TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

Implementation of LTE Mini receiver on GPUs

Supervisors:

Dipl.-Ing Michal Šimko

Dipl.-Ing. Mag. Dr.techn. Sebastian Caban

Univ.Prof. Dipl.-Ing. Dr.techn. Markus Rupp

Institute of Telecommunications

Vienna University of Technology

Faculty of Electrical Engineering and Information Technology

Of

José Vieira

Number: 1028204

Portugal, Guimarães

Vienna, March 2011

Abstract

Long Term Evolution (LTE) is the latest standard for cellular mobile communication. To fully exploit the available spectrum, LTE utilizes feedback. Since the radio channel is varying in time, the feedback calculation is latency sensitive. In our upcoming LTE measurement with the Vienna Multiple Input Multiple Output (MIMO) Testbed, a low latency feedback calculation is necessary. The presented work consists of exploring the computational potential of Graphics processing unit (GPUs) NVIDIA Technology and Compute Unified Device Architecture (CUDA) programming model, with aim to optimize the execution time of a LTE Mini Receiver. Two implementations are presented, first is sequential and second is parallel using CUDA. Thus, I will present a brief study of some techniques for parallel programming and CUDA architecture. In the end, I will present and compare all obtained results between MATLAB, CUDA and C. The comparison is made through the processing time of these three implementations.

Sumário

Long Term Evolution é o mais novo padrão para a comunicação móvel celular. Para explorar plenamente o espectro disponível, a LTE utiliza *feedback*. Uma vez que o canal de rádio é variável no tempo, o cálculo do *Feedback* é sensível à latência. Nos LTE *measurements* do grupo MIMO *Testbed* de Viena, o cálculo do retorno de baixa latência é necessário. O trabalho apresentado consiste em explorar o potencial computacional da tecnologia dos GPU *Nvidia* e modelo de programação CUDA, com o objectivo de optimizar o tempo de execução de um Mini Receptor LTE. Duas implementações são apresentadas, uma sequencial usando C e outra em paralelo usando CUDA. Apresentarei ainda um breve estudo de algumas técnicas de programação paralela e arquitectura CUDA. No final, serão mostrados e comparados os resultados obtidos entre CUDA, MATLAB e C.A comparação é feita através dos tempos de processamento de cada um destes processos.

Acknowledgments

To my parents, values, love and all the support they always gave me. Besides, they always have provided me with great sacrifice, what is required for my academic success depended solely on me instilling in me a great sense of responsibility.

To my girlfriend Alexandra, for the support, trust and understanding, which in the period when I was away, always had a word of support in those bad days.

To my sister for the help and advice that she always gave me and example that she always represented to me. To my godson, thank you.

To Michal Šimko my advisor, by the competence with which guided this thesis and the time that he spent generously, sending me the best and most useful teachings, patience and clarity. By the access to a broader research and enriching and its always so timely criticism as constructive, I am very grateful.

To Professor Markus Rupp for allowing this work to accept me, even being a foreign student. I would also like to thank all the colleagues of the Institute of Telecommunications, in particular attention to Martin Mayer, for the support and availability provided.

Although all my friends for friendship and companionship. In particular the assistance and friendship provided by Miguel Sousa these past months.

Contents

List of Figures

List of Tables

1 Introduction

Long Term Evolution (LTE) is the latest standard for cellular mobile communication. To fully exploit the available spectrum, LTE utilizes feedback. Since the radio channel is varying in time, the feedback calculation is latency sensitive. In our upcoming LTE measurement with the Vienna Multiple Input Multiple Output (MIMO) Testbed, a low latency feedback calculation is necessary.

The graphics processor can be used to efficiently execute data parallel computations, and using Compute Unified Device Architecture (CUDA), it can be programmed using a C-like language. The implementation of this new method is a promising way to reduce the computation time.

MATLAB is a powerful tool in which, through MEX Files, it is possible to exploit the computational power offered by the NVIDIA Graphics processing unit (GPU) architecture. Through MEX Files it is possible to run CUDA functions directly in MATLAB, much like built-in functions.

The purpose of this master thesis is to develop, implement and test a low-latency mini receiver for LTE. In the end, the various implementations will be tested and compared in order to test which is the best method.

The structure of this thesis is as follows. On the first section, the LTE physical layer will be discussed in order to explain the theoretical aspects. On the second section, we present the architecture of NVIDIA GPUs and the CUDA programming model. The different implementations will be presented at third section. The final section presents the discussion of results and conclusions.

2 Long Term Evolution: Physical Layer

The Long Term Evolution is the latest standard in the mobile network technology, LTE is an evolution of the Global System for Mobile Communications/ Universal Mobile Telecommunications System (GSM/UMTS) systems family, and specifies the next generation mobile broadband access system [1].

2.1 Overview

LTE supports and takes advantage of a new modulation technology based on Orthogonal Frequency Division Multiplexing (OFDM) and MIMO data transmitting,. The benefits of LTE comes from increased data rates, improved spectrum efficiency, improved coverage, and reduced latency. For operators, a flexible spectrum usage is included [2].

LTE provides an uplink speed of up to 50 megabits per second (Mbit/s) in Quadrature Phase Shift Keying, 57Mbps in 16 Quadrature Amplitude Modulation (QAM) and 86Mbps in 64QAM. The provided downlink speed is up to 100 Mbps in SISO, 172 Mbps in 2x2 MIMO and 326 Mbps in 4x4 MIMO [26]. It can bring a lot of technical benefits to cellular networks. The amount of flexibility in LTE, that allows operators to determine the spectrum that will be deployed. It also features scalable bandwidth, the amount of bandwidth in an LTE system can be scaled from 1.4 MHz to 20 MHz [27].

This will suit the needs of different network operators that have different bandwidth allocations. Networks can be launched with a small amount of spectrum, alongside existing services and adding more spectrum as transit users and also allow operators to provide different services based on spectrum, adjusting their network deployment strategies used. The spectral efficiency in 3G networks will be also improved [3].

The LTE transmission is divided into the following structure. Layers of 10 ms that are divided into 10 sub-layers of 1 ms and each of these layers is divided into two slots of 0.5 ms each then these slots are divided into seven OFDM symbols in the case of normal cyclic prefix, or 6 if the cyclic prefix is configured in an extended cell and it is represented in Figure 1.

In the frequency domain features are grouped in units of 12 subcarriers (occupying a total bandwidth of 180 kHz).This unit of 12 subcarriers is called a Resource Block (RB) and has 84 Resource Element (RE) in the normal case or 72 RE in the case of extended cyclic prefix. The small units are called RE [4].

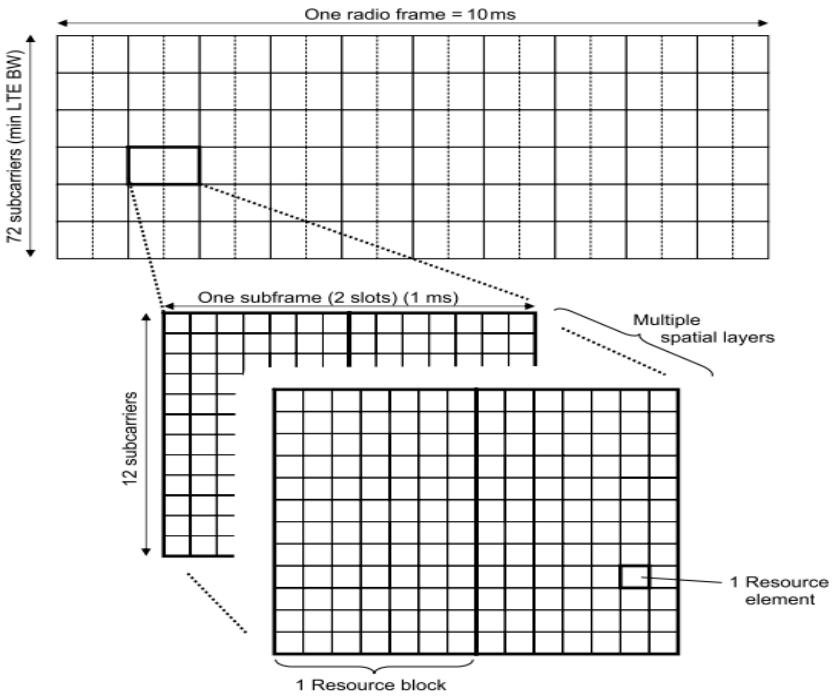

Figure 1 - Basic time-frequency resource structure of LTE [4]

The primary goal of LTE are to minimize the system and User Equipment (UE) complexities, permitting flexible spectrum deployment in existing or new frequency spectrum and to enable co-existence with other 3GPP Radio Access Technologies.

2.2 Reference Symbols

Pilot symbols are used in LTE to facilitate Channel Estimation, to monitor the temporal variation and frequency selectivity of the channel, known pilot symbols are inserted periodically both in the time and frequency dimension [6]. But there is a problem, because these pilot symbols consumes also bandwidth and reduces too the transmitted energy for data symbols [28]. Through these pilot symbols is possible to estimate the channel at the given locations within a resource grid. The channel response across an entire resource grid is determined by interpolation between these estimates [6].

The Figure 2 shows the principle of the downlink reference signal structure for 4 antennas transmission. The colored squares correspond to pilot symbols in a particular antenna port and crosses correspond to the positions in time-frequency grid, which are 0. Within each resource block in the first and second transmission antenna port, there are four pilot symbols, and at 3 and 4 transmit antenna port, just 2. If the number of antennas increases, the number of pilot's symbols and symbols which are zero will increase too, in consequence the spectral efficiency will decrease [7].

For the 3 and 4 antenna will be fewer pilots. Therefore, the quality of the estimates of channel will be worse than the quality of the estimated channel of the first and second transmission antenna. Accordingly, the use of four ports of the transmitting antenna should be restricted to situations where the channel is not changing rapidly.

Thus, the reference signal can be seen as two dimensional cell identifier sequences. Each reference signal sequence is generated as a symbol-by-symbol product of an orthogonal sequence (3 of them existing) and a pseudo-random sequence (170 of them existing) [31].

Each cell identity corresponds to a unique combination of one orthogonal sequence and one pseudorandom sequence, allowing 510 different cell identities [31].

Thus, the complex value of pilot symbols is cell dependent. The frequency domain position of the pilot symbols may vary between consecutive sub frames. In Figure 2 we can see the relative position of the pilot symbols is always the same. The frequency hopping can be described as adding frequency offset to the basis pilot symbols position structure [7].

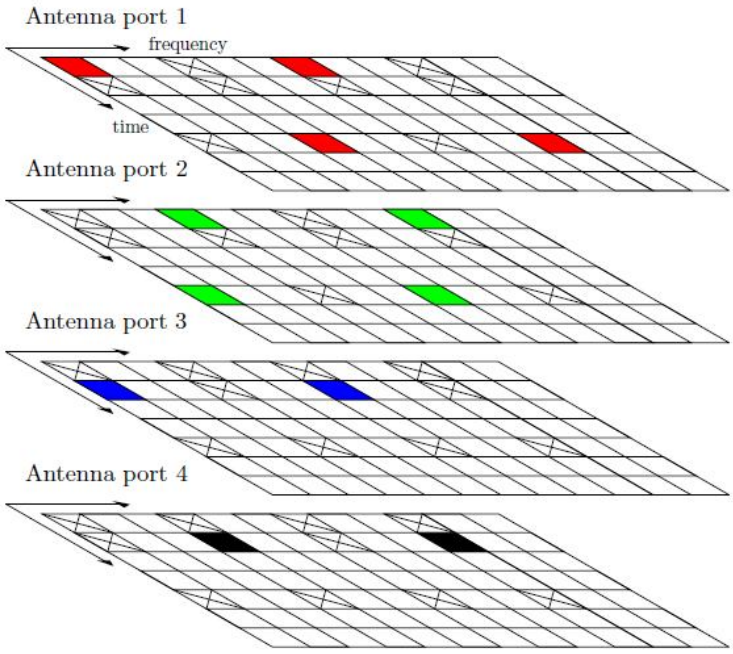

Figure 2 - **Pilot symbols structure [7]**

2.3 MIMO

MIMO is a smart antenna technique based on the use of multiple antennas at both the transmitter and receiver to improve radio link communication performance [30]. The MIMO technology takes advantage of a radio-wave phenomenon called multipath signal reflections to actually improve radio transmission performance where transmitted information gets reflected from windows, walls, and other objects, reaching the receiving antenna multiple times via different angles and at slightly different times [29].

The MIMO technique is gradually becoming more and more used in wireless data transmission, and essentially uses multiple antennas at the receiver and transmitter. The schemes employed in MIMO are a little different in the uplink and downlink. In case the user terminal, there is a strong restriction of the product cost factor which ultimately limits the design possibilities.

For the LTE downlink, accordingly schemes using 2 x 2, 4 x 2 and 4 x 4 antenna matrices can be used. A 2x2 scheme has two transmit antennas at base station and two receive antennas in the mobile terminal, while a 4 x 4 scheme has four antennas at transmitter and two or four at receiver, respectively [30].

Single user MIMO (SU-MIMO) is the most common form of MIMO, the main objective is to increase the data rate for an user. Single user MIMO communications systems exploit multiple transmit and receive antennas to improve capacity, reliability, and resistance to interference, primarily through the use of space-time combined with transmission of multiplexed stream. In such a single user view of the traditional MIMO systems, the extra spatial degrees of freedom brought by the use of multiple antennas are exploited to enlarge the dimensions available for signal processing and sensing, acting mainly as a booster of the physical (PHY) layer .In this approach, the link layer protocols for multiple accesses (uplink and downlink) indirectly reap the performance benefits of MIMO antennas in the form of higher rates per user, or more reliable channel quality, while not requiring the full awareness MIMO capacity [5].

The Figure 3 shows the MIMO scheme SU-MIMO in LTE.

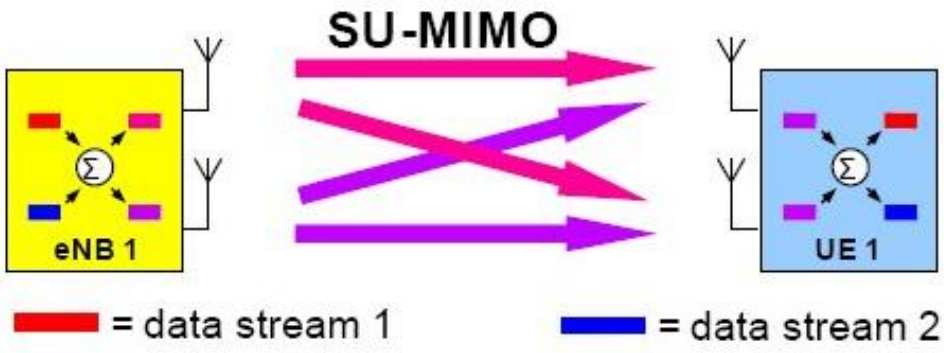

Figure 3 - MIMO schemes in LTE [11]

2.4 Channel Estimator

A radio channel bandwidth is typically frequency selective and time variant. For an OFDM mobile communication system, the transfer function of the channel at different subcarriers appears unequal in both frequency and time domain. Therefore, channel estimation is required in a coherent OFDM system which provides a spectrally efficient modulation in comparison with the non-coherent method [19].

