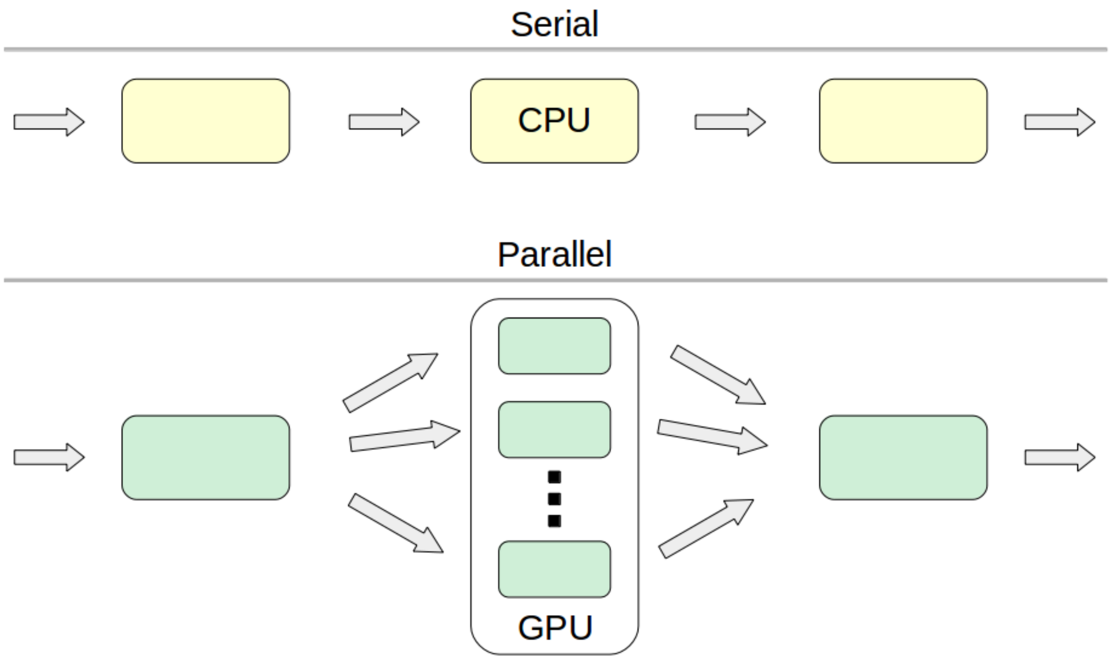
CHALMERS
UNIVERSITY OF TECHNOLOGY



Parallel computing for 5G

Evaluating the use of CUDA for downlink parallel decoding of synchronization signals

Master's thesis in Communication Engineering

Sofia Sundin and Ulrika Yring

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Master's thesis EX013/2018

Parallel computing for 5G

Evaluation of CUDA for downlink parallel decoding of synchronization signals

Sofia Sundin and Ulrika Yring

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg, Sweden 2018

Parallel computing for 5G
Evaluating the use of CUDA for downlink parallel decoding of synchronization signals
Sofia Sundin and Ulrika Yring

Supervisor: Erik Sandgren, Combitech
Andreas Buchberger, Department of Electrical Engineering
Examiner: Thomas Eriksson, Department of Electrical Engineering

Master's Thesis EX013/2018
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Parallel computing for 5G
Evaluating the use of CUDA for downlink parallel decoding of synchronization signals
Sofia Sundin and Ulrika Yring
Department of Electrical Engineering
Chalmers University of Technology

Abstract

During recent years the data traffic demands for the mobile networks have increased rapidly. In the upcoming standard Fifth Generation Cellular Network (5G) the industry will attempt to address this growing demand and many other new demands that have risen over the past few years. One researched topic in 5G is wireless fiber where the aim is to reach fiber speeds within a wireless network. This entails that User Equipments (UEs) must process signals much faster. This thesis project investigated the use of a Graphics Processing Unit (GPU), specifically a Compute Unified Device Architecture (CUDA) platform, for the decoding of three synchronization signals. The three synchronization signals are included in a 5G standard established by Verizon. To be able to have a reference to compare to the parallel implementation a system was also conducted in serial on a CPU. The results of the investigation showed that the parallel system performed, in the majority of test cases, significantly faster than the serial program. One of the largest time consumers in the parallel system was that the calculations did not utilize even half of the GPU resources. This could be drastically decreased using CUDA streams, where using two streams almost cut the time of decoding in half.

Keywords: 5G, wireless fiber, CUDA, GPU, synchronization signals, CUDA streams, complexity, throughput, latency

Acknowledgements

We would like to offer the most sincere thanks to the supervisors of this thesis, Erik Sandgren and Andreas Buchberger. Your guidance has helped us greatly in the work to accomplish this thesis.
A big thanks to Combitech for allowing us to do this rewarding thesis work. Finally we would like to thank our examiner Thomas Eriksson.

Sofia Sundin, Ulrika Yring, Gothenburg, August 2018

Abbreviations

3GPP	Third Generation Partnership Project
5G	Fifth Generation Cellular Network
5GTF	5G Terrestrial Fiber
CN	Cellular Network
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
ESS	Extended Synchronization Signal
FFT	Fast Fourier Transform
GPU	Graphics Processing Unit
HSDPA	High Speed Downlink Package Access
HSPA	High Speed Packet Access
ICT	Information and Communications Technology
IFFT	Inverse Fast Fourier Transform
ITU-R	International Telecommunication Unions Radio Sector
LTE	Third Generation Partnership Project Long Term Evolution
MP	Multiprocessors
OFDM	Orthogonal Frequency-Division Multiplexing
PSS	Primary Synchronization Signal
SSS	Secondary Synchronization Signal
UE	User Equipment
V5GTF	Verizon 5G Technical Forum

List of Symsbols

B_{max}	Maximum number of blocks that can run concurrently on a GPU for maximum block size
$B_{\text{max,MP}}$	Maximum blocks per multiprocessor
B_{min}	Maximum number of blocks that can run concurrently on a GPU for minimum block size
d	Synchronization sequence for one OFDM symbol
d^l	Synchronization sequence for one OFDM symbol dependent on the OFDM symbol
Δf	Subcarrier spacing
k	Subcarrier
l	OFDM symbol
n_{cores}	nVIDIA Cores
N_{CP}	Cyclic prefix length
$N_{\text{ID}}^{(1)}$	Cell identity group
$N_{\text{ID}}^{(2)}$	Sector identity
$N_{\text{ID}}^{\text{cell}}$	Cell identity
n_{MP}	Number of Multiprocessors
T_{core}	Number of threads per CUDA core
$T_{\text{max,MP}}$	Maximum number of threads per multiprocessor
T_s	Sample time
T_{symb}	Symbol time
$T_{\text{max,block}}$	Maximum threads per block symbol time
$T_{\text{min,block}}$	Minimum threads per block
T_{warp}	Warp size

Contents

List of Figures

List of Tables

1
Introduction

As introduction to this thesis report, background, aim, scope, methodology and report structure will be presented.

1.1 Background

Communication has always been an important part of society, and as society has gone through an industrial and technical revolution a large part of the communication has become digital. It all started with a Morse code message and has now expanded into 1G, 2G, 3G and the most recent standard 4G.
The upcoming generation of Cellular Networks (CNs) is the 5G. Since data traffic demands are growing more than ever the industry aspires to achieve higher data rates, higher bandwidth and lower latency for the new generation of mobile broadband. This is made possible with the new high-band spectrum technique [1],[2]. As the data rates increases and environmental challenges are emerging, the need for energy efficient systems enlarges. This is one of the bigger challenges to take into consideration in the design of 5G [3].
During late 2016 Verizon 5G Technical Forum (V5GTF) released the latest version of their 5G standard, which is a collaboration between a number of large companies within the telecommunication industry whose mission is to come up with a common standardization, whereof Ericsson is one [4]. This standard is called 5G Terrestrial Fiber (5GTF) and has the purpose to replace fiber links with wireless links. As these wireless links are expected to have similar performance as the fiber, it is sometimes referred to as wireless 5G fiber.
In Sweden today 46% of households and workplaces do not have access to fiber [5]. When installing wired fiber to the customer, a cable has to be dug down. This creates economic challenges as prices range from 14 000 - 25 000 SEK. For larger distances, these numbers only escalate [6],[7]. Additionally, running a fiber creates many other logistical challenges since a lot of ground and sometimes roads have to be dug up. Many of these problems could be either decreased or averted with the help of the 5G wireless fiber. Currently different types of solutions are tested. This is where this masters thesis plays a role.

1.2 Aim

The aim of this thesis project is to investigate if the performance of the downlink decoding of the synchronization signals in 5G could be improved by using a CUDA graphics card. The main goal is to investigate whether or not lower system latency and higher throughput could be obtained using a GPU compared to a Central Processing Unit (CPU).

