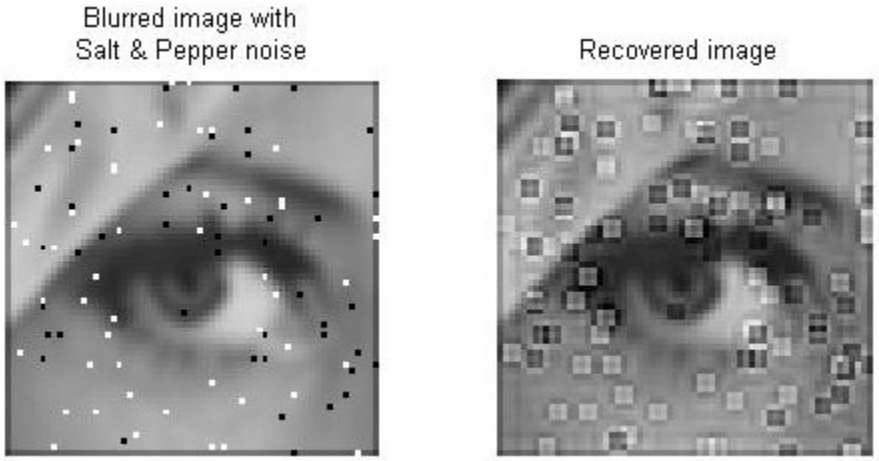

Figure 6.3. Blurred image with salt and pepper noise and its recovered image using the inverse matrix reconstruction method.

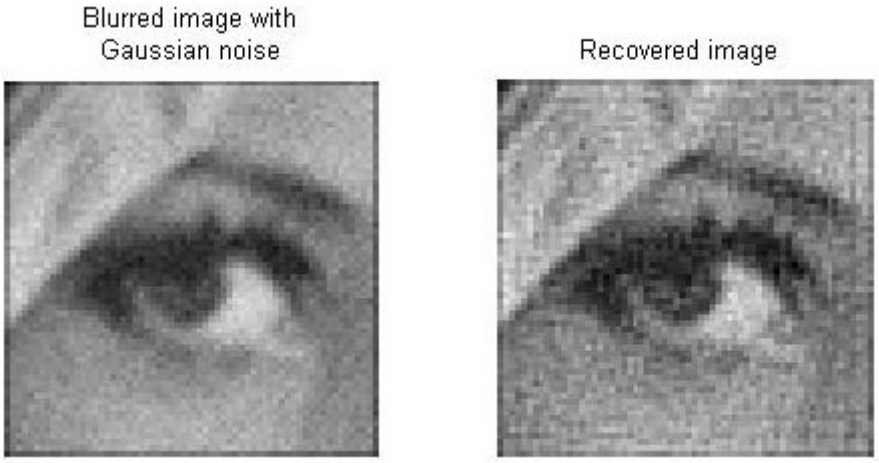

Figure 6.4. Blurred image with Gaussian noise and its recovered image using the inverse matrix reconstruction method.

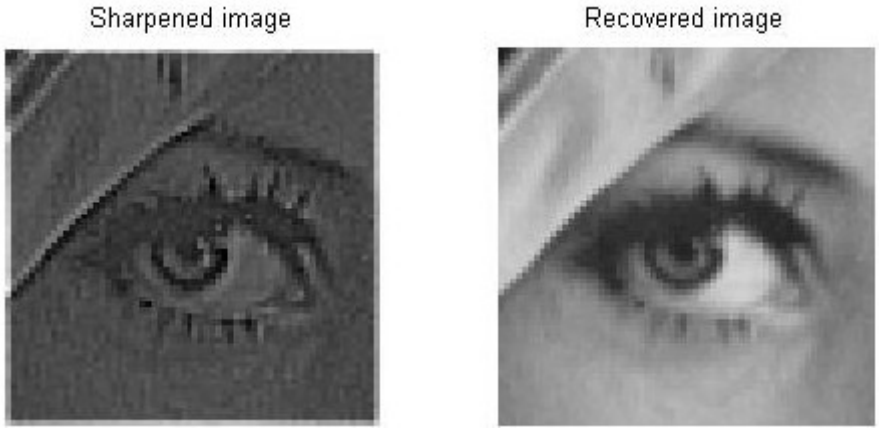

Figure 6.5. Sharpened image without noise and its recovered image using the inverse matrix reconstruction method.