The estimation can be based in few different methods, the Least squares (LS) will be discussed in this sub-chapter.

2.4.1 Least Squares Channel Estimation

When a pilot is present on the k-th subcarrier within the ns-th OFDM symbol on the nt-th transmit antenna port, the symbols of the remaining transmit antenna ports, on the same subcarrier and OFDM symbol, are 0 [7].

The spectral efficiency is reduced due to this scheme, however, preserves the orthogonality between the pilot symbols at different transmit antenna ports, and this makes it possible to estimate a MIMO channel as Nt Nr independent Single Input Single Output (SISO) channels [7].

A common method for pilot-based channel estimation, called least squares estimation, is a method that provides a good performance and reasonable complexity, without using any knowledge of the statistics of the channels. This complexity depends on the number of transmit antennas and the number of training symbols per antenna [20][21].

However, this method requires pilot symbols and matrix inversion. At the beginning of each burst of data, is transmitted by each transmit antenna, training sequence known by the receiver [32]. The training sequence consists of pilot symbols, which are orthogonal among different transmit antennas [21].

2.5 Feedback Calculation

In the mobile communication system LTE, at the receiver the calculation of three different values of feedback is required, in order to adapt the channel. So called closed loop spatial multiplexing transmission mode. The three feedback values are the Channel Quality Indicator (CQI), Rank Indicator (RI) and Pre-coding Matrix Indicator (PMI) [22] [25].

In a communication system, Channel Quality Indicators (CQI) are messages that are sent and establish a connection with remote information about the quality of the channel [23]. Through the CQI, one of 15 modulation alphabet and code rate combinations for transmission can be selected by transmitter [22] [25].

The UE's recommendation for the number of layers is called the rank indicator (RI). When is operating in a MIMO mode with spatial multiplexing, RI is reported, when operating a single antenna or diversity TX, is not reported. The RI is always associated

with one or more CQI reports, being considered for the calculation of the CQI reported that particular RI value. The rank is reported less frequently because it is usually slower to vary than the CQI. The frequency selective RI reports are not possible [24].

The PMI provides information and signals the codebook index of the preferred pre-coding matrix. The number of eNodeB antenna ports affects the number of pre-coding matrices in the codebook. Depending upon the feedback CSI, the PMI reports may be wideband, or else selectively in frequency unlike the RI reports [24].

Due to limitation of immediate feedback and hardware limited by the signal processing, finding a solution, ideal for all possible combinations between the three values of feedback, in most cases is not always possible. For this it is necessary to reduce the computational complexity by sacrificing global optimality. This reduction can be achieved through the separation process of global optimization in several steps so as to find the values for three feedback parameters [22] [25].

3 Compute Unified Device Architecture

CUDA is a platform that utilizes Software and Hardware for parallel computation of high performance from general proposes that utilizes the processing power of NVIDIA GPU's.

3.1 Overview

CUDA can be applied in several fields, mathematics, science, biomedicine and engineering. Due to the characteristics of applications in these fields, which are highly parallelizable, CUDA is very useful. This technology, despite some disadvantages over current supercomputers has drawn considerable attention from all over academic community due to its great computational power.

Driven by strong demand and increased market's insatiable real time and high-definition 3D graphics, programmable Graphic Processor Unit has evolved in parallel, multithreaded, manycore processors with tremendous computational horsepower and very high memory bandwidth. It can be illustrated by Figure 4 and Figure 5 [9].

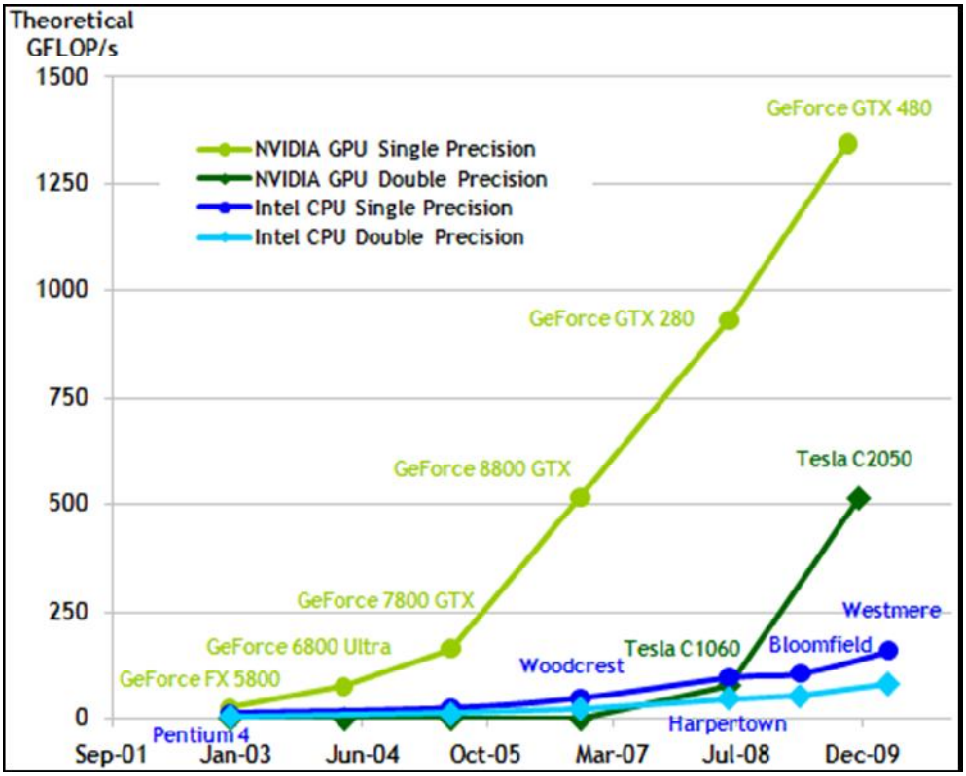

Figure 4 - GFLOPS Evolution of GPU's and CPU [8]

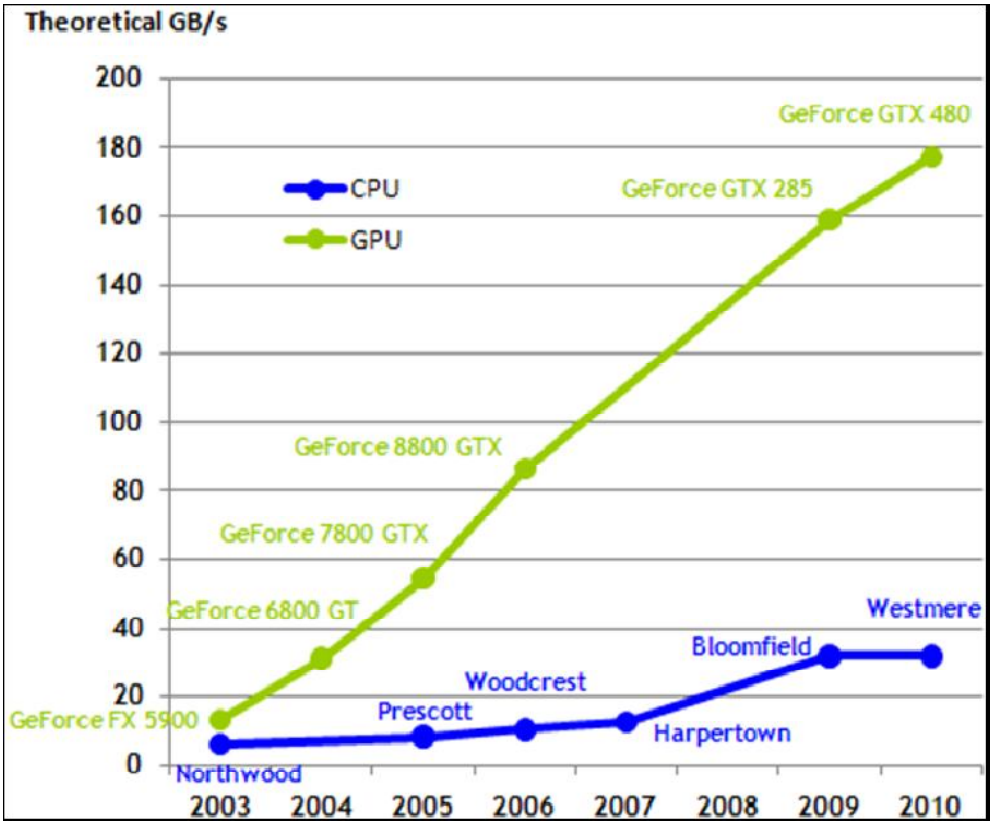

Figure 5 - Bandwidth Evolution of GPU's and CPU [8]

3.2 GPU versus CPU

Comparing GPU with Central processing unit (CPU), CPUs have more complex cores and more cache, better for independent operations, random access memory and operations with little data. On other way the GPU are better for simple tasks in large arrays, GPU is specialized for compute-intensive, highly parallel computation and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control, as schematically illustrated by Figure 6.

Figure 6 - The GPU Devotes More Transistors to Data Processing [8]

More specifically the GPU is suited to solve the problems that can be expressed as data parallel computations with high arithmetic intensity. For the CPU the same program is executed for each element in contrast to the GPU, which is performed for all elements at the same time.
Thus, many applications that process large data sets can use a programming model for data in parallel to speed up calculations, applications, media and image processing, such as post-processing of rendered images, video encoding and decoding, expansion image, stereo vision, and pattern recognition can map image blocks and pixels for segments of parallel processing [8].

3.3 Programming Model

This sub-chapter introduces the main concepts behind the CUDA programming model by outlining how they are exposed in C [8].

The execution of an application using CUDA is performed in both, GPU (device) and CPU (host), only the code that involves the policies introduced special runs on the GPU, the rest is normally performed by the CPU.

3.3.1 Kernel Functions

A kernel function specifies the code to be executed by all threads of a parallel phase. This function, when called, is executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions. A kernel controls the GPU Execution of a set of threads, which are arranged in grids of thread blocks.

To define a kernel we need to use the __ *global*__ declaration specifier and the number of CUDA threads that will be executed, using the following syntax, Kernel<<<*dim_grid*, *dim_block*>>> (…). Each thread that executes the kernel has an unique thread ID that can accessible within the kernel through the built-in *threadIdx* variable [8].

A thread is a flow of independent execution that has a shared memory with the parent process and can be scaled by the operating system. Typically, the use of threads occurs in conjunction with the distribution of data to be processed. Thus, each thread can process a subset of data in parallel [10].

3.3.2 Grids and Blocks

The scheduling of threads in CUDA uses two concepts: block and grid. Through this we organize the distribution and organization of data, also the data distribution to hardware. The computational grid consists of a grid of thread blocks. A Grid is the basic unit where the blocks are distributed, being the complete structure that holds the threads responsible for executing a specific function, and can have one or two dimensions. A block is the basic unit of organization and mapping of threads to the hardware is a set of threads that will run synchronously and will have shared memory between them. Communication between blocks is done through the global memory, which is slower than the shared memory between the threads of a block. Blocks can have one, two, or three dimensions, providing a natural medium for use in computing elements of a vector field, matrix, or, scalar fields one, two or three dimensions [9].

The Figure 7 shows a grid with dimensions 2x3 and size of blocks 3x4.

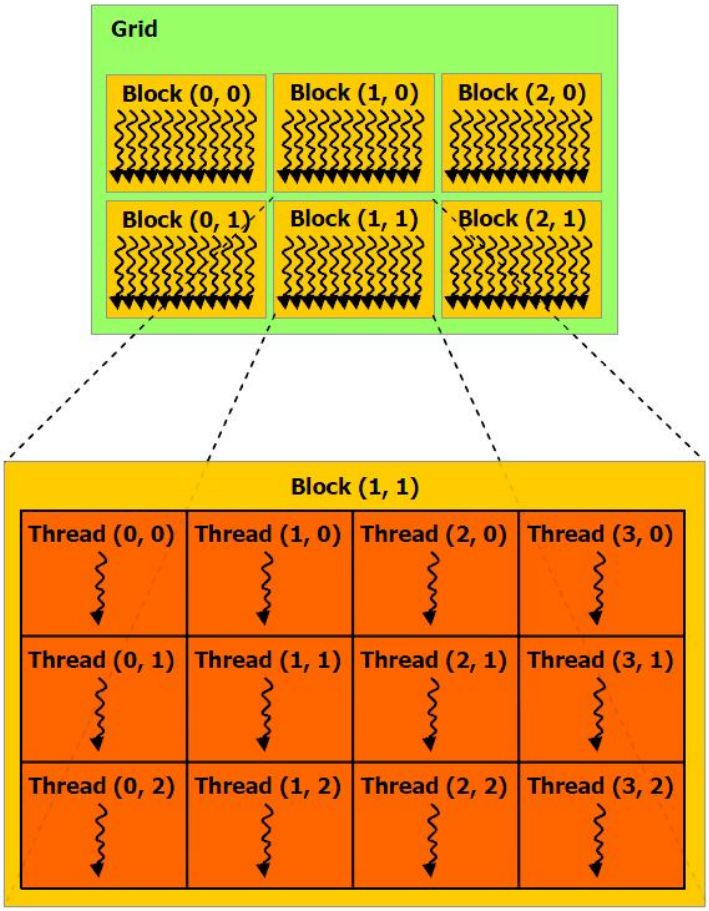

Figure 7 - Grid of Thread Blocks [8]

3.3.3 Execution Flow

Overall, the CUDA program is organized in the main program, consisting of one or more sequential threads running on the CPU and one or more kernels that run on the GPU. The threads of a same block are able to synchronize between them through barriers, gaining access to a very fast communication. The threads of different blocks on the same grid can only be coordinated through global memory operations visible to all threads [9]. The execution begins with the execution of the host (CPU). When a kernel function is called, execution is moved to the device (GPU), where a large number of threads are created to take advantage of the abundant data parallelism. Each kernel is executed one at a time. With this a new kernel is invoked just when all the threads of a current kernel complete their execution [12]. The execution path of a CUDA application can be shown in Figure 8.