1.3 Scope

In this project the mission is to analyze if and how the performance of decoding a 5G downlink signal can be improved by parallelizing the decoding using a CUDA GPU. The data being decoded is simulated as a recording of what the base station would send from baseband to the radio antenna. This signal is discrete and has no noise.
When analyzing the performance, the most essential aspect will be the throughput and latency of the decoder. In other words, is it possible to increase the number of samples handled by the system during a fixed amount of time. Other aspects such as costs and complexity will also be discussed. The initial goal for this thesis project is to measure throughput during the decoding using the synchronization signals (primary synchronization signal, secondary synchronization signal and extended synchronization signal) from the V5GTF standard and not signals with actual messages.

1.3.1 Limitations

Since the scope of the project is to decode the synchronization signals broadcasted to a device only a signal decoder will be modeled and not the encoder.
Considering that the major part of this project is to see whether or not throughput and latency are affected by parallel computing there is no need to add fading to the recorded signal. In this 5G standard a complex type of beamforming is used in the Massive-MIMO antenna but with the data recordings provided there is no need to construct the beamforming mechanism for this case. Since the aim is to find the time difference between serial and parallel decoding and that the time for this project is limited the system is not going to perform synchronization. The frames sent into the system will thereby be limited in variation and nothing will be done to the reading of the signal with the knowledge of one or more synchronization signals. To measure the throughput as well as the latency only one subframe will be assessed at a time. Variations in timing based on the signal length will not be studied.

1.4 Methodology

To conduct this project a significant amount of literature studies (5G standard, programming and communication techniques) had to be completed in order to find all the parameters and methods to decode 5G signals according to the V5GTF standard as well as how to do calculations on the GPU using the CPU. The major part of the project was spent writing programs in C and CUDA C. The GPU was directed using C programming from the CPU. The CPU instructed the GPU, through a program in the CPU, to do the calculations and thereafter copy all the data blocks to the

GPU. The GPU calculated the blocks and thereafter copied back the data to the CPU.
The throughput was, at the end of the project, measured using programs in C that revealed time usage. From this information it is possible to retrieve the throughput and latency.

1.5 Ethics and Environment

In a world where the greater part of the western world is connected during all hours of every day, there are still a large amount of places that have no internet available. Some may argue that since there is a requirement for higher energy efficiency of the network this could help places that have limited access to the electrical grid, since this means that diesel generators and such other money consuming tools have to be used to gain internet access and lowering the energy use would also lower this cost [8].

2
Fifth Generation Wireless Systems

This chapter, covering the fifth generation of wireless systems, includes a brief explanation of the theory behind 5G followed by a thorough explanation of the Verizon 5G Terrestrial Fiber Standard, where the main focus lies upon three synchronization signals.

2.1 System properties and technical information

The evolution of the CNs has been gradual over, so far, four generations. The first system was the 1G, this was an analog voice-only system. During the evolution of 2G, whilst still being focused on voice, the CN became digital and the GSM technique was introduced. The EDGE technology was introduced in what later became 2.5G. The third generation, 3G, was the first CN to be centralized around mobile broadband. 3G was also the start of the organization Third Generation Partnership Project (3GPP), whose purpose is to make cellular technology standards global. In 3G High Speed Downlink Package Access (HSDPA) was introduced and after in what is now called 3.5G came High Speed Packet Access (HSPA). In a later 3G phase, also known as 3.9G, Third Generation Partnership Project Long Term Evolution (LTE) was introduced [8].
In 4G circuit-switched voice is no longer supported and the updated LTE technique LTE-advanced is used. This means that the CN can now reach much higher data rates and account for many types of usages, including machine to machine type communication. However, the need for high performing Information and Communications Technology (ICT) systems keeps growing and now it is time for 5G, also known under the International Telecommunication Unions Radio Sector (ITU-R) project IMT-2020, which is the United Nation's international collaboration of setting requirements, without setting any technical specifications, for 5G [8],[9].
Despite LTE being in an initial period of deployment, the communication industry has been researching and developing the fifth generation of mobile communication for a rather long period of time. The aspiration for 5G is to wirelessly connect a large amount of different devices. There are applications that would benefit from connectivity, one example is traffic safety and control. For this application there is a demand for both extreme reliability and low latency. Meanwhile, there are other applications which have very different requirements. One example of this is massive machine type communication where the data rates and the amount of data per device are relatively low whilst demanding very low power consumption to save battery. Another aspect of 5G is that it needs to cover future, yet unknown, fields

of applications which makes for other requirements [8].

"Connectivity will be provided essentially anywhere,
anytime to anyone and anything" [8]

The expected raise in active devices in 5G entails that a requirement for higher data rates has been set, this is one of the main properties of 5G. To achieve all of the mentioned improvements there are three properties that will be revised. The first is the spectral efficiency, which will be improved but only slightly since it is already so optimized. The second property is to increase the bandwidth, this does on the other hand also introduce a lot of challenges since the free bandwidth that will be used is very high (3-6 GHz and 6-30 GHz) which will affect the quality of the signal over larger distances and through obstacles like thick walls and rain. Because of this, new radio access techniques are needed as a compliment to LTE-advanced. The third one is to increase the number of base stations deployed whilst decreasing the distance between them [8].

2.1.1 Orthogonal Frequency-Division Multiplexing

Orthogonal Frequency-Division Multiplexing (OFDM) is a version of multicarrier modulation and was introduced in CNs when designing 4G and is kept when designing 5G [8]. As the name of OFDM entails, the bandwidth of a channel can be thought of as many smaller sub-channels with frequency spacing Δf, i.e., the channel's allocated bandwidth has been divided into a number of subcarriers. When the subcarrier signal, s_{k} has not been modulated yet it is defined as

$$s_k = \begin{cases} \mathrm{e}^{j2\pi k\Delta f t} & \forall t \in [0, T_{\text{symb}}] \\ 0 & \text{elsewhere} \end{cases} \tag{2.1}$$

where k is the subcarrier and symbol time T_{symb}, is calculated using $T_{\text{symb}} = \frac{1}{\Delta f}$. When sending the signal the transmitter modulates all subcarriers independent from one another multiplying the subcarrier signal with an information signal according to

$$s(n) = \sum_{n=\infty}^{\infty} c_{ki} s_k(t - iT_{\text{symb}}) \tag{2.2}$$

where c_{ki} is the information symbol at index i [10].

2.1.2 Fast Fourier Transform

To be able to modulate a signal according to OFDM the signal has to be handled in both frequency and time domain, this can be achieved using a Fast Fourier Transform (FFT). An FFT is a time efficient way of converting a time domain signal into a frequency based one as follows

$$S(k) = \sum_{n=0}^{N-1} s(n) e^{\frac{-j2\pi nk}{N}} \tag{2.3}$$

where N represents the length of the time domain signal $s(n)$ and k represents the number of frequency representations, where $n = k$. Transforming a signal from the frequency domain to the time domain is performed via the Inverse Fast Fourier Transform (IFFT) which is calculated according to

$$s(n) = \sum_{k=0}^{N-1} S(k)e^{\frac{j2\pi nk}{N}} \tag{2.4}$$

where the IFFT is used to create a time domain signal to transmit and the FFT will, in the receiver, transform the received time-domain signal to a frequency-domain signal.

2.1.3 Cross correlation

To be able to find the correlation, and in its turn find the best match between signals, a cross correlation is used. The cross correlation is the correlation of two signals as follows

$$(f \star g)(n) = \sum_{m=-\infty}^{\infty} f^*(m)g(m+n) \tag{2.5}$$

giving as an output a numeric value of how much two signals correspond to each other, one value for each signal fitting over the other, giving

$$n = 3 \cdot \text{length}(m) - 2 \tag{2.6}$$

number of outputs from the cross correlation in equation 2.5. The higher the cross correlation the more alike the two signals are, making this a useful tool when trying to classify signals.

One alternative way of calculating the cross correlation is by using the FFT, the IFFT and the complex conjugate

$$f \star g = \text{IFFT}\Big(\text{FFT}(f) \cdot \text{FFT}^*(g)\Big) \tag{2.7}$$

and where the signals have to be padded with at least the size of the signal number of zeros, giving an alternative way of implementing the cross correlation.

2.1.4 Cell search

For a device to be able to communicate with the 5G network it needs to synchronize with a base station in the network. The range of where a UE can communicate to a base station is called a cell. The network consist of 3 types of sector identities, referred in this report as $N_{ID}^{(2)}$, 168 cell identity groups, referred in this report as $N_{ID}^{(1)}$ leading to 504 cell identities, see figure 2.1.