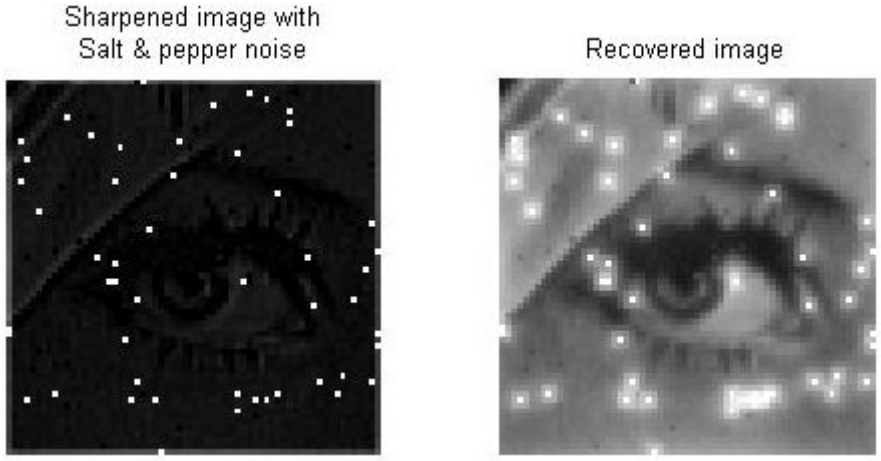

Figure 6.6. Sharpened image with salt and pepper noise and its recovered image using the inverse matrix reconstruction method.

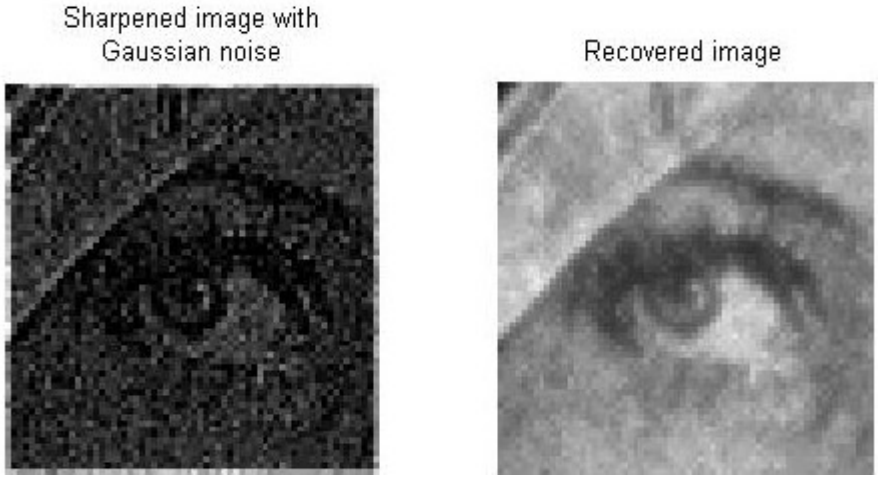

Figure 6.7. Sharpened image with Gaussian noise and its recovered image using the inverse matrix reconstruction method.

The mean square error (mse) between the previous images and the original (the one that in a real case is unknown) are the following:

Table 6.3. Mean square error between degraded images and recovered images with the matrix inversion reconstruction method.

	MSE degraded image	MSE recovered image
Blurred without noise	0.0032	6.6104e-005
Blurred with Salt & Pepper noise	0.0087	0.0112
Blurred with Gaussian noise	0.0047	0.0044
Sharpened without noise	0.2469	1.0408e-011
Sharpened with Salt & Pepper noise	0.2469	0.03
Sharpened with Gaussian noise	0.2459	0.0034

BIBLIOGRAPHY

[1] CUDA http://www.nvidia.com/object/cuda_home_new.html

[2] MATLAB MEX Guide. http://www.mathworks.com/support/tech-notes/1600/1605.html#ingredients

[3] Iterative Image Reconstruction. David S. Lalush and Miles N. Wernick.

[4] CS738 Project : Cholesky Decomposition
http://www.cse.iitk.ac.in/users/arajput/files/Projects/cs738_report.pdf

[5] NUMERICAL RECIPES, Third Edition (2007), 1256 pp, Cambridge University Press

[6] Gauss-Seidel algorithm source (from NVIDIA Forums).
http://forums.nvidia.com/index.php?showtopic=80108