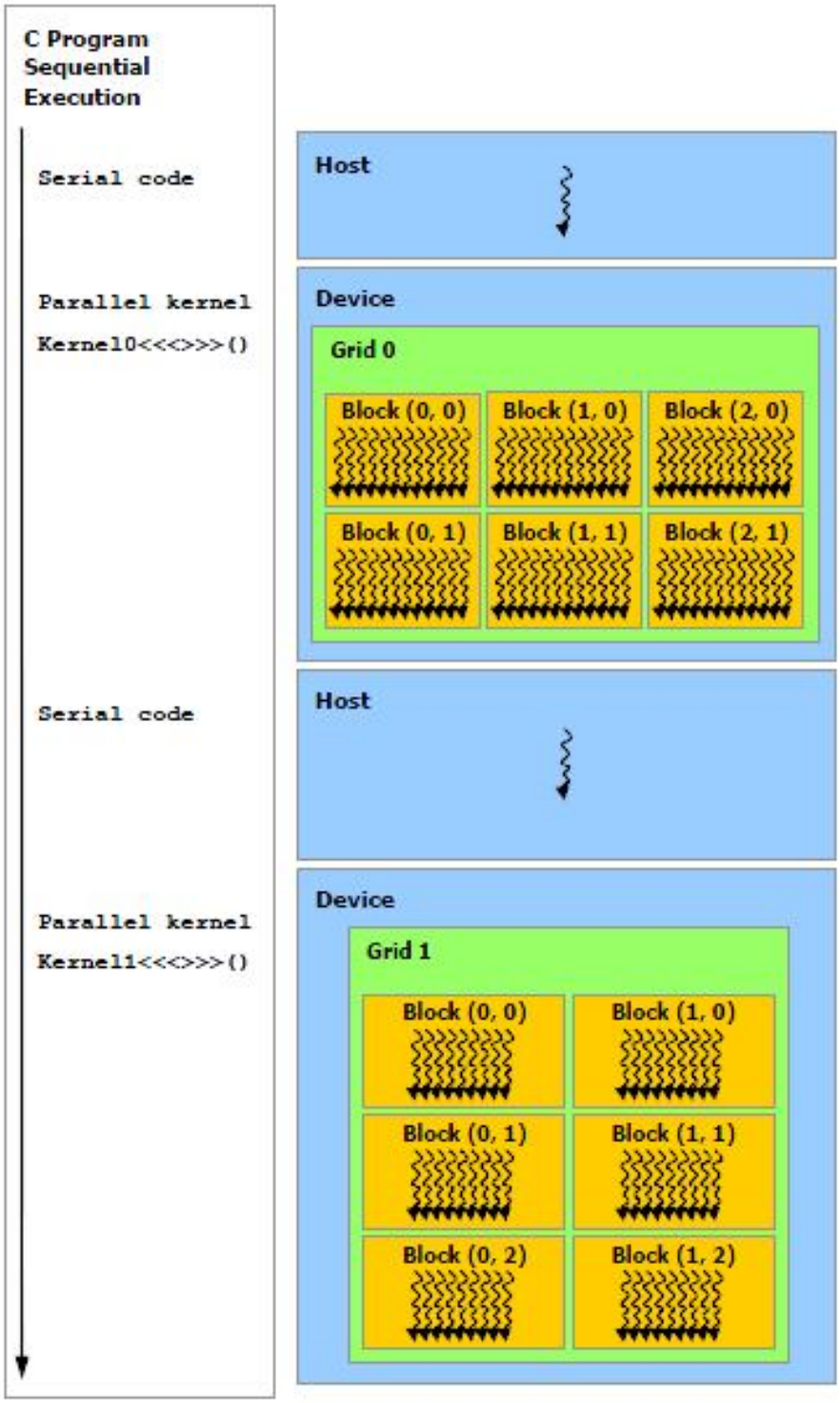

Figure 8 - Execution path of a CUDA application [8]

3.4 CUDA programming

A Basic C CUDA Program should have these containers:

- Selecting the device to be used
- Allocating memory on the host
- Data entry in the host memory
- Allocating memory on the device
- Data transfer from host to device
- Invocation of the kernel
- Data transfer from device to host
- Release memory on the host
- Release memory on your device

In this point, the basic stuff of CUDA programming will be shown.

3.4.1 Device Management

A host can have multiple devices, these devices can be enumerated and one of them can be selected for kernel executions, using *cudaSetDevice()*, it sets device to be used for GPU executions. By default, the device associated to the host thread is implicitly selected as device 0 [8].

```
__host__  int main()
{
CudaSetDevice(1);
…
}
```

3.4.2 GPU Memory Allocation / Release

The *cudaMalloc()* function can be called from the host code to allocate Global Memory for an object, this is used to allocate linear memory. This function has two parameters, where the first is the address of a pointer that should point to the object allocated after a piece of Global Memory to be assigned to it. The second parameter of *cudaMalloc()* gives the size of the object being allocated.

For example, if a programmer wished to perform a 14*512 matrix and have a pointer variable Input that can point to the first element of a single precision array, the programmer just need to do like the following code. After the computation, just need to use *cudaFree()* for deallocate device Global Memory [12].

```
float *d_Input;

int size = 14 * 512 * sizeof(float);

cudaMalloc((void**)&d_Input, size);

…

CudaFree(Md);
```

3.4.3 Memory Management

After allocating memory on the device, can be requested data transfers between memories, from Host to Device or vice versa. For this is called a function *cudaMemcpy()*, this function has four parameters.

Lets analyze the header of the function, *cudaMemcpy*(void **dst*, void **src*, size_t *nbytes*, *cudaMemcpyKind direction*). As we can see the first parameter is a pointer to the source data object to be copied, the second parameter points to the destination location for the copy operation. The third parameter specifies the number of bytes to be copied. The fourth and last parameter indicates the types of memory used in the operation. It can be from host memory to host memory, from host memory to device memory, from device memory to host memory, and from device memory to device memory [12]. For example, to transfer the data variable *h_Input*, from the host to device, we use the following code.

```
cudaMemcpy(d_Input, h_Input, size, cudaMemcpyHostToDevice);
```

3.4.4 Function Qualifiers

There are three qualifiers to function type: *__device__*, *__global__* and *__host__*. The qualifier *__device__* defines a function that will run on the GPU and can only be initiated from the GPU. The qualifier *__global__* defines what is called CUDA kernel, or in other words, a function that runs on the GPU and is initiated from the CPU. Finally, the qualifier *__host__* defines a function that will run on the CPU and that can only be initiated from the CPU [8].

```
__device__          __globa__          __host__.
```

3.4.5 CUDA Built-in Device Variables

CUDA has some Built-in Device Variables, this variables are: *gridDim*, *blockDim*, *blockIdx*, *threadIdx* and *warpSize*.

The *gridDim* and *blockDim* give the dimension of a grid and of a block respectively. The *blockIdx* give the index of a block inside a Grid and *threadIdx* give the index of a thread inside a block. Finally the *warpSize* has the warp length of the threads.

Trough the Figure 9 is possible to see, the Grid Dimension (gridDim) is 3 because the grid has 3 blocks, each block has 5 threads, so the Block dimension (*blockDim*) is 5. The Block Index (*blockIdx*) can take values between 0 and 2, and the Thread Index (*threadIdx*) can take values between 0 and 4.

If we want to know the Global Index of a thread, just need to apply the formula presented in Figure 9. For example, if we want to know the index of the last thread of the last block, index=2*5+4, the index will be 14.

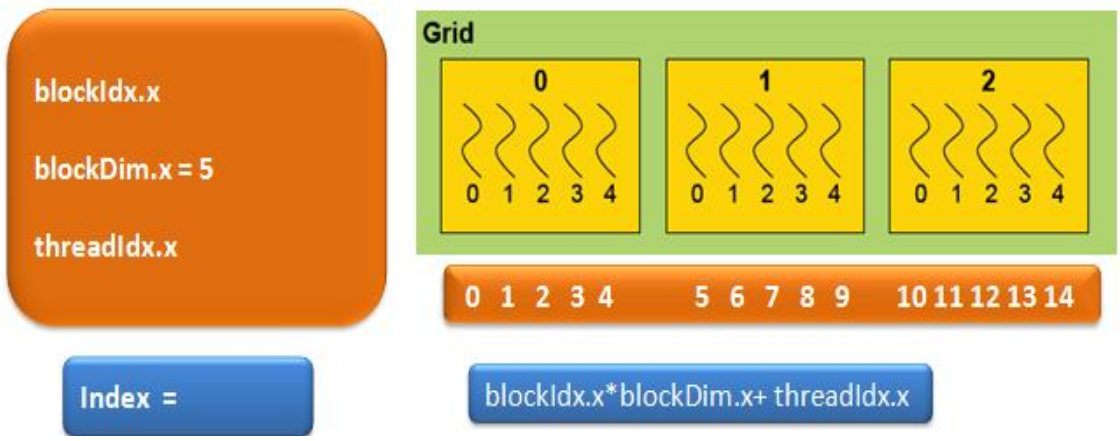

Figure 9 - Built-in Device Variables Example

3.5 MEX Files

MATLAB is a potent instrument for prototyping and analysis and could be easily extended via MEX files to take advantage of the computational power offered by the latest NVIDIA graphics processor unit (GPU). MEX files are a way to call custom C code or CUDA code directly from MATLAB as if they were MATLAB built-in functions [13].

Even though MATLAB is calling optimized libraries internally, there is still space for more optimization. MEX files have been used in the past as a way to invoke multithreaded libraries, now is possible to use this MEX Files to call our own CUDA function. All MEX Files should include the following 4 items [14]:

1. The first item to include in a MWX file is the library *#include "mex.h"*.

2. The gateway routine to every MEX-file is called *mexFunction*, this *mexFunction* is has 4 parameters, and it's always represented in this way:

*mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const mxArray *prhs[ ]) { . }*

The first parameter is *nlhs* that represent the number of expected *mxArrays*, the second parameter is the *plhs*, this parameter represent an array of pointers to expected outputs. The third *nrhs*, is the number of inputs and the last one is *prhs* that represent an array of pointers to input data. The input data is read-only.

3. Then we need to include the *mxArray*. The *mxArray* is a particular structure that contains MATLAB data. It is the C representation of a MATLAB array. All types of MATLAB arrays (scalars, vectors, matrices, strings, cell arrays, etc.) are *mxArray*s.

4. The last item to include is the Application Programming Interface (API) functions that can be used to allocate and free memory.

4 Implementation

In this work, we started from a Mini Receiver implemented in MATLAB, and the main idea was to accelerate this Mini Receiver with NVIDIA technology C CUDA. The first stage of the work is to remove the cyclic prefix. Afterwards, we use the Fast Fourier Transform to convert the signal into the frequency domain. Finally, using the channel estimation and equalization, the data estimates are obtained. To help with this implementation, several steps have been taken.

The first step was to understand what was being done in every part of the simulator. After that, it was necessary to translate MATLAB code to C code (both serial implementations), being the later similar to C-CUDA. Finally, after we obtain the simulator in C Language, it was translated to C CUDA. For this task, it was key to drift from the conventional approach when it comes to programming and start thinking in parallel, because NVIDIA GPUs take advantage of implementations in parallel in order to increase performance.

By itself, this Mini Receiver has several stages, as we can see in Figure 10, the ones implemented were:

- Cyclic Prefix Removal
- FFT
- Circ shift
- Remove zero DC Carrier
- Disassemble reference symbols

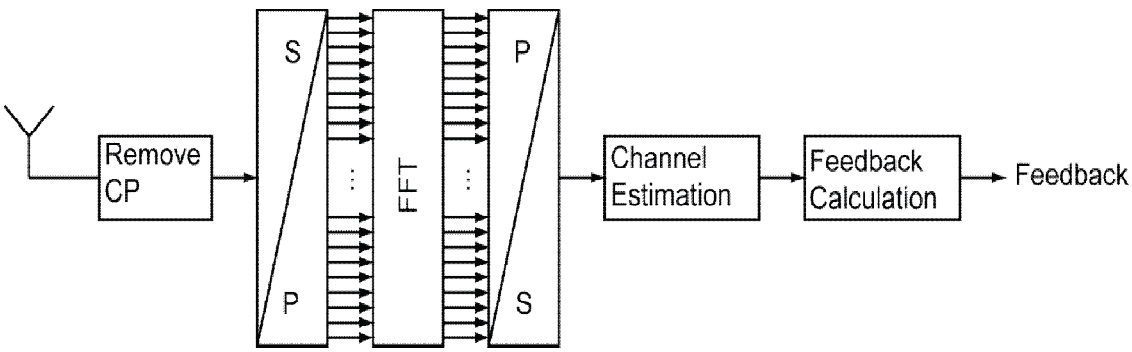

Figure 10 – Simple schematic of LTE Mini Receiver

4.1 Cyclic Prefix Removal

Cyclic prefix is a guard interval, and it refers to the prefixing of a symbol with a repetition of the end. It consists of transmitting the end/final part of the symbol (data to be sent) OFDM previous to the current one. This is done because, at the time that the symbol sent reaches the receiver, there's still the end of the previous one, due to both the numerous reflections that occur and the propagation delay. If the prefix didn't exist, the symbol data could be lost, due to interference from the ISI (Inter Symbol Interference) with the end part of the previous one. Thus, the duration of the prefix must be chosen as the greatest delay in the channel. On the following example, one can see the transmission of two such symbols [15].

On the flowing Figure 11 we have an example of a transmission of two consecutive symbols [15].

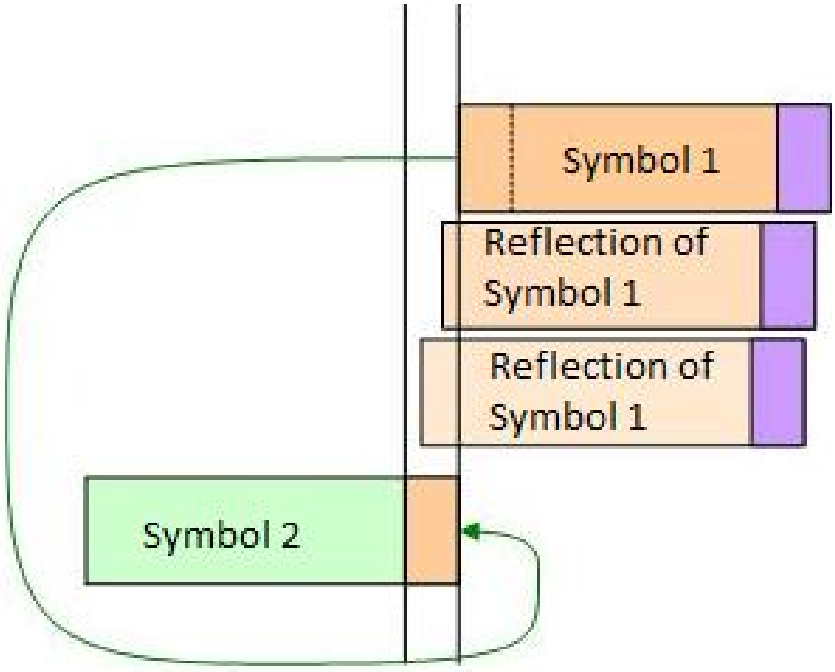

Figure 11 - Demonstration of the principle of cyclic prefix [15]

For better understanding of this process, we will assume that the Data contained in the Mini receiver is the values shown in Figure 12. Here, the process is shown for only one antenna, but it remains the same if more are used.

Looking at Figure 12, for one antenna, there is an array with length 7680. This array contains 7168 symbols and 512 Cyclic Prefix. The first 40 indexes of this array are cyclic prefix, between the indexes 41 and 552 there are 512 symbols, as we need the symbols we will remove the Cyclic Prefix (CP) and put only these 512 symbols in an output array, starting at index 1. Then, we will find more 36 CP, like before, we don't need them, so we will remove these symbols and we will put the next 512 symbols of the input array with indexes between 589 and 1100, in the output array, now starting at the index 513. This process will repeat until the end of the input array, doing always the same, removing the CP and transferring the symbols to the output array, in sequential order.

In the end, the output array will contain only the 7168 symbols.