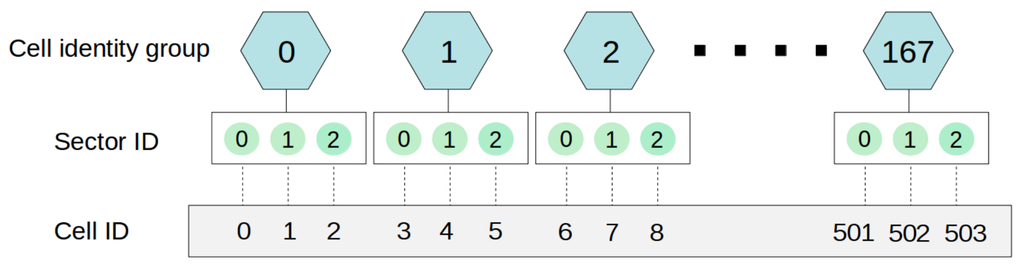


Figure 2.1: The relation between cell identity, cell identity group and sector identity

The cell search is not only executed when first trying to connect to the network but continuously during its run time, to handle mobility of the UEs. To obtain $N_{ID}^{(2)}$, $N_{ID}^{(1)}$ and the cell identity there are two synchronization signals that are constructed to find the identities. To determine the sector identity a signal called the Primary Synchronization Signal (PSS) is used and regarding the cell identity group a signal called the Secondary Synchronization Signal (SSS) is used. Figure 2.2 represents how the cell identities are implemented in practice [8].

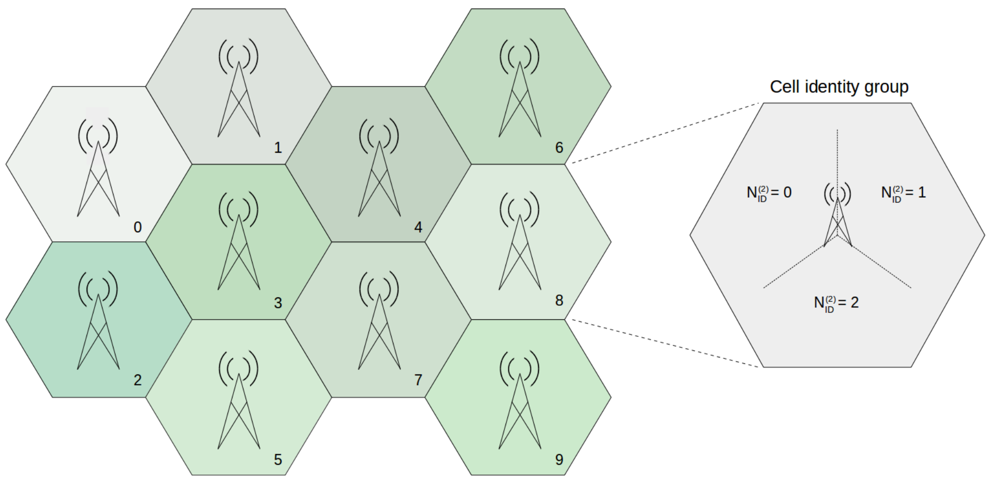


Figure 2.2: Demonstrating the placement of cell identity groups and how they are connected to the sector ID

2.2 Verizon 5G Terrestrial Fiber Standard

As explained in 1.1 V5GTF has created a standard for 5G. In this section all theory is retrieved from V5GTF's documentation "Verizon 5G TF; Air Interface Working Group; Verizon 5th Generation Radio Access; Physical channels and modulation (Release 1)" [11].

2.2.1 Synchronizations signals

Each radio frame is 10 ms and divided into 50 subframes, lasting 0.2 ms each. For each subframe there are 1200 subcarriers. In each subframe there are 14 OFDM symbols in time domain. In table 2.1 set parameters for the v5GTF standard are found.

Table 2.1: Parameters specified for the v5GTF standard

v5GTF parameters		
Number of samples	N	2048
Sample time	T_s	$6.5104 \cdot 10^{-9}$ s
Subcarrier spacing	Δf	75 kHz

Each symbol has N samples of data and cyclic prefix lengths of $N_{\text{CP}} = 160$ for OFDM symbols 0 and 8, and $N_{\text{CP}} = 144$ for the remaining 12 OFDM symbols. This is represented in figure 2.3. Each sample has a sample time T_s which can be calculated from the subcarrier spacing using

$$T_s = \frac{1}{N \cdot \Delta f}. \tag{2.8}$$

The OFDM symbols are roughly the time of a subframe divided by 14, but differs a bit since the cyclic prefix lengths are not fixed for all OFDM symbols.

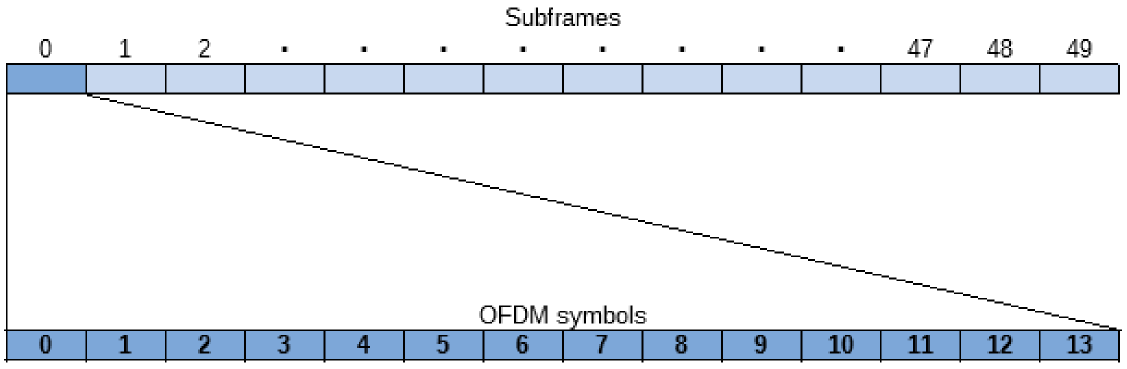


Figure 2.3: Relationship between subframes and OFDM symbols

The synchronization process in the 5GTF standard is divided into three parts; part one is the PSS covered in section 2.2.1.1 where $N_{\text{ID}}^{(2)}$ is determined, part two is determining $N_{\text{ID}}^{(1)}$ and subframe from the SSS covered in section 2.2.1.2, and part three is determining the OFDM symbol from the Extended Synchronization Signal (ESS) covered in section 2.2.1.3, with the knowledge of cell identity using

$$N_{\text{ID}}^{\text{cell}} = 3 \cdot N_{\text{ID}}^{(1)} + N_{\text{ID}}^{(2)} \tag{2.9}$$

gives the system essential information for future reception and transmission purposes. In figure 2.4 the synchronization signals and their position in both time (subframes) and frequency domain (subcarriers) is shown, where all three synchronization signals stretch over multiple subcarriers.

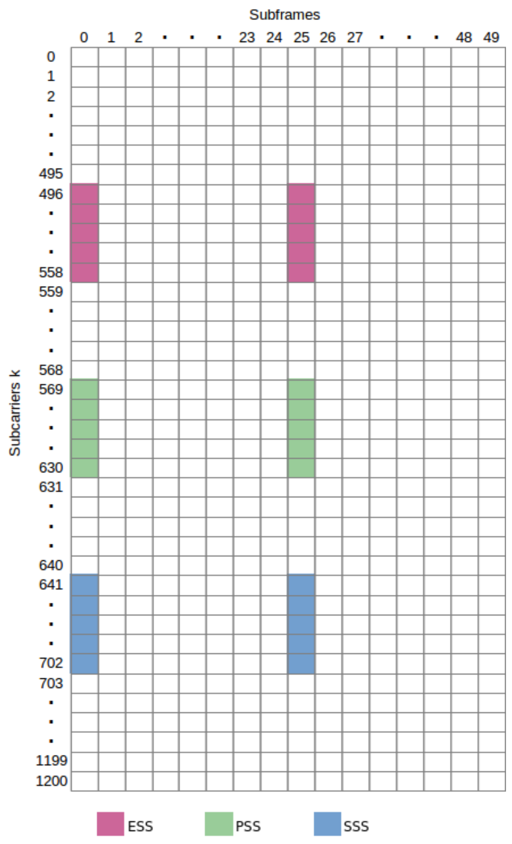


Figure 2.4: PSS, SSS and ESS are in subframes 0 and 25 but in different subcarriers

The data transmission is structured as follows; the base station broadcasts one radio frame at a time which contains all synchronization data required. No communication in uplink is needed for this. When decoding the cell identity a cross-correlation is performed between the received base station signal and signals representing the different cell identities to find the best match and thereby finding which ID the cell has as well as synchronization.

2.2.1.1 Primary Synchronization Signal

The primary synchronization is the first signal that the UE will use to decode what cell identity the base station has by finding the sector identity. The sent signal is built using a slightly modified Zadoff-Chu sequence

$$d_u(n) = \begin{cases} e^{-j\frac{\pi u n(n+1)}{63}} & n = 0, 1, 2, ..., 30 \\ e^{-j\frac{\pi u (n+1)(n+2)}{63}} & n = 31, 32, 33, ..., 61 \end{cases} \tag{2.10}$$

where the denominator of the exponent is 63 instead of the length of the sequence (62).
The benefit of using a Zadoff-Chu sequence is that the peak of the autocorrelation is very distinct and therefore it is useful when searching for a signal with noise.