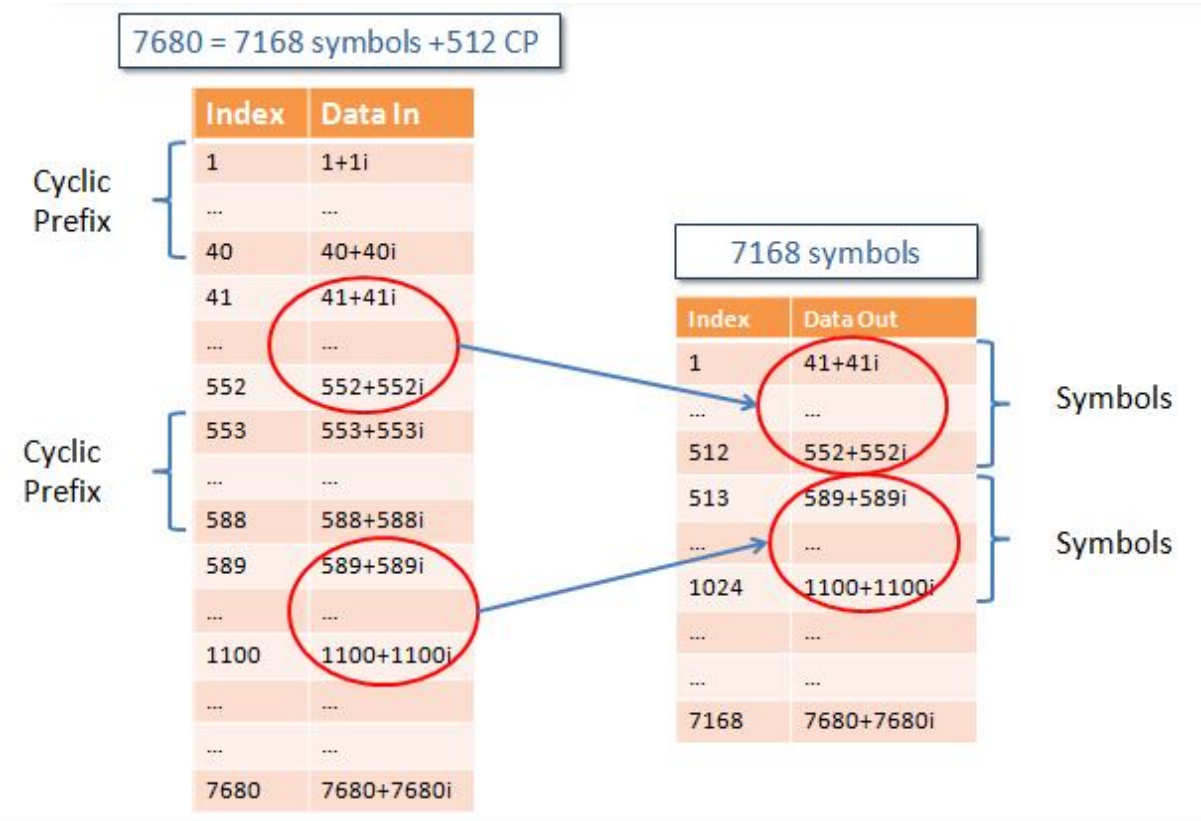

Figure 12 – Remove CP Scheme

The CUDA function to remove CP is now shown in Figure 13. The input parameters are the *UE_input_re* and *UE_input_im*, on other hand the output parameter is *y_rx_assembled_ifft*.

Then we need to calculate the index of the thread. For this we use that formula present in line 4. The offset and offset_2 are used to simplify the code (with this two variables it's not necessary to repeat all the code for more than one antenna). For example, if we have a two-dimensional grid, Number of antennas is one dimension and the 14 is the other. When the number of antennas is 1, offset is 0, but if the number of antennas is 2, offset will be 7680. The principle of offset_2 is the same, but this is used to control the index to be assigned. Let's assume that the index is 7681, nothing will happen because this is a cp, but if the index is 7721, it will execute the line 12, the offset is 7860 and offset_2 is 512, the index (7680+40>= index < 7680+551), *y_rx_assembled_ifft[7681]* will be assigned with *UE_input[7721]*, in other words for the seconds antenna *y_rx_assembled_ifft[1]* will be assigned with *UE_input[41]*.

```
1   __global__ void remove_cp(float *UE_input_re,float *UE_input_im,complex *y_rx_assembled_ifft)
2   {
3
4       long index=(blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x + threadIdx.x;
5       int offset;
6       int offset_2;
7
8
9       offset  = blockIdx.y*7680;
10      offset_2= blockIdx.y*512;
11
12      if(index>=40+offset && index <=551+offset)
13      {
14          y_rx_assembled_ifft[index-(40+offset_2)].re=UE_input_re[index];
15          y_rx_assembled_ifft[index-(40+offset_2)].im=UE_input_im[index];
16      }
17      if(index>=588+offset && index <=1099+offset)
18      {
19          y_rx_assembled_ifft[index-(76+offset_2)].re=UE_input_re[index];
20          y_rx_assembled_ifft[index-(76+offset_2)].im=UE_input_im[index];
21      }
22      /*
74      if(index>=6620+offset && index <=7131+offset)
75      {
76          y_rx_assembled_ifft[index-(476+offset_2)].re=UE_input_re[index];
77          y_rx_assembled_ifft[index-(476+offset_2)].im=UE_input_im[index];
78      }
79      if(index>=7168+offset && index <=7679+offset)
80      {
81          y_rx_assembled_ifft[index-(512+offset_2)].re=UE_input_re[index];
82          y_rx_assembled_ifft[index-(512+offset_2)].im=UE_input_im[index];
83      }
84
85  }
86
```

Figure 13 - Remove CP CUDA code

The C code is presented in Figure 14. This serial implementation is slower, but the code is more perceptible. In this way, for example, to define the offset_1 and offset_2, is

much easier than in CUDA, because we just need to put the index where we want to add the value, On other hand, in CUDA we have to define all conditions because everything is executed at the same time for every index.

```
void remove_cp(complex *UE_input,complex *y_rx_assembled_ifft,long size_r)
{
  int index;
  int offset_1=0;
  int offset_2=512;


      for (index=0;index<size_r;index++)
      {
        if(index==40 || index==3880)
        {
            offset_1 +=40;
            offset_2 +=512;
        }
        if(index==588 || index==1136 || index==1684 || index==2232 || index==2780 || index==3328 ||
        index==4428 || index==4976 ||index==5524 ||index==6072 || index==6620 || index==7168)
        {
            offset_1 +=36;
            offset_2 +=512;
        }
        if((index>=offset_1) && (index < (offset_2+offset_1)))
        {
            y_rx_assembled_ifft[index-offset_1].re=UE_input[index].re;
        }
      }

}
```

Figure 14 - Remove CP C code

4.2 FFT

The fast Fourier transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse, which reduces the number of computations needed for N points from 2N^2 to 2N*lg N, where lg is the base-2 logarithm [16].

The DFT has the power to transform one function into another, this transformation is called frequency domain representation [16].

The aim of Figure 15 is to explain how this FFT was implemented in this Mini receiver. Assuming just one antenna, we will have one array with 7168 symbols, and we will apply the FFT to each set of 512 symbols. For more antennas the behavior is the same, each antenna will have 7168 symbols.

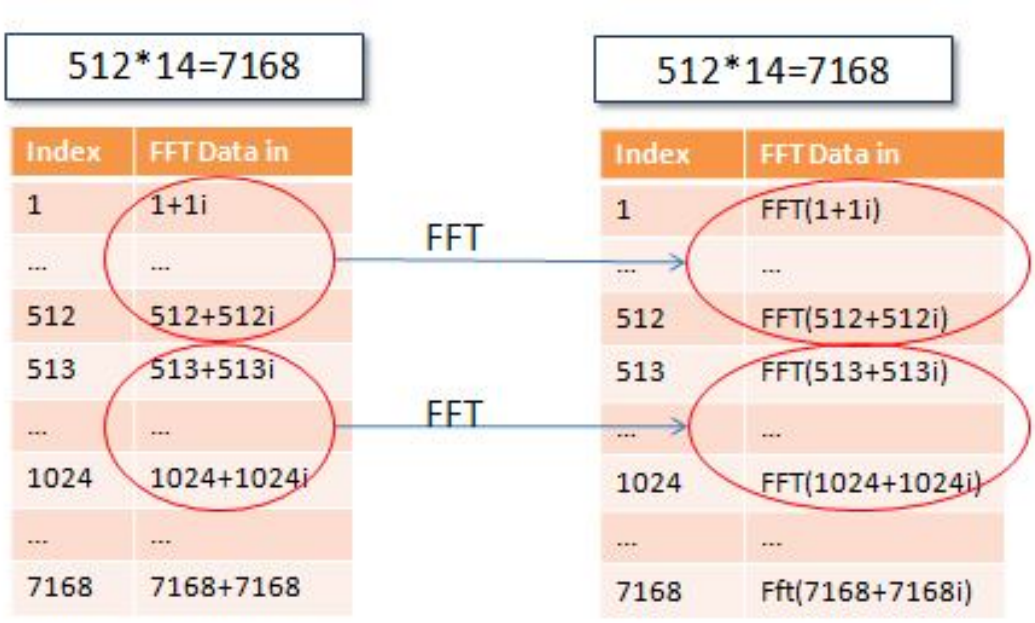

Figure 15- FFT scheme

In Figure 16 we can see the CUDA function used to implement the FFT. This function has 2 inputs, *cnst* and *y_rx_assembled_ifft* and one output *y_rx_assembled_shifted*.

At line 3 there is the index calculation. This formula to calculate is different from the last function because in this case the Grid has only one dimension.

This function is not fully parallel, for each antenna are created 512 *14 threads, but inside each thread, the *for* loop (from line18 to 26) is executed, and this loop is serial code. This algorithm will implement the FFT for each set of 512 symbols. This implementation in CUDA is much better than the C implementation presented in Figure 17, but an optimized library could be better. But it will be discussed later. Now, comparing the two functions, we can note that for the C code we will have three *for* loops in cascade, and it will be very slow.

```
__global__ void dft( float cnst,complex *y_rx_assembled_ifft,complex *y_rx_assembled_shifted)
{
    long index = blockIdx.x * blockDim.x + threadIdx.x;

    int i, j;
    int length=512;
    double arg;
    double cosarg,sinarg;
    int off;

    off=blockIdx.x*512;
    i=threadIdx.x;

    y_rx_assembled_shifted[i+off].re = 0;
    y_rx_assembled_shifted[i+off].im = 0;
    arg = -1.0 * 2.0 * 3.141592654 * (double)i / (double)length;

    for(j=0; j<length; j++)
    {
        cosarg = cos(j * arg);
        sinarg = sin(j * arg);
        y_rx_assembled_shifted[i+off].re += (y_rx_assembled_ifft[j+off].re * cnst * cosarg
                                            - y_rx_assembled_ifft[j+off].im * cnst * sinarg);
        y_rx_assembled_shifted[i+off].im += (y_rx_assembled_ifft[j+off].re * cnst * sinarg
                                            + y_rx_assembled_ifft[j+off].im * cnst * cosarg);
    }

}
```

Figure 16 - FFT CUDA code

```
148  void dft(float cnst, complex *y_rx_assembled_ifft,complex *y_rx_assembled_shifted,int nRX)
149  {
150    long int i,j,k;
151    int length=512;
152    double arg;
153    double cosarg,sinarg;
154    int off=0;
155
156
157
158            for(k=0;k<14*nRX;k++)
159            {
160                 for(i=0; i<length; i++)
161                 {
162                   y_rx_assembled_shifted[i+off].re = 0;
163                   y_rx_assembled_shifted[i+off].im = 0;
164                   arg = -1.0 * 2.0 * 3.141592654 * (double)i / (double)length;
165
166                   for(j=0; j<length; j++)
167                   {
168                     cosarg = cos(j * arg);
169                     sinarg = sin(j * arg);
170                     y_rx_assembled_shifted[i+off].re += (y_rx_assembled_ifft[j+off].re * cnst * cosarg
171                                                          - y_rx_assembled_ifft[j+off].im * cnst * sinarg);
172                     y_rx_assembled_shifted[i+off].im += (y_rx_assembled_ifft[j+off].re * cnst * sinarg
173                                                          + y_rx_assembled_ifft[j+off].im * cnst * cosarg);
174                   }
175                 }
176                 off+=512;
177            }
178
179  }
```

Figure 17 – FFT C code

4.3 Circshift

After the FFT we need to rotate the signal, the folowing Figure 18 aims to explain how this works. For each antenna, we have one array FFT Data Out with 7168 symbols that needs to be shifted. After this, it will be saved in a new array Shift Data Out with 7168 symbols too.

We will take each set of 512 symbols and then rotate them 150 times, supposing the first 512 symbols, we will take the firsts 362 symbols of the FFT Data Out and we will put this symbols in the position 151 of the array Shift Data Out, then we take the next 150 symbols of the FFT Data Out, from index 362 to 512 and we will put this symbols in the array Shift Data Out starting at the index 1. This way, the first 512 symbols are now rotated, for the rest of the symbols, we just need to do this process 14 times, if the antennas's number is more than one, the process can be applied too.

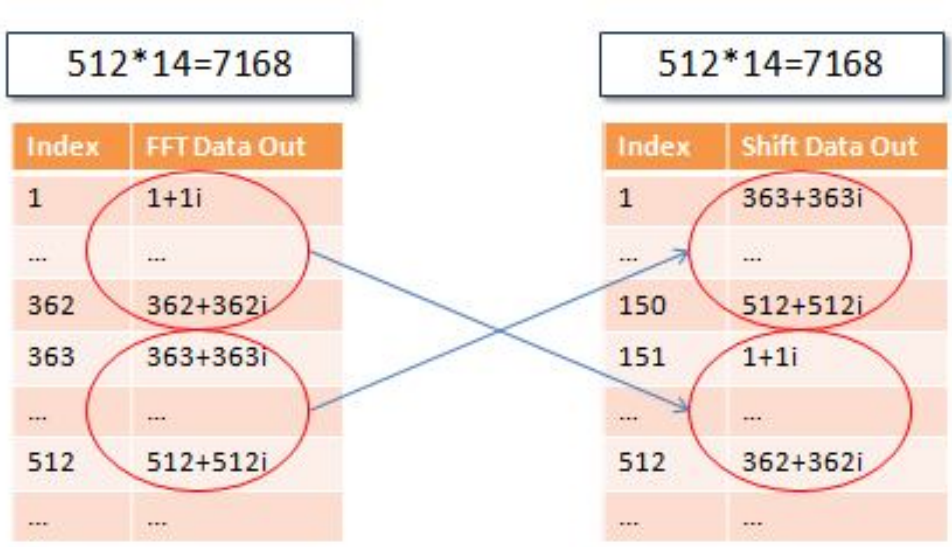

Figure 18 - CIRCSHIFT Scheme

After this brief explanation, let us analyze the CUDA code presented in Figure 19. This function has one input parameter *y_rx_assembled_shifted* and an output parameter *y_rx_assembled_padded*. At line 6 the index is assigned, and at line 8, the offset_2 is declare with *blockIdx.y* *7168. This kernel has grids with 2 dimensions, where the *blockDim.y* is the number of antennas, and the *blockIdx.y* takes a value between 0 and Number of antennas. For the first antenna the *blockIdx.y* is 0, two antennas is 1 and so on. So, this offset_2 is used to control the flow of the threads, because for different threads, different actions will be taken and with this offset_2 is possible to know in which antenna the execution is taking place. The offset has the same purpose, but this offset is global and this will change with the location of the threads and antennas.