Table 2.2: PSS root indexes

$N_{\text{ID}}^{(2)}$	u
0	25
1	29
2	34

$$n = \begin{cases} 0, 1, ..., 61 & \text{for downlink transmission} \\ -5, ..., -1, 62, ..., 66 & \text{for resource elements} \end{cases} \tag{2.11}$$

$$k = n - 31 + \frac{N_{\text{RB}}^{\text{DL}} N_{\text{sc}}^{\text{RB}}}{2} \tag{2.12}$$

$$l = 0, 1, ..., 13 \tag{2.13}$$

For the PSS there are three different reference signals each connected to a $N_{\text{ID}}^{(2)}$, the reference signals are then calculated according to equation 2.10 using the root indices as seen in table 2.2. Values $d_u(n)$ corresponding to n = 0,...,62 will later be placed in subcarriers k = 569,...,630 and duplicated in all 14 of the OFDM symbols in subframes 0 and 25. In this step $N_{\text{ID}}^{(2)}$ of the cell is detected and latter known in detection of $N_{\text{ID}}^{(1)}$.

2.2.1.2 Secondary Synchronization Signal

The secondary synchronization signal is placed in subframes 0 and 25, in subcarriers k = 641,...,703 and the OFDM symbols are duplicated in each subcarrier 14 times. It is thereby possible to detect if it is either subframe 0 or 25 from the SSS. The calculation of the SSS is more complex than the PSS by combining three different scrambling sequences in $d(n)$

$$d(2n) = \begin{cases} s_0^{(m_0)}(n) c_0(n) & \text{for subframe 0} \\ s_1^{(m_1)}(n) c_0(n) & \text{for subframe 25} \end{cases} \tag{2.14}$$

$$d(2n+1) = \begin{cases} s_1^{(m_1)}(n) c_1(n) z_1^{(m_0)}(n) & \text{for subframe 0} \\ s_0^{(m_0)}(n) c_1(n) z_1^{(m_1)}(n) & \text{for subframe 25} \end{cases} \tag{2.15}$$

where $s_0^{(m_0)}(n)$, $s_1^{(m_1)}(n)$, $c_0(n)$, $c_1(n)$, $z_1^{(m_0)}(n)$, $z_1^{(m_1)}(n)$ are scrambling sequences defined in subsections 2.2.1.2.1, 2.2.1.2.2 and 2.2.1.2.3. The scrambling sequences uses the same initial values but different constants in a scrambling equation, using modulus 2 and 31, to create three different scrambling patterns. In equations

$$m_0 = (m') \bmod 31 \tag{2.16}$$

$$m_1 = \left(m_0 + \left\lfloor \frac{m'}{31} \right\rfloor + 1\right) \bmod 31 \tag{2.17}$$

$$m' = N_{\text{ID}}^{(1)} + \frac{q(q+1)}{2}, \quad q = \left\lfloor \frac{N_{\text{ID}}^{(1)} + \frac{q'(q'+1)}{2}}{30} \right\rfloor, \quad q' = \left\lfloor \frac{N_{\text{ID}}^{(1)}}{30} \right\rfloor \tag{2.18}$$

m_0 and m_1 are dependent on $N_{\mathrm{ID}}^{(1)}$. This makes for 168 different SSS.

$$n = \begin{cases} 0, 1, ..., 61 & \text{for downlink transmission} \\ -5, ..., -1, 62, ..., 66 & \text{for resource elements} \end{cases} \tag{2.19}$$

$$k = n + 41 + \frac{N_{\mathrm{RB}}^{\mathrm{DL}} N_{\mathrm{sc}}^{\mathrm{RB}}}{2} \tag{2.20}$$

$$l = 0, 1, ..., 13 \tag{2.21}$$

2.2.1.2.1 Sequences s use the previously created m_0 and m_1, which are dependent on $N_{\mathrm{ID}}^{(1)}$, to create the sequences

$$s_0^{(m_0)}(n) = \tilde{s}\Big((n + m_0) \bmod 31\Big) \tag{2.22}$$

$$s_1^{(m_1)}(n) = \tilde{s}\Big((n + m_1) \bmod 31\Big) \tag{2.23}$$

using the function

$$\tilde{s}(i) = 1 - 2x(i), \text{ where } 0 \leq i \leq 30 \tag{2.24}$$

which is dependent on the initial sequence

$$x(i+5) = \Big(x(i+2) + x(i)\Big) \bmod 2 \qquad 0 \leq i \leq 25 \tag{2.25}$$

where $x(0) = 0,\ x(1) = 0,\ x(2) = 0,\ x(3) = 0$ and $x(4) = 1$.

2.2.1.2.2 Scrambling sequences c use $N_{ID}^{(2)}$ to create the sequences

$$c_0(n) = \tilde{c}\Big((n + N_{ID}^{(2)}) \bmod 31\Big) \tag{2.26}$$

$$c_1(n) = \tilde{c}\Big((n + N_{ID}^{(2)} + 3) \bmod 31\Big) \tag{2.27}$$

using the function

$$\tilde{c}(i) = 1 - 2x(i), \text{ where } 0 \leq i \leq 30 \tag{2.28}$$

which is dependent on the initial sequence

$$x(i+5) = \Big(x(i+3) + x(i)\Big) \bmod 2 \qquad 0 \leq i \leq 25 \tag{2.29}$$

where $x(0) = 0,\ x(1) = 0,\ x(2) = 0,\ x(3) = 0$ and $x(4) = 1$.

2.2.1.2.3 Scrambling sequences z use $N_{\mathrm{ID}}^{(2)}$ to create the sequences

$$z_0^{(m_0)}(n) = \tilde{z}\Big(\Big(n + (m_0 \bmod 8)\Big) \bmod 31\Big) \tag{2.30}$$

$$z_1^{m_1}(n) = \tilde{z}\Big(\Big(n + (m_1 \bmod 8)\Big) \bmod 31\Big) \tag{2.31}$$

but differs from scrambling sequence c (and s) in the creation of the x sequence

$$x(i+5) = \Big(x(i+4) + x(i+2) + x(i+1) + x(i)\Big) \bmod 2 \qquad 0 \le i \le 25 \tag{2.32}$$

where $x(0) = 0,\ x(1) = 0,\ x(2) = 0,\ x(3) = 0$ and $x(4) = 1$. Using the function

$$\tilde{z}(i) = 1 - 2x(i), \text{ where } 0 \le i \le 30. \tag{2.33}$$

2.2.1.3 Extended Synchronization Signal

The ESS is dependent on the cell ID detected in PSS and SSS, and varies over the 14 different OFDM symbols

$$d(n) = \mathrm{e}^{-j}\frac{25\pi n(n+1)}{63} \qquad n = 0, 1, ..., 62 \tag{2.34}$$

$$\tilde{d^l}(n) = d\Big((n + \Delta_l) \bmod 63\Big) \qquad n = 0, 1, ..., 62 \tag{2.35}$$

using the cyclic shift constants in table 2.3, making it possible to detect OFDM symbol ID. The initial sequence for $x_1(n)$ is already defined as

$$x_1(n+31) = \Big(x_1(n+3) + x1(n)\Big) \bmod 2 \tag{2.36}$$

where $N_c = 1600$ and sequence initialized with $x_1(0) = 1$, $x_1(n) = 0$, $n = 1, 2, ..., 30$ and $n = 0, 1, ..., M_{\text{PN}} - 1$, whilst for $x_2(n)$ it has to be retrieved from

$$c_{\text{init}} = 2^{10}(i+1)\Big(2N_{\text{ID}}^{\text{cell}} + 1\Big) + 2N_{\text{ID}}^{\text{cell}} + 1 \tag{2.37}$$

$$c_{\text{init}} = \sum_{i=0}^{30} x_2(i) \cdot 2^i \tag{2.38}$$

$$c(n) = \Big(x_1(n + N_c) + x_2(n + N_c)\Big) \bmod 2 \tag{2.39}$$

$$x_2(n+31) = \Big(x_2(n+3) + x_2(n+2) + x_2(n+1) + x_2(n)\Big) \bmod 2 \tag{2.40}$$

which are based on the cell ID and the subframe i. Functions 2.34 and 2.35 and the x sequences are linked using

$$r_i(n) = \frac{1}{\sqrt{2}}\Big(1 - 2c(2n)\Big) + j\frac{1}{\sqrt{2}}\Big(1 - 2c(2n+1)\Big) \qquad n = 0, 1, ..., 62 \tag{2.41}$$

$$d^l(n) = r_i(n)\tilde{d^l}(n) \qquad n = 0, 1, ..., 62 \tag{2.42}$$

$$n = \begin{cases} -0, 1, ..., 62 & \text{for downlink transmission} \\ -4, ..., -1, 64, ..., 67 & \text{for resource elements} \end{cases} \tag{2.43}$$

$$k = n - 104 + \frac{N_{\text{RB}}^{\text{DL}} N_{\text{sc}}^{\text{RB}}}{2} \tag{2.44}$$

$$l = 0, 1, ..., 13. \tag{2.45}$$

Table 2.3: Time domain indices l mapped to its cyclic shift Δ_l

l	Δ_l
0	0
1	7
2	14
3	18
4	21
5	25
6	32
7	34
8	38
9	41
10	45
11	52
12	59
13	61

2.2.2 OFDM signal generation in baseband

The baseband signal generation is defined as follows

$$s_l(t) = \sum_{k=-\lfloor N_{\mathrm{RB}}^{\mathrm{DL}} N_{\mathrm{sc}}^{\mathrm{RB}}/2 \rfloor}^{-1} a_{k^{(-)},l} e^{j2\pi k \Delta f (t - N_{\mathrm{CP,l}} T_s)} + \sum_{k=1}^{\lceil N_{\mathrm{RB}}^{\mathrm{DL}} N_{\mathrm{sc}}^{\mathrm{RB}}/2 \rceil} a_{k^{(+)},l} e^{j2\pi k \Delta f (t - N_{\mathrm{CP,l}} T_s)} \tag{2.46}$$

where the time lies in the interval $0 \leq t \leq T_s(N + N_{CP})$, a cyclic prefix is considered in the signal generation and where $k^{(-)} = k + \left\lfloor N_{RB}^{DL} N_{sc}^{RB}/2 \right\rfloor$ and $k^{(+)} = k + \left\lfloor N_{RB}^{DL} N_{sc}^{RB}/2 \right\rfloor - 1$ represents which subcarriers the different symbols are generated to.
In equations

$$a_{k^{(-)},l} = \sum_{t=0}^{T_s(N-1)} s(t) e^{-j2\pi k \Delta f t} \text{ for } k = -\left\lfloor N_{RB}^{DL} N_{sc}^{RB}/2 \right\rfloor, ..., -1 \tag{2.47}$$

$$a_{k^{(+)},l} = \sum_{t=0}^{T_s(N-1)} s(t) e^{-j2\pi k \Delta f t} \text{ for } k = 1, ..., \left\lfloor N_{RB}^{DL} N_{sc}^{RB}/2 \right\rfloor \tag{2.48}$$

the received signal $s(t)$ is modified from N number of samples in time domain to 1200 subcarriers in frequency domain.
When referring back to the reference signals the following equivalences upholds: PSS and SSS $a_{k,l} = d(n)$ and for ESS $a_{k,l} = d^l(n)$ for $k = 0, ..., 1199$, see equations 2.10, 2.14, 2.15 and 2.10. Where n and k are linearly connected according to equations 2.12, 2.20 and 2.44.

3
Compute Unified Device Architecture

In the beginning of this chapter specifications about the systems used in this project will be presented. This will be followed by a brief explanation about common CUDA concepts and the chapter is completed with a discussion about the important trade-off between time efficiency and latency while using CUDA.

3.1 Information about PC

The CPU, on which all code is executed, that the GPU is connected to is a Taurus, system model MS-7A59, with the following system information as listed in table 3.1.

Table 3.1: System Information about used PC

System Information	
Processor	Intel(R) Core(TM) i7-7700K CPU
Processor Speed	4.2 GHz
Number of Cores	4
Number of Logical Processors	8
RAM Memory	16 GB
Operating System	Windows 10

3.2 Information about CUDA

In contrast to the traditional type of programming where the code is executed in a serial sequence in the CPU, the GPU uses many smaller processing units, called cores, to divide the workload. This makes it possible for the GPU to reach much higher calculation speeds than the CPU, but only under the right circumstances. Since the GPU is built out of many small processing units it makes it quite inflexible in what to calculate which creates a trade-off between the CPU which can perform a much more complex flow of calculations but can lack in speed whilst the GPU is faster as long as it gets a big set of data to perform more uniform calculations on [12]. It is thereby of high importance which calculations to implement for the parallel program. The GPU platform used in this project is CUDA created by nVIDIA, model Titan Xp [13]. The graphics card has the following specifications as in table 3.2 [14].

Table 3.2: System Information about CUDA graphics card

System Information		
nVIDIA Cores	n_{cores}	3840
Memory Speed	v_{mem}	11.4Gbps
Number of Multiprocessors	n_{MP}	30
Total Available Graphics Memory	M_{GPU}	20441MB
Memory Bandwidth	BW_{mem}	547.7GB/s
Maximum blocks per Multiprocessors (MP)	$B_{\text{max,MP}}$	32
Maximum threads per block	$T_{\text{max,block}}$	1024
Minimum threads per block	$T_{\text{min,block}}$	32
Number of threads per CUDA core	T_{core}	16
Warp size	T_{warp}	32
Overload compatible	-	Yes

3.2.1 Blocks and Threads

When CUDA runs there is an option of how many threads to run at the same time and how many times. An execution of the kernel code is called a thread [15]. In this case the maximum number of threads that can run simultaneously on a so called block is 1024 as mentioned in table 3.2. If more than 1024 threads need to be executed, there is an option of running the threads in blocks, see figure 3.1. Blocks are collections of threads that should be executed in parallel, putting a number of threads in a block ensures that they run at the same time whilst putting threads in different blocks does not [15]. Which block the threads are in also affects the memory access. All threads can access the `__global__` memory, which is also the slowest memory to access. The `__shared__` memory on the other hand can only be accessed by the threads within the block, this memory access is faster than the `__global__` one. The fastest memory access is the local memory which can only be accessed by the thread it is declared in.

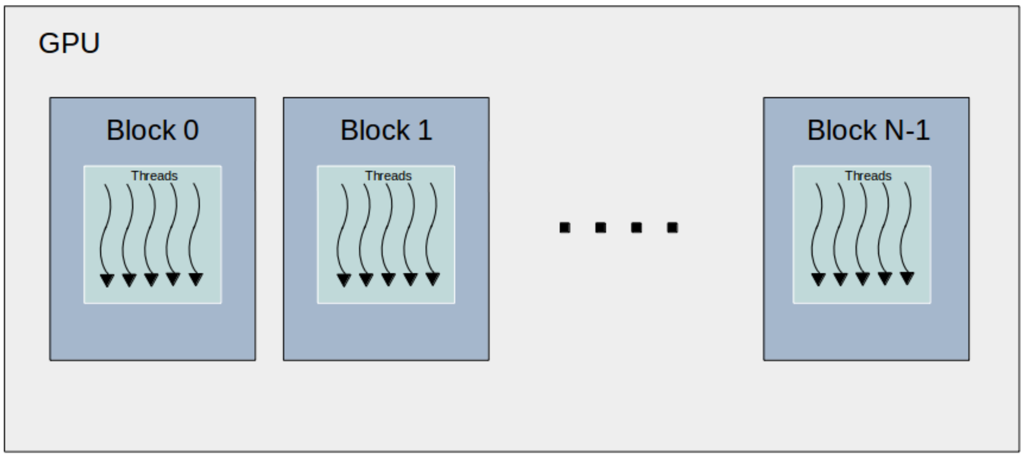


Figure 3.1: Visualization of blocks and threads

When designing the system in a time efficient way it is important to keep track of how many blocks can be executed at the same time. This is possible using the parameters as referenced in table 3.2. To find how many blocks are possible to run at the same time first the maximum amount of threads per MP ($T_{\text{max,MP}}$) has to be calculated as follows

$$T_{\text{max,MP}} = \frac{n_{\text{cores}}}{n_{\text{MP}}} \cdot T_{\text{core}} = \frac{3840}{30} \cdot 16 = 2048 \text{ threads} \tag{3.1}$$

and with the use of $T_{\text{max,MP}}$ the maximum amount of blocks that can run simultaneously with maximum thread count on the blocks (B_{max}) is calculated by

$$B_{\text{max}} = \frac{T_{\text{max,MP}} \cdot n_{\text{MP}}}{T_{\text{max,block}}} = \frac{2048 \cdot 30}{1024} = 60 \text{ blocks} \tag{3.2}$$

meaning that 60 blocks of size 1024 threads can be run at the same time. The maximum amount of running blocks with the minimum thread count per block (B_{min}), note T_{warp} = minimum thread count / block, can be calculated as follows

$$B_{\text{min}} = B_{\text{max,MP}} \cdot n_{\text{MP}} = 32 \cdot 30 = 960 \text{ blocks} \tag{3.3}$$

showing that it is possible for the GPU to run 960 blocks of size 32 threads at the same time.

3.2.1.1 Warps

All threads are executed in groups called warps and an entire warp follows the same instructions [16]. This means that when a thread encounters an `if/else`-statement and only some of the threads within the warp go down the `if`, the threads within that warp that did not fit the requirement will wait for the threads in the statement. The fact that some threads have to wait for the other threads is time consuming and has to be considered to the greatest extent possible when programming the kernel.