For example, if the index number is 10, 10 is less than 512, so, the execution will be at line 10, the offset_2 is zero because the index of the antenna 1 is zero, the offset will be zero too, now the execution will be at line 67, index is 10 (index<150), *y_rx_assembled_padded[10]* will be assigned with *y_rx_assembled_shifted[372]*, the rotation is complete.

```
1  __global__ void padded(complex *y_rx_assembled_shifted,complex *y_rx_assembled_padded)
2  {
3
4      int offset=0;
5      int offset_2=0;
6      long index=(blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x + threadIdx.x;
7
8      offset_2=blockIdx.y*7168;
9
10     if(index < 512 + offset_2)
11     {
12         offset = 0 + offset_2 ;
13     }
14     if(( index>=512 + offset)   && (index<1024 + offset_2))
15     {
16         offset = 512 + offset_2 ;
17     }
18     /*      ...
63     if((index>=6656 + offset)  && (index<7168 + offset_2))
64     {
65         offset = 6656 + offset_2 ;
66     }
67     if(index<(150+offset))
68     {
69         y_rx_assembled_padded[index].re=y_rx_assembled_shifted[index+362].re;
70         y_rx_assembled_padded[index].im=y_rx_assembled_shifted[index+362].im;
71     }
72     else
73         {
74             y_rx_assembled_padded[index].re=y_rx_assembled_shifted[index-150].re;
75             y_rx_assembled_padded[index].im=y_rx_assembled_shifted[index-150].im;
76         }
77
78
79 }
```

Figure 19 - CIRCSHIFHT CUDA code

In C, Figure 20, once again, the code is slower but easier to understand. The *for* loop will execute size times (512*14**n_antennas*), then we just need to know in which index it is necessary to increment the offset. In previous example, if the index is 10, at line 20, index is 10 (index<150), *y_rx_assembled_padded[10]* will be assigned with *y_rx_assembled_shifted[372]*.

```
1  void padded(complex *y_rx_assembled_shifted,complex *y_rx_assembled_padded,int size)
2  {
3  int offset,x;
4  for(x=0;x<size;x++)
5                  {
6
7                      if(x==512 || x==1024 || x==1536 || x==2048 || x==2560 || x==3072 || x==3584
8                          || x==4096 || x==4608 || x==5120 || x==5632 || x==6144 || x==6656 )
9                      {
10                         offset += 512;
11                     }
12                     /*
20                     if(x<(150+offset))
21                     {
22                         y_rx_assembled_padded[x].re=y_rx_assembled_shifted[x+362].re;
23                         y_rx_assembled_padded[x].im=y_rx_assembled_shifted[x+362].im;
24                     }
25                     else
26                     {
27                         y_rx_assembled_padded[x].re=y_rx_assembled_shifted[x-150].re;
28                         y_rx_assembled_padded[x].im=y_rx_assembled_shifted[x-150].im;
29                     }
30
31                 }
32
33 }
```

Figure 20 - CIRCSHIFT C code

4.4 Remove zero DC Carrier

DC carrier belongs to the Null subcarriers, in these subcarriers there is no transmission, these are frequency guard bands. The DC subcarrier is the subcarrier whose frequency is equal to the RF centre frequency of the transmitting station [17].

The Figure 21 is a basic scheme to explain how to remove the zero frequency. For one antenna, Zero Data IN is an input array with 7168 symbols, and Zero Data Out is the output array with 4200 symbols. We need to remove the zero frequency, with this, we will analyze the first 150 symbols. They are not zero frequency, so we will need to transfer these symbols to the array Zero Data Out, starting at index 1. Then we will find one zero frequency, thus we will remove this symbols. After that we will take the next 150 symbols and move it to the output array, beginning at the index 151, therefore, the index 151 of the output array will contain the data from the index 152 of the input array. Afterwards, the next symbols from index 301 to 512 will be removed. From index 513 to 662 the symbols will be moved to output array, starting at index 301, then we will find another zero to remove. This process will repeat until the end of the input array. At the end, the output array will have only the correct symbols without frequency guard bands.

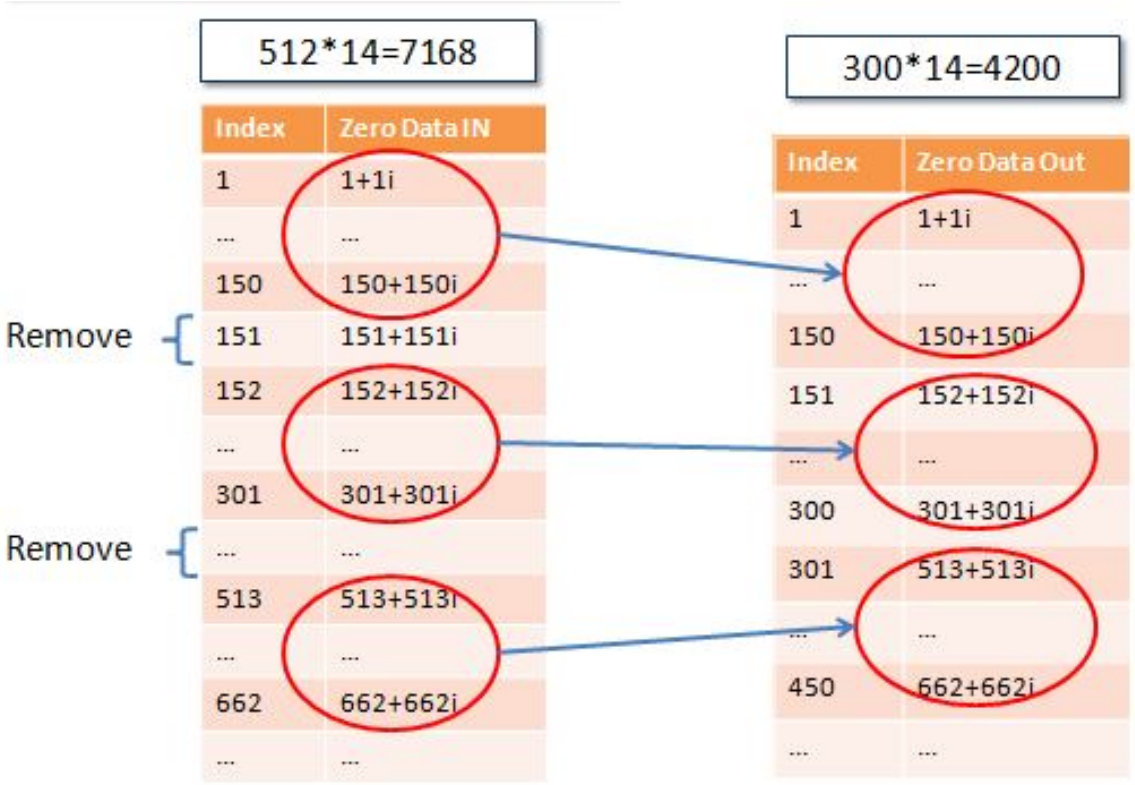

Figure 21 - Remove zero DC Carrier scheme

The CUDA code to remove the zero frequency is presented in Figure 22. The input for this function is one complex array *y_rx_assembled_padded*, and the output is *y_rx_assembled_re* and *y_rx_assembled_im*. After the variables are declared, we assign the thread index. This function uses 4 auxiliary variables, offset, offset_2, offset_3, offset_4, this variables help to control the flow of the threads.

The kernel has grids with 2 dimensions and blocks with one dimension, the blockIdx.y represent the index of the antennas, the *BlockDim* is 300, which means that each block will have 300 threads. The total number of threads will be *gridDim.x* gridDim.y* BlockDim*.

For example, assuming that the index is 10, the *blockIdx.y* is 0, because is the first antenna, consequently, offset_3 and offset_4 will be zero. The execution will be at line 15, offset=0 and offset_2=0, hereafter the execution will move to line 150, *y_rx_assembled_padded_re[10]* will be assigned with *y_rx_assembled_padded[10].re*.

If the index is 151, *y_rx_assembled_padded_re[151]* will be assigned with *y_rx_assembled_padded[152].re*.

```
1  __global__ void remove_dc_zero(complex *y_rx_assembled_padded,
2                  float *y_rx_assembled_re,float *y_rx_assembled_im)
3  {
4      int offset=0;
5      int offset_2=0;
6      int offset_3=0;
7      int offset_4=0;
8
9      long index=(blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x + threadIdx.x;
10
11     offset_3 = blockIdx.y*4200;
12     offset_4 = blockIdx.y*2968;
13
14
15     if(index<300 + offset_3)
16     {
17         offset =0 + offset_3;
18         offset_2 =0 + offset_4;
19     }
20     /*
82     if(index>=3900 + offset_3 && index<4200 + offset_3)
83     {
84         offset =3900 + offset_3;
85         offset_2 =2756 + offset_4;
86     }
87     if(index<(150+offset) && index >= offset )
88     {
89         y_rx_assembled_re[index]=y_rx_assembled_padded[index+offset_2].re;
90         y_rx_assembled_im[index]=y_rx_assembled_padded[index+offset_2].im;
91     }
92     if(index>=(150+offset) && index <(300+offset) )
93     {
94         y_rx_assembled_re[index]=y_rx_assembled_padded[index+1+offset_2].re;
95         y_rx_assembled_im[index]=y_rx_assembled_padded[index+1+offset_2].im;
96     }
97
98 }
```

Figure 22 - Remove zero DC carrier CUDA code

At the Figure 23, we will find the C code to remove zero frequency. This code compared with the CUDA code, this implementation is simpler and has less lines of code, being serial, it has the advantage to compute the offset easier, but in the end the important thing here is the time and CUDA is faster. Assuming the index=301, the execution flow will pass through line 11 at index 300, and the offset will take the value 300 and the offset2 will take the value 212. Hereafter the execution will move to line26, and will check if the index is minor than offset+150(301 <150+300). The condition is

true, *y_rx_assembled[301].re* will be assigned with *y_rx_assembled_padded[513].re*. As we can see the zero frequency was removed.

```
void remove_dc(complex *y_rx_assembled_padded,complex *y_rx_assembled,int size_z)
{
    int offset    =0;
    int offset_2  =0;
    int offset_4  =0;
    int offset_3  =0;

    int index;

    for(index=0;index<4200;index++)
    {
        if( index==300  ||  index==600  ||  index==900  ||  index==1200 ||
            index==1500 ||  index==1800 || index==2100  ||  index==2400 ||
            index==2700 ||  index==3000 || index==3300 || index==3600   ||
        index==3900                                                     )
        {
            offset +=300;
            offset2 +=212;
        }
        if(index<150+offset)
        {
            y_rx_assembled[index].re=y_rx_assembled_padded[index+offset2].re;
            y_rx_assembled[index].im=y_rx_assembled_padded[index+offset2].im;
        }
        if(index>(149+offset) && index <(300+offset) )
        {
            y_rx_assembled[index].re=y_rx_assembled_padded[index+1+offset2].re;
            y_rx_assembled[index].im=y_rx_assembled_padded[index+1+offset2].im;
        }
    }
}
```

Figure 23 - Remove zero DC Carrier C code

4.5 Disassemble Reference Symbols

Reference Symbols (RS) are inserted into the time-frequency grid of OFDM to enable coherent channel estimation in the user device. RS are located in the 1st OFDM symbol (1st RS) and 3rd to last OFDM symbol (2nd RS) of every sub-frame, as shown in Figure 24 for an LTE system with one antenna in normal CP mode . Furthermore, reference symbols are staggered in both time and frequency. The response of the channel on subcarriers bearing the reference symbols can be computed directly. Interpolation is used to estimate the channel response on the remaining subcarriers [7] [18].

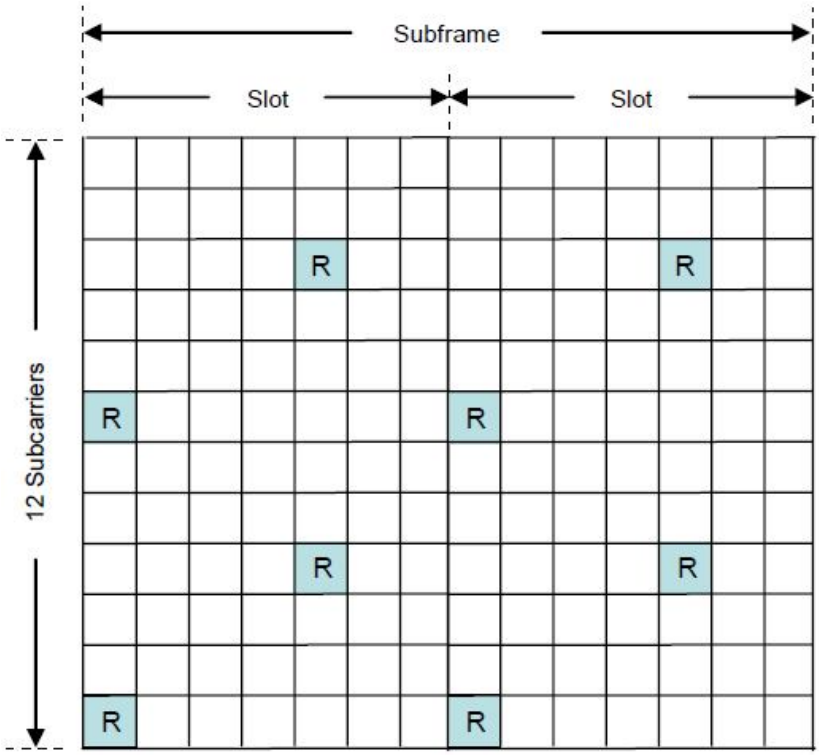

Figure 24 – Location of reference symbols [18]

After this brief explanation about reference symbols, let's analyze the Figure 25, to explain the basic scheme of disassemble. In this case, there are two input arrays both with 4200 symbols, the first one is Data In and the second is the array Pilots. This array has only zeros and ones, the reference symbols are represented for 1. Data out is the output array with 200 symbols for each pilot, for each antennas there is a set of 4 pilots.

To disassemble the reference symbols, we need to search for one symbol "1" in the Pilots, for example in the index 2 we found "1", thus, we will go to the Data In and move this data to the output array Data out, then in the index 8 we will find another "1", the process will be same, in the end for one antenna we will have 4 output arrays, the first 2 with 200 symbols, the others will have 100 symbols each because there are less reference Symbols for more than 2 antennas.

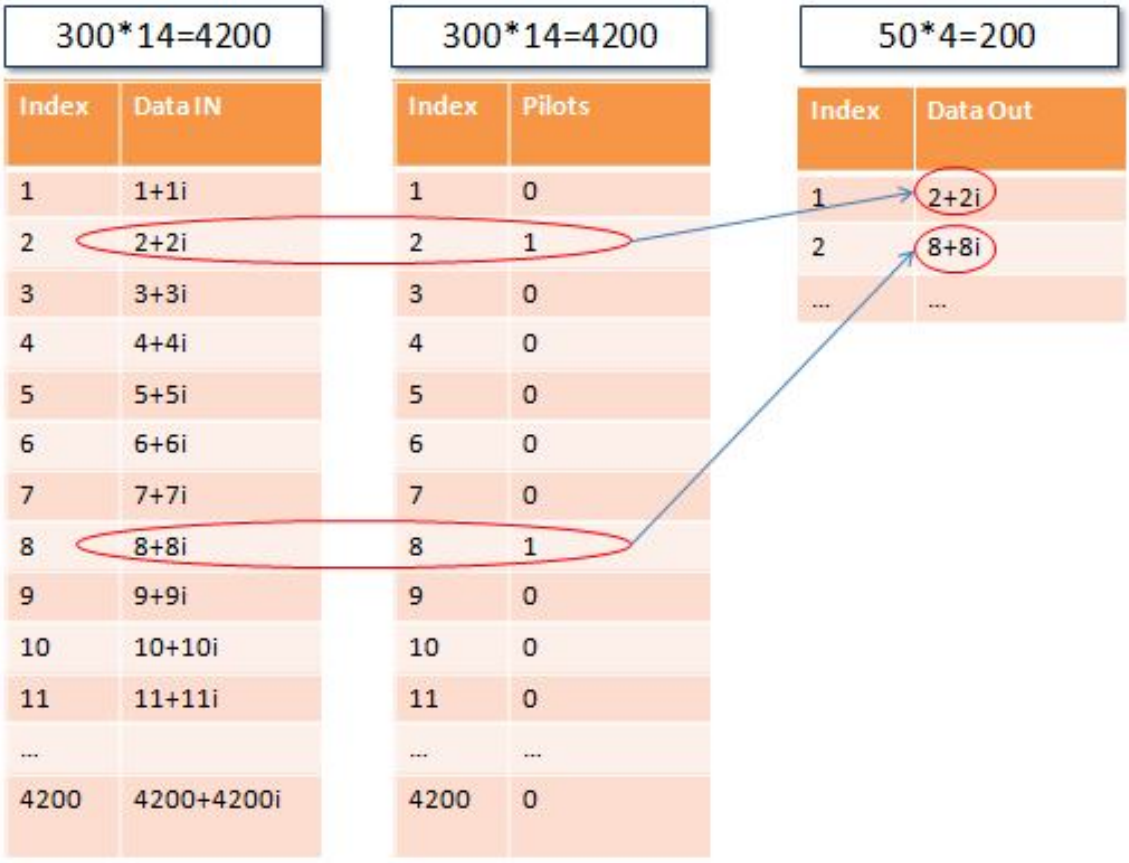

Figure 25 - Disassemble Reference Symbols scheme

The following CUDA implementation, Figure 26, has some serial code, is not only parallel programming. The inputs are the array *RefMapping* , *y_rx_assembled_re* and *y_rx_assembled_im*, the out puts are *rx_ref_symbols_re* and *rx_ref_symbols_im*.

At line 6, the index is calculated, and then at line 9 and10 the offset and offset_2 are assigned. The offset is assigned with *blockIdx.y* *4200, where *blockIdx.y* is the index of the antenna, the offset_2 is assigned with *blockIdx.x**4200, where the *blockIdx.x* is the index of the pilot. The auxiliary variable aux at line 11, is used to control the index of the output array. With this implementation the serial code presented from line 13 to 22, will execute 16 times in parallel. The *gridDim.x* and *gridDim.y* is 4, the *blockDim* is just 1, with this the number of threads is *gridDim.x * gridDim .y * blockDim*, a total of 16 threads.

For example, for the first antenna and first Pilot, the offset and offset_2 will be 0, the index will be 0. Entering in for loop, 16 threads are executing, but for the first thread, it will analyze every indexes of the input variable *RefMapping* and if one 1 is found, this data will be saved in the output array in the correct index, after this the variable aux will

be increment. For the second thread the index is 1, offset is 0 and offset_2 is 4200, entering in the for loop, this time, we will try to find the symbols 1 but in the *RefMapping* of the index 4200 to 8400, it means the second pilot, then when the symbols is found, the data will be save in the output array but now starting at index 200.

```
__global__ void disassemble(bool *RefMapping,float *y_rx_assembled_re,float *y_rx_assembled_im,
                            float *rx_ref_symbols_re,float *rx_ref_symbols_im)
{
    int aux=0;
    int i,offset,offset2;
    long index=(blockIdx.y * gridDim.x + blockIdx.x) * blockDim.x + threadIdx.x;


    offset= blockIdx.y*4200;
    offset2= blockIdx.x*4200;
    aux=index*200;

    for(i=0;i<4200;i++)
    {   if(RefMapping[i+offset2]==1)
        {
            rx_ref_symbols_re[aux]=y_rx_assembled_re[i+offset];
            rx_ref_symbols_im[aux]=y_rx_assembled_im[i+offset];
            aux +=1;
        }
    }

}
```

Figure 26 -Disassemble Reference Symbols CUDA code

In Figure 27 the implementation in C is shown, comparing with the previous implementation, the C code has higher processing time and more lines of code, it will analyze all indexes of the 4 antennas, one at a time, searching for the symbols 1.

```
1   void disassemble(bool *RefMapping,complex *y_rx_assembled,complex *rx_ref_symbols,int size_
2   {
3       int index;
4       int aux;
5       int i;
6       for(index=0;index<size_pi;index++)
7          {
8              if(index<1*4200)
9              {   aux=0;
10                 for(i=0;i<4200;i++)
11                 {   if(RefMapping[i]==1)
12                     {
13                         rx_ref_symbols[aux].re=y_rx_assembled[i].re;
14                         rx_ref_symbols[aux].im=y_rx_assembled[i].im;
15                         aux +=1;
16                     }
17                 }
18             }
19             /*
175            if(index>=15*4200 &&index < 16*4200)
176            {   aux=3000;
177                for(i=0;i<4200;i++)
178                {   if(RefMapping[i+12600]==1)
179                    {
180                        rx_ref_symbols[aux].re=y_rx_assembled[i+12600].re;
181                        rx_ref_symbols[aux].im=y_rx_assembled[i+12600].im;
182                        aux +=1;
183                    }
184                }
185            }
186
187
188        }
```

Figure 27 - Disassemble Reference Symbols C code

4.6 Mini receiver host

The host function is very important for the CUDA programming Model because the execution starts always in this host function. Only with this function is possible to call the kernel (CUDA functions).

In this sub-chapter the host function of this Mini Receiver will be shown and commented.

In Figure 28, from line 1 to 6, we have the header of the function, then as was said before at line 12, the device is chosen. At line 18 starts the variable declaration, for example, the variable size will change depending on the number of antennas. At line 34 the pointer declaration is started, these pointers are used for dynamic allocation.

```
1  __host__ void MR(
2                   float *h_UE_input_re,float *h_UE_input_im,float *LTE_params_Nfft,
3                   float *LTE_params_Ntot,bool *h_RefMapping,int *h_nRX ,
4                   float *h_y_rx_assembled_re,float *h_y_rx_assembled_im,
5                   float *h_rx_ref_symbols_re,float *h_rx_ref_symbols_im
6                   )
7  {
8      ////////////////////////////
9      ////    Choose device   ////
10     ////////////////////////////
11
12     CUDA_SAFE_CALL(cudaSetDevice(0));
13
14     ////////////////////////////////////////
15     ////        Variables Declaration    ////
16     ////////////////////////////////////////
17
18     long size   =   N_ROWS*N_COLUMNS           *   h_nRX[0];
19     /*      ...
30     ////////////////////////////////////////
31     ////        Pointers Declaration     ////
32     ////////////////////////////////////////
33
34     complex *h_y_rx_assembled_ifft;
35     /*      ...
39     float   *d_UE_input_re;
40     float   *d_UE_input_im;
41     /*      ...
```

Figure 28 - Host Function part 1

The Figure 29 shows the Memory allocation and the Thread dimension. The CPU memory allocation starts at line 55 and GPU memory allocation starts at line 65, in both cases. At line 81 starts the Thread Dimension, *dimblock* has one dimension with 512 threads and *dimGrid* is a two dimensions grid, in x 14 blocks and in y *n_RX* blocks.

```
51     ////////////////////////////////////////
52     ////        CPU Memory allocation    ////
53     ////////////////////////////////////////
54
55     h_y_rx_assembled_ifft       =   (complex_t*)malloc(size  * sizeof(complex_t));
56     /*      ...
61     ////////////////////////////////////////
62     ////        GPU Memory allocation    ////
63     ////////////////////////////////////////
64
65     cudaMalloc((float   **)&d_UE_input_re,  size_r  * sizeof(float));
66     cudaMalloc((float   **)&d_UE_input_im,  size_r  * sizeof(float));
67     /*      ...
77     ////////////////////////////////////////
78     ////        Thread Dimension         ////
79     ////////////////////////////////////////
80
81     dim3 dimBlock   (N_ROWS);
82     /*      ...
88     dim3 dimGrid    (N_COLUMNS      , h_nRX[0]);
89     /*      ...
```

Figure 29 - Host Function part 2

To finish, at Figure 30, there is the kernel invocation and the memory release. First thing to do before one invokes the kernel, is to copy memory from the host to device, because, without this step the input variables from the kernel function will be empty. This step is executed at line 99 and 100, and then we just need to invoke the kernel, with the correct thread dimension and parameters. The kernels remove_cp, DFT, padded and remove_dc_zero, are invoked one after another, at line 114 the memory is copied from the device to host, because the variable *y_rx_asssembled* will be used as output of the host function. For the last kernel disassemble, it is necessary to copy the Pilots to the kernel, after the kernel invocation, rx_*ref_symbols* will be used like an output parameter in the host function.

```
96      ////////////////////////////////////////////////////
97      ////////////remove cyclic prefix//////////////////////
98      ////////////////////////////////////////////////////
99      cudaMemcpy(d_UE_input_re, h_UE_input_re,size_r*sizeof(float), cudaMemcpyHostToDevice);
100     cudaMemcpy(d_UE_input_im, h_UE_input_im,size_r*sizeof(float), cudaMemcpyHostToDevice);
101     remove_cp <<<dimGrid_r,dimBlock_r>>>(d_UE_input_re,d_UE_input_im,d_y_rx_assembled_ifft);
102     ///////////////////////
103     ///////  fft /////////
104     ///////////////////////
105     dft <<<dimGrid_d,dimBlock_d>>>(cnst,d_y_rx_assembled_ifft,d_y_rx_assembled_shifted);
106     ////////////////////////////////////
107     /////// Shifting the signal  //////
108     ////////////////////////////////////
109     padded <<<dimGrid,dimBlock>>>(d_y_rx_assembled_shifted,d_y_rx_assembled_padded);
110     ////////////////////////////////////
111     ////  remove zero DC carrier ////
112     ////////////////////////////////////
113     remove_dc_zero <<<dimGrid_z,dimBlock_z>>>(d_y_rx_assembled_padded,d_y_rx_assembled_re,d_y_rx_assembled_im);
114     cudaMemcpy(h_y_rx_assembled_re,d_y_rx_assembled_re,size_z*sizeof(float),cudaMemcpyDeviceToHost);
115     cudaMemcpy(h_y_rx_assembled_im,d_y_rx_assembled_im,size_z*sizeof(float),cudaMemcpyDeviceToHost);
116     ////////////////////////////////////
117     ////        Disassemble        ////
118     ////////////////////////////////////
119     cudaMemcpy(d_RefMapping, h_RefMapping,size_p * sizeof(bool), cudaMemcpyHostToDevice);
120     disassemble <<<dimGrid_di,dimBlock_di>>>(d_RefMapping,d_y_rx_assembled_re,d_y_rx_assembled_im,
121                                              d_rx_ref_symbols_re,d_rx_ref_symbols_im);
122
123     cudaMemcpy(h_rx_ref_symbols_re,d_rx_ref_symbols_re,size_di*sizeof(float),cudaMemcpyDeviceToHost);
124     cudaMemcpy(h_rx_ref_symbols_im,d_rx_ref_symbols_im,size_di*sizeof(float),cudaMemcpyDeviceToHost);
125     ////////////////////////////////////////
126     ////           Free memory          ////
127     ////////////////////////////////////////
128     cudaFree(d_UE_input_re);
129     /*
144 }
```

Figure 30 - Host Function part 3

4.7 Mini Receiver MEX

As said before, MEX files are a method to call CUDA functions directly from MATLAB. Figure 31 is the function used for this Mini Receiver, and this function will be explained. The libraries are included in the first 3 lines, we need the *mex.h*, *cuda.h* and Main_mini_receiver.cu (Host function). Afterwards, the pointers for the input and output variables will be declared. The fetch for the input variables starts at line 26, the real part of *UE_input* will be assigned with the argument 0 of the MEX Function. An output array with 3 dimensions is required, at line 36 one vector is created and at line 39 the first output argument of the MEX file is assigned with the tridimensional array, after that, *rx_assembled_re* will be assigned with this first output argument of the MEX function. To conclude, the CUDA function is invoked at line 54.

```
1  #include "mex.h"
2  #include <cuda.h>
3  #include "Main_Mini_receiver.cu"
4
5  void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
6  {
7      //      Input variables
8      float *UE_input_re;
9      float *UE_input_im;
10     /*      ...
17     //      Output variables
18     float  *rx_assembled_re;
19     float  *rx_assembled_im;
20     /*      ...
25     //  fetch Input variables
26     UE_input_re = (float *)(mxGetPr(prhs[0]));
27     UE_input_im = (float *)(mxGetPi(prhs[0]));
28     /*
35     //  fetch Output variables
36     int dimensions[3] = {300, 14, 0};
37     dimensions[2] = nRX[0];
38
39     plhs[0] = mxCreateNumericArray( 3, dimensions, mxSINGLE_CLASS, mxCOMPLEX);
40
41     rx_assembled_re = (float*)mxGetPr(plhs[0]);
42     rx_assembled_im = (float*)mxGetPi(plhs[0]);
43     /*      ...
53     //       Function call
54     MR(UE_input_re,UE_input_im,Nfft,Ntot,RefMapp,nRX,rx_assembled_re,
55                rx_assembled_im,rx_ref_symbols_re,rx_ref_symbols_im);
56 } /* mexFunction */
57
```

Figure 31 - MEX Function

5 Results

To evaluate the performance of CUDA implementation, the method used was the time of processing the kernel compared with others implementations, namely C and MATLAB code.

Each algorithm was run multiple times and was considered the mean time between runs, for all situations. During this chapter, the obtained results will be shown and compared.

5.1 C Versus MATLAB

The Table 1 shows the average processing time (in milliseconds) between serial implementations, C and MATLAB code.

In Figure 32, the x-axis represents the number of antennas and the y axis the processing time. One may observe through average processing time in Table 1, in my implementations, the MATLAB can be 250 times faster than my C code, this large difference is due to the fact that MATLAB have the best and most optimized libraries, while the C code implemented has no optimizations.