The warp size, T_{warp}, for Titan Xp can be found in table 3.2 and is 32 threads per warp. It is also important to have the correct amounts of threads per block, so that all the threads in the warps are used, for the efficiency of the system. The number of warps per block can be calculated by

$$\text{warps/block} = \left\lceil \frac{\text{threads/block}}{T_{\text{warp}}} \right\rceil \tag{3.4}$$

and if the number of threads per block is not divisible with the warp size the GPU will still activate some threads to fill out the warp [16]. Since this means that there are threads activated that have no instructions this is less efficent than when all threads in a warp have instructions.

3.2.2 `cudaMemcpy` and `cudaMalloc`

When executing CUDA code it is fundamental to first allocate memory using the command `cudaMalloc`, after the memory allocation it is also necessary to copy over all of the data using the command `cudaMemcpy` since the CPU and the GPU do not share memory. First when both `cudaMemcpy` and `cudaMalloc` have been executed can the kernel be successfully called on. When the kernel has run there is only the `cudaMemcpy` back the data to the CPU memory. This process of allocating memory and transferring data is time consuming and can take up to 50 times the kernel timing making it an important aspect while optimizing CUDA code [15],[17].

3.2.3 CUDA streams

When using a GPU to accelerate an application or system there is often either a large chunk of data to be processed or the CUDA cores are not used to its full potential. One way to solve these problems is to use CUDA streams. By using streams one can force the MP to always keep busy. To be able to use CUDA streams the graphics card must be able to handle overload. As it can be seen in table 3.2 the GPU used in this thesis is able to handle overload. The point of streams is to make the GPU work with small parts of the data simultaneously instead of handle all of the data in serial. If two streams are used, let them be called stream 0 and stream 1, when stream 0 has used `cudaMemcpy` on a part of the data, that is supposed to be copied onto the graphics card, and start to launch the kernel with that part of the data stream 1 can start copying the next part of the data and when launching the kernel for that part of the data stream 0 copies the next part and so forth. This way the application, if streams are used correctly, can have a significant performance enhancement [18].

3.2.4 `cudaHostAlloc` and `cudaMemcpyAsync`

To be able to run CUDA Streams a new set of allocation on the CPU and data copying tools needs to be utilized. The `cudaHostAlloc` command is used instead of the most commonly known `malloc`, the difference between these two allocation commands is that the `cudaHostAlloc` allocates page-locked memory on the CPU.

The reason this memory allocation method is used is both since it can perform faster copies and that the `cudaMemcpyAsync` requires it. The difference between the `cudaMemcpy` and the `cudaMemcpyAsync` is synchronous respectively asynchronous meaning that the synchronous function `cudaMemcpy` will return only when all of the data transfer is completed, whilst the asynchronous `cudaMemcpyAsync` will return as soon as it has been called, with the guarantee that it will be finished at the start of the next request within the same stream [18].

3.2.5 Reduce

A reduce function is a function where an operation of many elements can get better efficiency by being computed in parallel using a GPU. To be able to perform this algorithm the number of input values must be a power of two. After the threads are initialized the GPU starts by comparing the first value of the first half of the vector with the first value of the second half of the vector. There are different variations of a reduce function, such as additions and finding the minimum value. In the reduce function used in this thesis, the function takes the maximum value and stores the result in index 1. All running threads performs one of these calculations each until all values has been counted for once each. After this there are only half the number of threads left and the procedure starts over again until there is only one value left, which is the maximum of the entire counted for vector. This procedure is presented in figure 3.2.

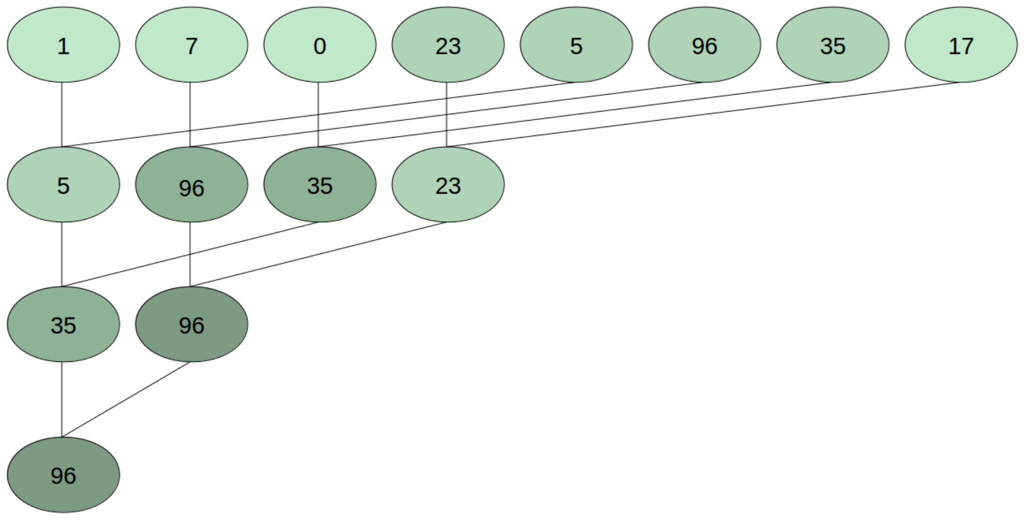


Figure 3.2: Visualization of max reduce function

3.3 Throughput and latency

The code can be optimized regarding to the two important criteria latency and throughput. Latency, also known as the response time, is important since e.g., in time-critical applications the UE needs to be connected as fast as possible. Low latency can be achieved by analyzing smaller packages of data at a time making

the data accumulation period as well as the service time shorter [19]. Throughput differs from the latency in the sense of how many requests are met per time unit and disregards how long or short the latency for each request is [19],[20].
When it comes to CUDA, when a stream of data is received, the average latency and average throughput is determined by the length of the data vector that is analyzed in each iteration. If the vector is short it would get the first data set through the system very fast. However the issue here with CUDA is that there is both a start up and a shut down time when transferring the data over from the CPU to the GPU. This extra time accounts for a considerable percentage of the execution time. For the long vector the situation would be opposite, the extra time spent by the data transfer would be smaller in proportion to the amount of data processed but the latency has to be larger since the system needs to accumulate the data for a longer period of time.

4

Implementation

Since the main part of this thesis is practical, this chapter will explain and present the implementation of the thesis project. The majority of this chapter will cover how the system was programmed in C, both for serial decoding, using only a CPU and for parallel decoding, using the GPU as well.

4.1 Reference Data

To analyze incoming data a set of reference data has to be created. This data is created before any timing processes and then stored in the PC's memory to save time, no reference data should be created during the data receiving process. The reference data for PSS is created using the equations in section 2.2.1.1 and is shown in figure 4.1, where three different sequences are created with a data length of 62. The reference data for SSS is created using the equations in section 2.2.1.2 and is shown in figure 4.2, where 504 sequences, also at data length 62, are created and sorted into which $N_{\mathrm{ID}}^{(2)}$ it is based on so it is easy to match which 168 sequences to perform the cross correlation based on what $N_{\mathrm{ID}}^{(2)}$ is found in the PSS. The ESS is created according to section 2.2.1.3, shown in figure 4.3, and has $504 \cdot 14$ sequences with a data length of 63 symbols each. All of the symbols in the reference data lies at a distance of $\frac{1}{\sqrt{2}}$ from the origin.

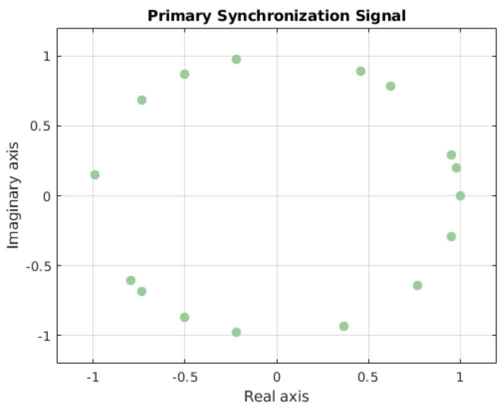


Figure 4.1: The PSS reference data in a constellation plot

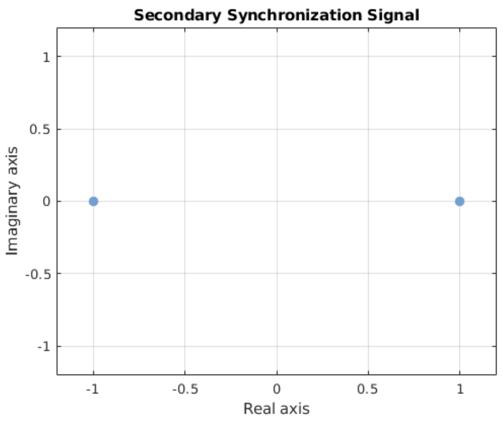


Figure 4.2: The SSS reference data in a constellation plot

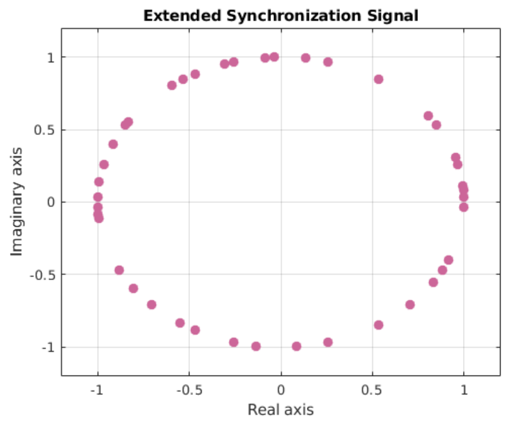


Figure 4.3: The ESS reference data in a constellation plot

When finding the correct sector ID the received synchronization signals are cross correlated with the reference signals in correct order. In figure 4.4 the scenario of finding the correct $N_{\text{ID}}^{(2)}$ is found and in figure 4.5 when a match is not found. The distinct spike in figure 4.4 makes it simple to derive what $N_{\text{ID}}^{(2)}$ is correct.