Table 1 – Comparison between C and MATLAB

Number Of received antennas	C Code (Time ms)	Matlab (Time ms)
1	611 ms	2.71 ms
2	1222 ms	6.29 ms
3	1875 ms	7.43 ms
4	2440 ms	9.99 ms

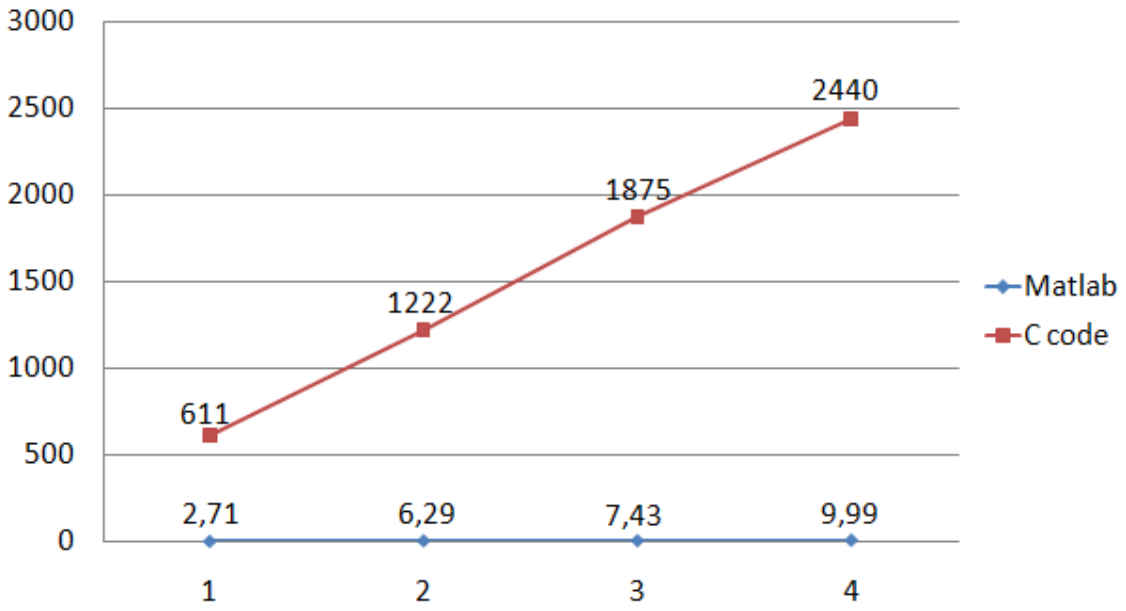

Figure 32 – MATLAB vs. C code Graphic Comparison

5.2 C versus CUDA

Table 2, shows the average processing time (in milliseconds) between serial and parallel implementations, C and CUDA. The CUDA implementation processing time is divided in two parts, kernel and Malloc. The processing time in C will increase with the number of antennas, but in both cases the CUDA implementation is faster than C, being 31 times faster in case of 4 antennas.

Table 2 – Comparison between C and CUDA

Number Of received antennas	C (Time ms)	C CUDA (Time ms)	Kernels CUDA (Time ms)	Malloc CUDA (Time ms)
1	611 ms	70 ms	8 ms	62 ms
2	1222 ms	74 ms	14 ms	60 ms
3	1875 ms	78 ms	16 ms	62 ms
4	2440 ms	78 ms	16 ms	62 ms

The Figure 33 is a graphical comparison between the two implementations. We have the kernel (green line) ranging from 8 to 16 ms as the number of antennas, at memory allocation Malloc (purple line), we can see that memory allocation takes more time than

the implementation of the kernel, however the two things cannot be separated, because is necessary to allocate memory to run a CUDA function. Still a very important fact is that for the first time that a CUDA function is executed, the memory allocation Malloc, takes 334ms. This happens because in the first run the device needs to boot, but still this is not worrisome, since you can boot your device before performing our CUDA functions.

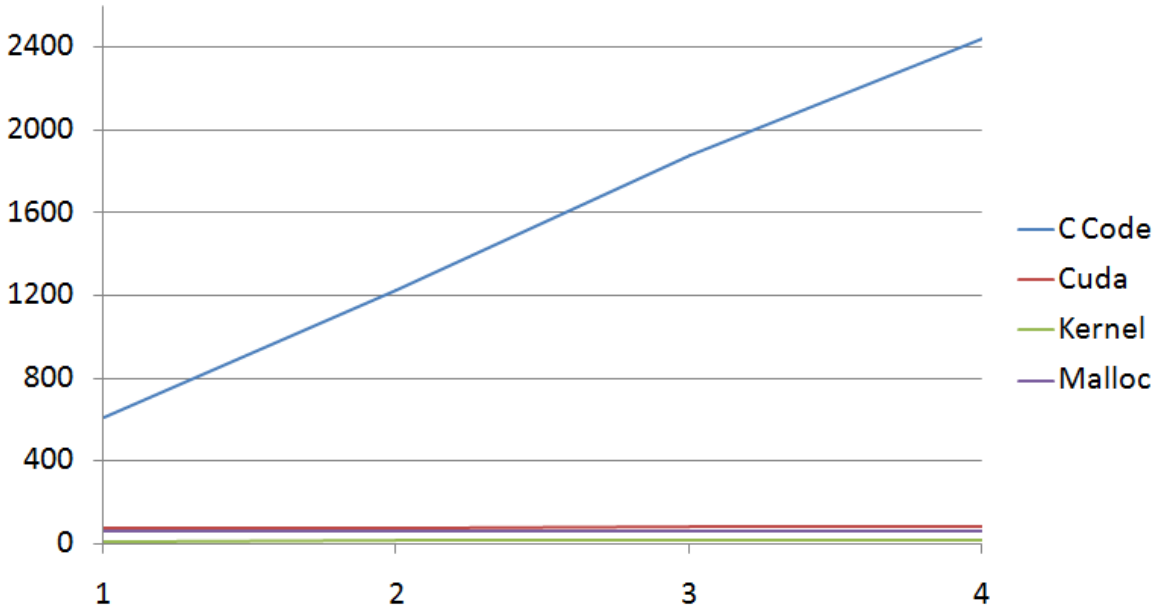

Figure 33 - C vs. CUDA code Graphic Comparison

5.3 MATLAB versus CUDA

Table 3 shows the average processing time (in milliseconds) between serial and parallel implementations, MATLAB and CUDA code. Like was said before, each algorithm was run multiple times and was considered the best time between runs, for all situations.

Table 3 - Comparison between MATLAB and C CUDA

Number Of received antennas	MATLAB (Time ms)	C CUDA (Time ms)	Kernels CUDA (Time ms)	MALLOC CUDA (Time ms)
1	2.71 ms	70 ms	8 ms	62 ms
2	6.29 ms	74 ms	14 ms	60 ms
3	7.43 ms	78 ms	16 ms	62 ms
4	9.99 ms	78 ms	16 ms	62 ms

In Figure 34, has represented the interval between 1 and 4 antennas. Across the range, MATLAB is faster. However, is important look at these graphic details. We can see that increasing the number of antennas, the performance of CUDA increases, on the other hand, MATLAB will increase the processing time. However, the memory allocation takes 60ms, delaying CUDA.

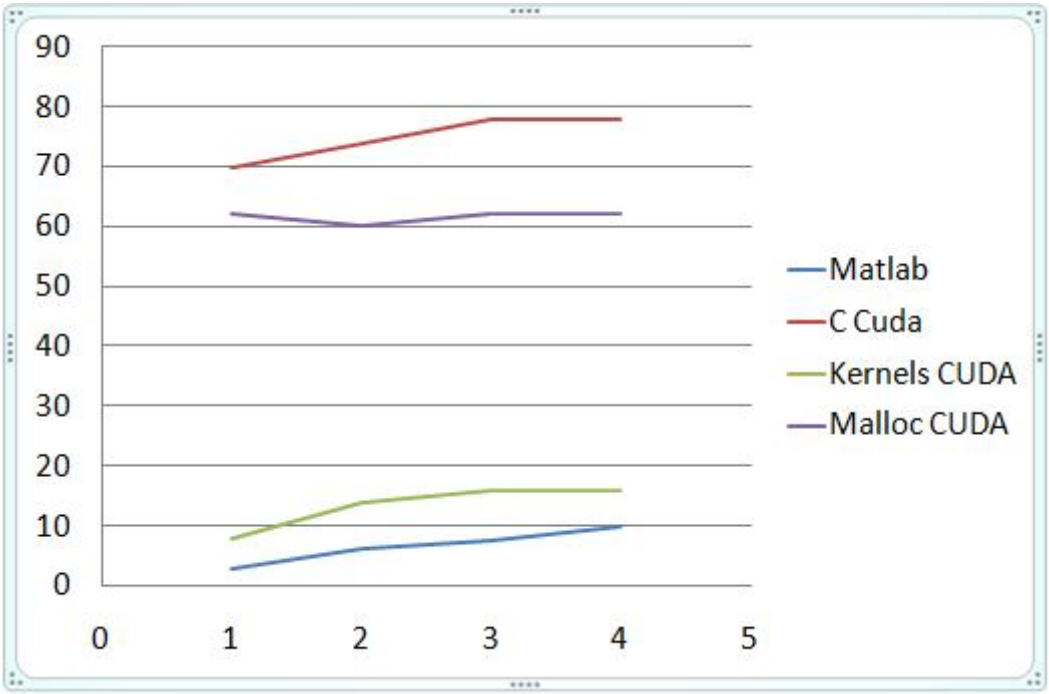

Figure 34 - MATLAB vs. CUDA code Graphic Comparison

5.4 Testing the performance of CUDA

With the results from one to four antennas, we can say that MATLAB is faster, but nevertheless, increasing the processing, CUDA begins to take advantage of his great power of parallelism. As such, it would be inappropriate to make this sentence based only on these results, in a way that we would need to increase the processing load in order to test CUDA's performance. The only way to do that would be to add more antennas to the simulator implemented in CUDA. However, this is only an academic level simulation, made only to test CUDA's processing power, because the real mini receiver only supports four antennas.

During this subchapter we will examine Table 4 in detail, separating the results, thereby realizing everything that happens in this implementation to evaluate the performance of CUDA.

Looking briefly at the table, we see that for fewer than 40 antennas MATLAB is better than CUDA, however increasing the number of antennas, CUDA begins to gain an advantage due to its parallelism power. For 40 antennas or more CUDA is rather than MATLAB. For about 120 antennas gets to be 3.35 times faster, that value will increase with the number of antennas.

Table 4 – Test of CUDA performance

Number Of received antennas	Matlab (Time ms)	C CUDA (Time ms)	Kernels CUDA (Time ms)	Malloc CUDA (Time ms)
1	2.71 ms	70 ms	8 ms	62 ms
2	6.29 ms	74 ms	14 ms	60 ms
3	7.43 ms	78 ms	16 ms	62 ms
4	9.99 ms	78 ms	16 ms	62 ms
10	68 ms	132 ms	63 ms	68 ms
20	142.7 ms	207 ms	127 ms	80 ms
40	380.1 ms	339 ms	258 ms	79 ms
80	1584 ms	660 ms	567 ms	93 ms
100	2552 ms	729 ms	652 ms	77 ms
120	2878.8 ms	854 ms	754 ms	99 ms

The Figure 35 is a graphical representation of the two implementations, MATLAB (blue line) and CUDA (red line). CUDA is still composed of the kernel (green line) and Malloc (purple line).

For better analysis, this chart is divided, compared and studied. The division will be made in order to analyze the most important points of the graph. On the x axis we have the number of antennas ranging between 1 and 120, on the y axis we have the processing time of the two implementations in milliseconds.

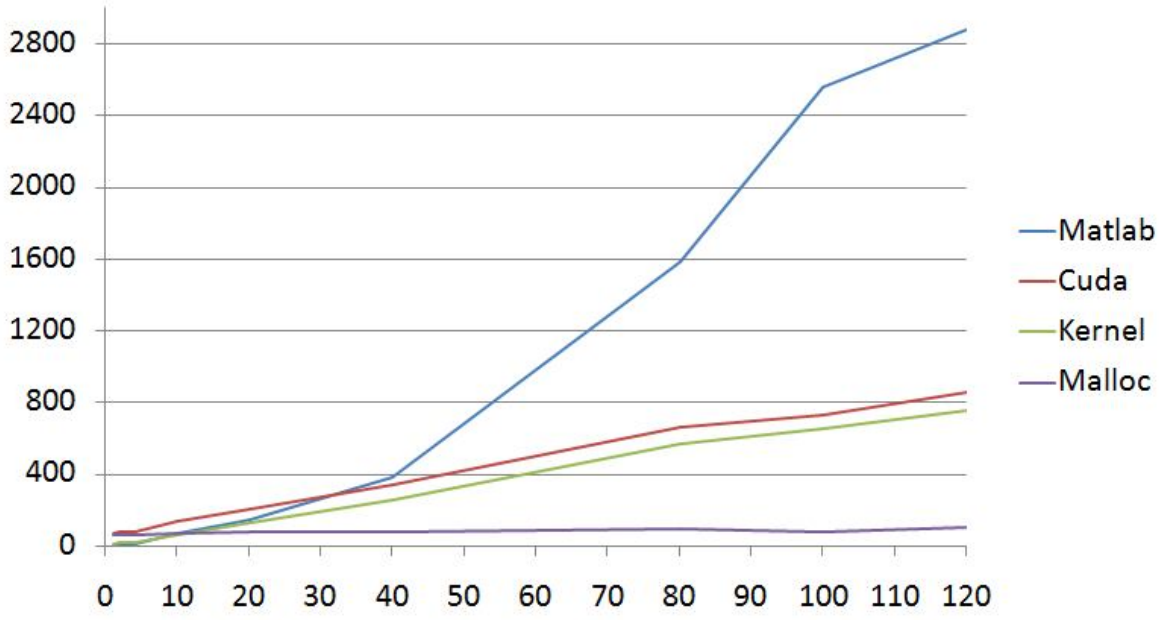

Figure 35 – Test of CUDA performance 1

In Figure 36, has represented the interval between 1 and 12 antennas. Across the range, MATLAB is faster. However, is important look at these graphic details. We can then see that 1-7 antennas, the kernel takes a little longer than MATLAB, when the number of antennas is 7, this is reversed and the kernel becomes faster. However, the memory allocation takes 60ms, delaying CUDA.

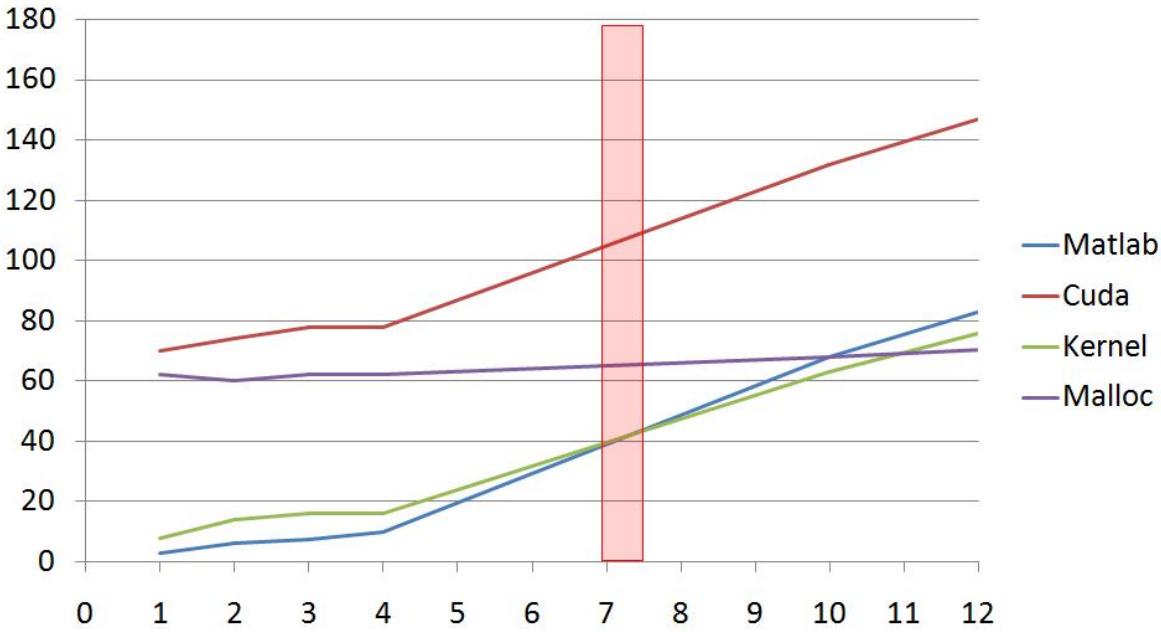

Figure 36 - Test of CUDA performance 2

In the image appear two very important points, and we will examine it to better understand this implementation.