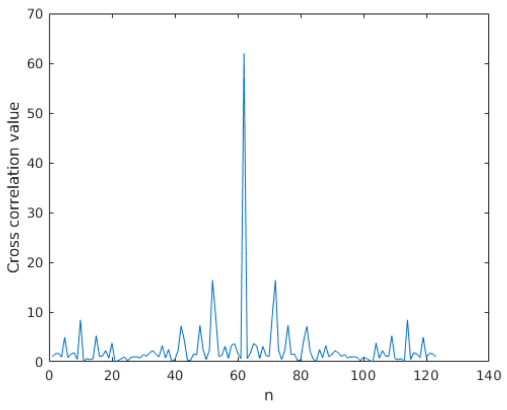


Figure 4.4: Cross correlation between sector ID = 1 with itself

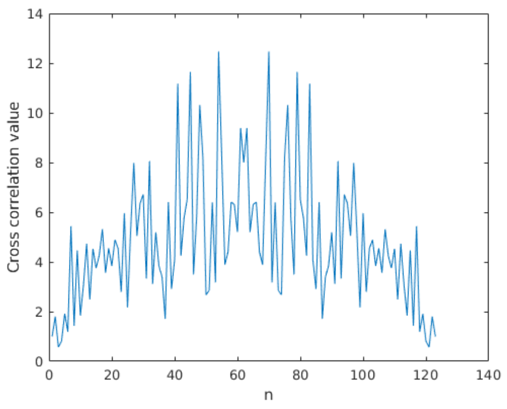


Figure 4.5: Cross correlation between sector ID = 1 and sector ID = 2

4.2 Graphics Processing Unit

In this section the implementations of the parallel decoder, for the three synchronization signals, will be explained in detail. The parallel decoder will be adapted for the GPU.

4.2.1 Memory Allocation and Data Transfer

As mentioned in section 3.2.2 before stepping into the kernel programs an important aspect is the time efficiency of the two way data transferring between CPU and GPU. This was done by allocating all of the memory on the GPU and transfer the reference data to the GPU before starting the program, letting it be statically saved on the GPU.

4.2.2 Time to Frequency Domain (FFT)

The equations 2.47 and 2.48 can be implemented in a more efficient way than parallelizing the equations by a FFT as defined in section 2.1.2 using the following equivalences

$$\begin{aligned} a_{k,l} &= \sum_{t=0}^{T_s(N-1)} s(t) \cdot \mathrm{e}^{-j2\pi k\Delta f t} = \sum_{t=0}^{N-1} s(t) \cdot \mathrm{e}^{-j2\pi k\Delta f T_s t} = \\ &= \sum_{t=0}^{N-1} s(t) \cdot \mathrm{e}^{-j2\pi k\Delta f \frac{1}{\Delta f N} t} = \sum_{t=0}^{N-1} s(t) \cdot \mathrm{e}^{\frac{-j2\pi k t}{N}} = \mathrm{FFT}\big(s(t)\big). \end{aligned} \tag{4.1}$$

These equivalencies shows how the degeneration of the baseband signal can be done by using a FFT as defined in equation 2.3.
For doing FFTs on CUDA there is already a pre-installed toolkit to use which performs FFTs in high speed. The values of $k = 1, ..., 600$ can be easily accessed in indexes $1, ..., 600$ in the FFT output whilst $k = -600, ..., -1$ can be retrieved from indexes $1448, ..., 2047$.

4.2.3 Cross correlation

As discussed in section 2.2.1 a cross correlation is performed in order to find what cell ID the base station, that the UE is trying to connect to, has. The cross correlations has been implemented separately for primary, secondary and extended signals for efficiency reasons. When performing the cross correlation the received signal has been padded with zeros on both sides. The number of zeros on one side is the size of the signal minus one to fit the cross correlation and is alike for PSS, SSS and ESS. The signal has length 123 after the padding. After the cross correlation for all three synchronization signals the kernel is completed with the maximum reduce-algorithm, see section 3.2.5, to find the N_{ID} in question.
The code snippet presented in listing 4.1 represents the cross correlation for the SSS with only real values. Since the signal in general is complex, calculating the cross correlation for the PSS and ESS is slightly more involved. All of these deviations are accounted for in the complexity calculations for the cross correlations.
Since one signal is reached by the memory for each reference signal this makes the memory access an significant part of the kernel timing. When the signal is `memcpy` into the GPU it automatically ends up in the global memory. Here there are two more considerable alternatives that could help optimize the system. The first alternative is to transfer the memory to the shared memory in the GPU, the other alternative is to at the start of the CPU program initialize the signal as the type `__constant__` and then copy the signal to the GPU.

Listing 4.1: Cross correlation

```
__global__ void xCorrSecKernel(int *refSig, int *sig, int *xcorr) {
    int i = threadIdx.x + blockDim.x * blockIdx.x; // 0 -> Cell identity group #
    int j = threadIdx.y; // Signal offset
    int signalSize; // Size of original signal
    int thread = j + threadIdx.x * blockDim.y; // Thread index in block

    double corrValues = 0;

    if (thread < 3 * signalSize - 2)
        sigShared[thread] = sig[thread];
    __syncthreads();

    // Cross correlation
    for (int m = 0; m < signalSize; m++) {
        corrValues = corrValues + refSig[m + i * signalSize] * sigShared[m + j];
    }
    xcorrShared[thread] = sqrtf(corrValues * corrValues);
}
```

4.2.3.1 Reduce

The reduce part of the cross correlation was implemented as shown in listing 4.2. The system performs add reduce in the same way as the max reduce is described in figure 3.2 but when there are only 32 threads left (one warp) 32 threads will perform 6 calculations each making the final value ending up in index 0. This final step makes the code more time efficient since not all threads are running in the final step [21]. To be able to perform the most efficient type of reduce function the number of elements in the investigated vector needs to be a power of 2.

Listing 4.2: Max reduce function

```
for (unsigned int s = blockDim.x / 2; s > 32; s >>= 1) {
    if (threadIdx.x < s) {
                if (xCorr[threadIdx.x] < xCorr[threadIdx.x + s]) {
                xCorr[threadIdx.x] = xCorr[threadIdx.x + s];
            }
    }
    __syncthreads();
}
if (threadIdx.x < 32) {
        if (xCorr[threadIdx.x] < xCorr[threadIdx.x + 32]) {
            xCorr[threadIdx.x] = xCorr[threadIdx.x + 32];
        }
        if (xCorr[threadIdx.x] < xCorr[threadIdx.x + 16]) {
            xCorr[threadIdx.x] = xCorr[threadIdx.x + 16];
        }
        if (xCorr[threadIdx.x] < xCorr[threadIdx.x + 8]) {
            xCorr[threadIdx.x] = xCorr[threadIdx.x + 8];
        }
        if (xCorr[threadIdx.x] < xCorr[threadIdx.x + 4]) {
            xCorr[threadIdx.x] = xCorr[threadIdx.x + 4];
        }
        if (xCorr[threadIdx.x] < xCorr[threadIdx.x + 2]) {
            xCorr[threadIdx.x] = xCorr[threadIdx.x + 2];
        }
        if (xCorr[threadIdx.x] < xCorr[threadIdx.x + 1]) {
            xCorr[threadIdx.x] = xCorr[threadIdx.x + 1];
        }
}
if (threadIdx.x == 0) {
    maxCrossCorrelation[blockIdx.x] = xCorr[0];
}
```

4.2.3.2 Cross correlation for the primary synchronization signal

When the kernel is started in one block where $3 \cdot 123 = 369$ active threads are launched in two dimensions of threads. One dimension is three threads (0 to 2) which are used as $N_{\text{ID}}^{(2)} = 0$, $N_{\text{ID}}^{(2)} = 1$ and $N_{\text{ID}}^{(2)} = 2$. The second dimension represents 123 threads (different fittings of the reference signal over the recorded signal) in y-led which is also the index for the padded signal. Since there are real and imaginary parts of the signal the correlation is calculated using the rule of multiplication of complex numbers.
The cross correlation takes up 369 threads but since the reduce is done in the same kernel as the cross correlation the number of active threads in the block has to be equal to any power of two less or equal to 1024. The closest higher number is 512. The cross correlations is thereby performed in one block of size 512 threads. As calculated in section 3.2 the GPU can hold 60 blocks of the thread size 1024 simultaneously, meaning that one run of the PSS cross correlation takes up 0.83% of the total available resources.