For the first point, we observe that when we have between 10 and 12 antennas, this is the time where the processing time exceeds the processing time memory allocation. Wear Malloc takes so long, this is a very important point, since although the CUDA take longer than MATLAB, this only happens because of the long memory allocation.

The second point to consider is when we have between 32 and 33 antennas. This is the turning point where the CUDA becomes faster, the set, kernel and Malloc, after this point has a processing time less than MATLAB.

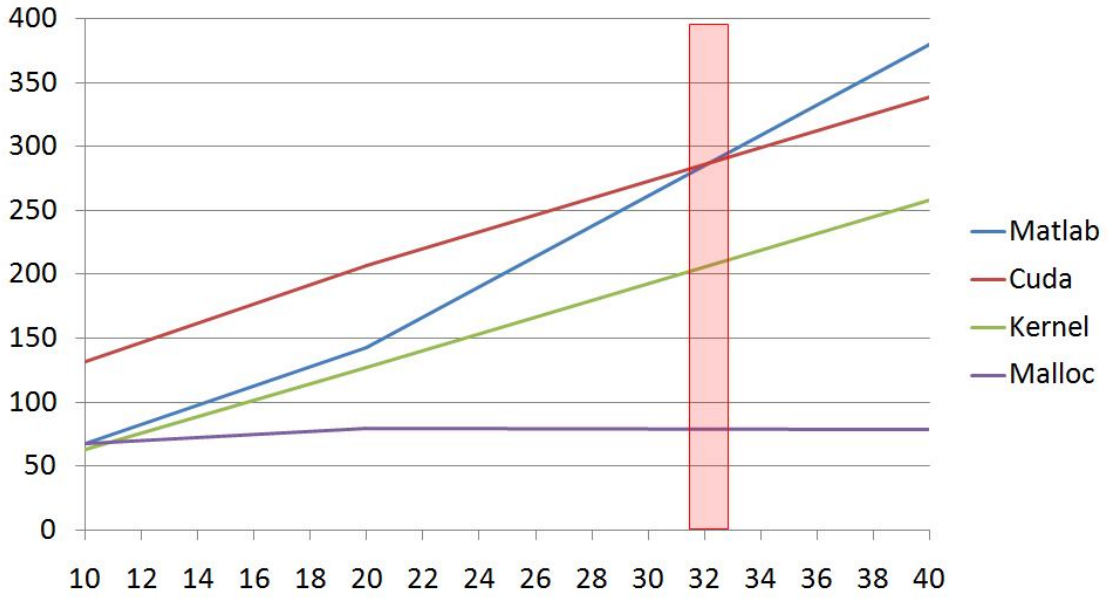

Figure 37 - Test of CUDA performance 3

Finally, we have the Figure 38 that shows the graph between 30 and 120 antennas. This graph shows that the processing time of MATLAB increase, while the processing time of CUDA stands to increase very slowly.

When we have 120 antennas CUDA is 3.35 times faster, if we continue to increase the number of antennas this figure would continue to increase while there is available memory on the graphics card to run all threads required.

One can also observe that for a large processing time memory allocation, thus no longer interferes with both the total processing time, because the total time is now much higher.

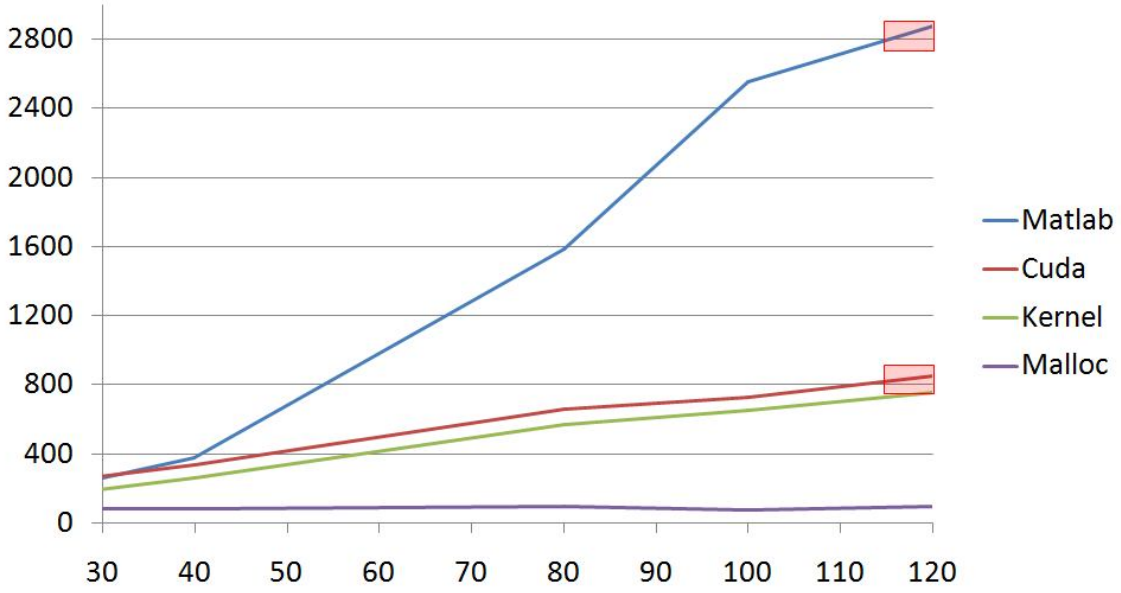

Figure 38 - Test of CUDA performance 4

6 Conclusions

6.1 Conclusion

In this work, we used a combination of MATLAB and CUDA, examining the CUDA programming model and NVIDIA GPUs architecture, as well as MEX Files. We tested two implementations, the first one using serial C code and the other one parallel code, using CUDA.

Comparing C with CUDA code, there is no doubt that the parallel implementation is much better, being 31 times faster for only four antennas.

Although the C code is quite similar to CUDA code, in fact, the way you think about programming for serial implementation is quite different from parallel. The parts implemented in CUDA for 4 antennas take, in MATLAB, approximately 10ms. On other hand, just to allocate memory on the GPU, it takes 60ms. Thus, for only 4 antennas, there is no enough processing to take advantage of CUDA.

The primary goal of this project was to use CUDA with MATLAB to speed up the computation time of the LTE Mini receiver. This goal was not achieved as desired. However, good conclusions can be taken from it.

In conclusion, we can then conclude that CUDA is much better when we have large data processing, and, in this specific case, this does not apply. As we have little data processing, even though the code is fairly optimized, only for the memory allocation, CUDA exceeds the processing time of MATLAB implementation. To take advantage of the processing power of CUDA would be necessary to implement all parts of the simulator, just for this case is not worth to use CUDA.

6.2 Future Work

CUDA, as reported earlier, has a large computing power, which may still be explored, as well as the, simulator that can be also improved.

As future work, we suggest some optimizations in the code implemented, including the implementation of the function CUDAFFT because it is an optimized library provided by NVIDIA, and also disassemble the function, because this code is not executed fully in parallel.

Both the channel estimator and feedback calculation should be implemented. Even for the first function, channel estimation, the result should be similar to the results obtained in this work because it has the same problem stated above, the function takes only 12ms, and allocation continues to take about 60ms. For feedback calculation, it will be different, since there is plenty of processing that can be parallelized, and this function takes about 450ms in MATLAB, which gives a greater margin of operation.

Acronyms

3GPP	3rd Generation Partnership Project
API	Application Programming Interface
CP	Cyclic Prefix
CPU	Central processing unit
CUDA	Compute Unified Device Architecture
DFT	Discrete Fourier Transform
EDGE	Enhanced Data rates for GSM Evolution
FFT	Fast Fourier Transform
GPU	Graphics processing unit
GSM	Global System for Mobile communications
ISI	Inter Symbol Interference
LTE	Long Term Evolution
MIMO	Multiple Input Multiple Output
MU-MIMO	Multi-User MIMO
OFDM	Orthogonal Frequency Division Multiplexing
PHY	Physical
QAM	Quadrature Amplitude Modulation
RB	Resource Block
RE	Resource Element
RS	Reference symbols
SDMA	Spatial Domain Multiple Access
SNR	Signal-to-noise ratio
SU-MIMO	Single User MIMO
UE	User equipment
UMTS	Universal Mobile Telecommunications System

Bibliography

[1] '**LTE Encyclopedia**', February 2011, [Online]. Available: http://sites.google.com/site/lteencyclopedia/home)

[2] '**GSA - The Global mobile Suppliers Association**', February 2011, [Online] Available: http://www.gsacom.com/news/gsa_240.php4

[3] Zyren J.,(2007), '**Overview of the 3GPP Long Term Evolution Physical Layer',** Freescale Semiconductor, Inc.

[4] Sesia S., Toufik I., Baker M., (2009), '**LTE – The UMTS Long Term Evolution: From Theory to Practice**', John Wiley & Sons Ltd., United Kingdom.

[5] Gesbert D., Kountouris M., Robert W. Heath Jr., Chae C.,Sälzer T.,(2007), '**From Single User to Multiuser Communications: Shifting the MIMO Paradigm**', Mobile Communications Department, Eurecom Institute.

[6] '**LTE Toolbox–Chanel Estimation**', Copyright 2009-2011 Steepest Ascent Ltd, February 2011, [Online] Available: http://www.steepestascent.com/content/mediaassets/html/LTE/Help/Channel%20Estimation.html

[7] Šimko M., (2009), '**Channel Estimation for UMTS Long Term Evolution**', Vienna University of Technology, Institute of Communications and Radio-Frequency Engineering

[8] NVIDIA Corporation, (2010),'**NVIDIA CUDA C programming Guide(Version 3.2)**', NVIDIA, 2010

[9] Yano L.,(2010),'**Avaliação e comparação de desempenho utilizando tecnologia CUDA**', Department of Computer Science and Statistics, State University Paulista "Júlio de Mesquita Filho"

[10] Vasconcellos F., (2009), '**PROGRAMANDO COM GPUs: PARALELIZANDO O METODO LATTICE-BOLTZMANN COM CUDA'** University of Rio Grande do Sul, Institute of Informatics, course of computer science

[11] Ghadialy Z.,'**3G and 4G Wireless Blog: MIMO schemes in LTE**', February 2011, [Online] Available: http://3g4g.blogspot.com/2009/02/mimo-schemes-in-lte.html

[12] Kirk D. and Hwu W., (2006-2008),**' CUDA textbook: CUDA Programming Model** (Chapter 2)', NVIDIA and UIU.

[13] Fatica M. and Jeong W., '**Accelerating MATLAB with CUDA**', NVIDIA, University of Utah.

[14] NVIDIA Corporation, (2007), '**Accelerating MATLAB with CUDA Using MEX Files**' NVIDIA, 2007.

[15] '**Redes PLC Internas**', February 2011, [Online] Available: http://www.gta.ufrj.br/grad/04_1/redesplc/3.html

[16] '**Wolfram MathWorld**: **Fast Fourier Transform**', February 2011, [Online] Available: http://mathworld.wolfram.com/FastFourierTransform.html

[17] '**Wimax Technology for broadband wireless access: OFDM Transmission (chapter 5.2)**', February 2011, [Online] Available: http://etutorials.org/Networking/wimax+technology+broadband+wireless+access/

[18] Rayal F.,'**EE Times: An overview of the LTE physical layer--Part II'**, February 2011, [Online] Available: http://www.eetimes.com/electronics-news/4200493/An-overview-of-the-LTE-physical-layer-Part-II.

[19] Dash S. and Dora B., (2009), '**CHANNEL ESTIMATION IN MULTICARRIER COMMUNICATION SYSTEMS**', Electronics and Communication Engineering National Institute of Technology Rourkela.

[20] Tiiro S., Ylioinas J., Myllylä A., Juntti M,(2009), '**IMPLEMENTATION OF THE LEAST SQUARES CHANNEL ESTIMATION ALGORITHM FOR MIMO-OFDM SYSTEMS'**, International ITG Workshop on Smart Antennas International ITG Workshop on Smart Antennas, (WSA 2009, February), 16-18.

[21] Trepkowsk R, (2004), '**Channel Estimation Strategies for Coded MIMO**' Virginia Polytechnic Institute and State University, Institute of Electrical Engineering.

[22].Schwarz S., Mehlführer C., M., Rupp M., (2010), '**Calculation of the Spatial Preprocessing and Link Adaption Feedback for 3GPP UMTS/LTE**', Institute of Communications and Radio-Frequency Engineering, Vienna University of Technology.

[23] '**ALTHOS - simplifying Knowledge**', February 2011, [Online] Available: http://www.wirelessdictionary.com/Wireless-Dictionary-Channel-Quality-Indicator-CQI-Definition.html

[24] Holma H.and Toskala A.,(2009), '**LTE for UMTS – OFDMA and SC-FDMA Based Radio Access'** , John Wiley & Sons Ltd., United Kingdom.

[25] Schwarz S., Wrulich M., Rupp M., (2010), '**Mutual Information based Calculation of the Precoding Matrix Indicator for 3GPP UMTS/LTE**', International ITG Workshop on Smart Antennas (WSA 2010), 55-58.

[26] '**3G LTE Tutorial - 3GPP Long Term Evolution'**, March 2011, [Online] Available: http://www.radio-electronics.com/info/cellulartelecomms/lte-long-term-evolution/3g-lte-basics.php

[27] Hontzeas A. ,(2009),'**EVOLUTION IN COMMUNICATION –Long Term Evolution**', Antonis Hontzeas for free public use.

[28] Leung A. and You Y., (2008), '**Semi-Blind Equalization for OFDM using Space-Time Block Coding and Channel Shortening**', Multidimensional Digital Signal Processing.

[29] Intel (2011),'**Wireless Networking: MIMO**', March 2011, [Online] Available: http://www.intel.com/support/wireless/sb/cs-025345.htm

[30] Pagès A., (2009),'**A Long Term Evolution Link Level Simulator**', Universitat Politècnica de Catalunya

[31] Rayal F., (2010), '**LTE in a Nutshell: The Physical Layer**', Telesystem Innovations

[32] Švirák M. and Šebesta V., '**SIMULATION OF NON-ORTHOGONAL SPACE-TIME BLOCK CODED SYSTEM WITH CHANNEL ESTIMATION ERRORS'**, Institute of Radio electronics, Brno University of Technology