4.2.3.3 Cross correlation for the secondary synchronization signal

The cross correlation for the SSS does not differ much more from the PSS in more ways than the dimensions, since in the SSS the system is trying to figure out which one of 168 cell group identities the signal has as well as that the SSS only has real values in the complex plane. To be able to perform the cross correlation 168 different cell identity groups and 123 cases per cross correlation are examined in one block, this makes for 20664 threads. The 123 cases has to be in the same block, but the 168 cell identity groups do not require this. The third requirement is that the block size is exactly a power of two. This leads to the constraints

$$\begin{cases} \text{Block Size} \leq T_{\text{max,block}} & \\ \text{Block Size} \geq 123 \cdot i & \text{where i = Cell Identity Groups / Block} \\ \text{Block Size} = 2^j & \text{where j = 0,1,2,...} \end{cases} \tag{4.2}$$

which has to be satisfied whilst one priority is to get as few blocks as possible to minimize the reduce outside of the GPU. The amount of blocks for the maximum amount of cell identity groups per block is

$$i_{\text{max,block}} = \left\lceil \frac{\text{threads}}{\text{threads/block}} \right\rceil = \left\lceil \frac{20664}{1024} \right\rceil = 21 \text{ blocks} \tag{4.3}$$

and the cross correlation thereby takes up 21 blocks of size 1024 threads. This makes one run of the SSS take up 35% of the GPU's capacity.
Since the block size is 21 and there is no synchronization function for the blocks this means that to be able to properly compare the 21 values, as well as their corresponding $N_{\texttt{ID}}^{(1)}$, they have to be outputted from the GPU to the CPU and then later be reduced.

4.2.3.4 Cross correlation for the extended synchronization signal

The code for the cross correlation of the ESS is very similar to the one of the PSS except for the fact that the signal is longer for the ESS than both the PSS and SSS. When it comes to the dimensioning of the blocks the cross correlation needs $14 \cdot 125 = 1750$ threads, which makes for

$$i_{\text{max,block}} = \left\lceil \frac{1750}{1024} \right\rceil = 2 \text{ blocks} \tag{4.4}$$

causing the GPU to run 2 blocks of 1024 threads each. This procedure uses 3.33% of the GPU's total capacity.

4.2.4 CUDA streams

As mentioned in section 3.2.3 multiple kernels can be executed concurrent using CUDA streams. To implement this, some consideration into the hardware and software is made. There are three program kernels that needs to be launched subsequently, the cross correlations for PSS, SSS and ESS. Because of this CUDA streams cannot be used to calculate all the identities simultaneously. However the 14 OFDM symbols within the subframe can be executed concurrently since there is no restraint with the OFDM symbols using parameters in order. The kernel executing the most amounts of threads is the cross correlation for SSS and one execution of that kernel uses 35% of the GPU's total amount of available resources, see section 4.2.3.3. Hence only two streams can run concurrently, since $3 \cdot 35\% > 100\%$ and $2 \cdot 35\% < 100\%$.

4.2.5 Complexity

When dealing with the complexity of the system a number of rules to be able to compare is set up. In a `for`-loop the number of actions taken are first setting the initialized variable, then it will check if the variable has reached the maximum once every iteration, making $n + 1$ iteration and finally it will have to increase the variable once every iteration giving n action. So for one `for`-loop of length n, it will require $1 + (n + 1) + n$ actions. All memory relocation and all additions, subtractions, divisions and so on will also count as one action each. Thoroughly note that this is only a rough approximation of the complexity made by the authors. The complexity is estimated by making the size of the synchronization signals become

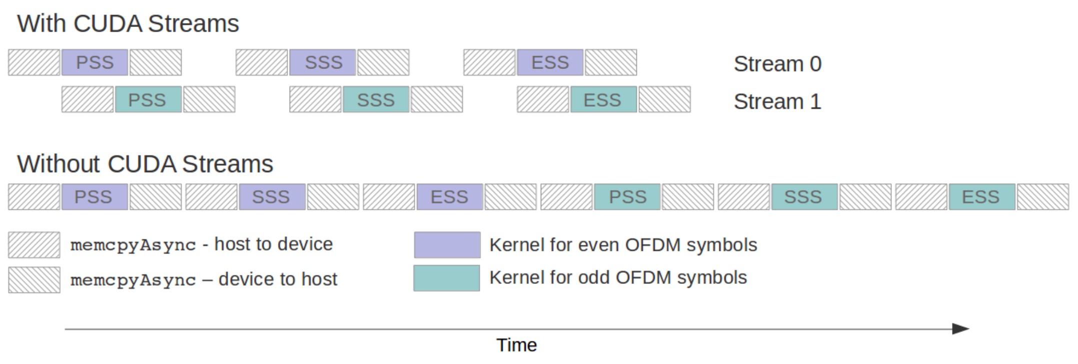

With CUDA Streams
PSS
SSS
ESS
Stream 0
PSS
SSS
ESS
Stream 1
Without CUDA Streams
PSS
SSS
ESS
PSS
SSS
ESS
memcpyAsync - host to device
memcpyAsync – device to host
Kernel for even OFDM symbols
Kernel for odd OFDM symbols
Time

Figure 4.6: Visualization of time consumption for the system with and without CUDA streams

large and then calculate how many times each statement of the program is read by the computer. This means that the complexity calculated in this section is an estimation the amount of work the program or function carries out.
In table 4.1 the parameters a and b can be found. These parameters are used to simplify the complexity expressions and are the values used in the cross correlation that differ for PSS, SSS and ESS. The a for PSS represent the number of sector ID reference signals to cross correlate the signal with. For SSS a is the number of cell identity group reference signals the signal is cross correlated with and for ESS it is the number of reference OFDM symbols to cross correlate the signal with. As for b, it is set to 2 or 1, depending whether the signal is complex or real.

Table 4.1: Parameters a and b for PSS, SSS and ESS

	PSS	SSS	ESS
a	3	168	14
b	2	1	2

4.2.5.1 Cross correlation

The complexity for the three synchronization signals on thread-level is presented in table 4.2. The variables a and b can be found in table 4.1. Line 10 in table 4.2 restricts the complexity to the threads that the `if`-statement is true for. On line 9 and 11 the complexity equation $= 1$ and in this case it basically means that all threads execute this statement once and then moves on to next statement. $2b^2$ in line 15 originates from the fact that multiplying two real vectors requires less calculation steps than multiplying two complex vectors. The complexity equations for the parallel cross correlation functions can be simplified to

$$n \cdot (4 + 2b^2) + \frac{3bn - 2b}{a \cdot (2n - 1)} + 3b + 2 \tag{4.5}$$

and when

$$\begin{aligned}
\lim_{n\to\infty}\left(n\cdot(4+2b^2)+\frac{3bn-2b}{a\cdot(2n-1)}+3b+2\right) = \\
= (4+2b^2)\lim_{n\to\infty} n + 3b\lim_{n\to\infty}\frac{n}{a(2n-1)} - 2b\lim_{n\to\infty}\frac{1}{a(2n-1)} + 3b + 2 = \\
= (4+2b^2)\lim_{n\to\infty} n + 3b\cdot\frac{1}{2a} - 0 + 3b + 2 \qquad (4.6)
\end{aligned}$$

the order of complexity is $\mathcal{O}(n)$ for one thread. For all active threads the complexity becomes

$$\begin{aligned}
\lim_{n\to\infty}\left(\left(n\cdot(4+2b^2)+\frac{3bn-2b}{a\cdot(2n-1)}+3b+2\right)\cdot n\right) = \\
= (4+2b^2)\lim_{n\to\infty} n^2 + 3b\lim_{n\to\infty}\frac{n^2}{a(2n-1)} - 2b\lim_{n\to\infty}\frac{n}{a(2n-1)} + (3b+2)\lim_{n\to\infty} n = \\
= (4+2b^2)\lim_{n\to\infty} n^2 + 3b\cdot\frac{1}{2a}\lim_{n\to\infty} n - 2b\cdot\frac{1}{2a} + (3b+2)\lim_{n\to\infty} n = \\
= (4+2b^2)\lim_{n\to\infty} n^2 + \left(\frac{3b}{2a}+3b+2\right)\lim_{n\to\infty} n - \frac{b}{a} \qquad (4.7)
\end{aligned}$$

which leads to an order of complexity of $\mathcal{O}(n^2)$ for all threads joint.

Table 4.2: Line by line complexity for cross correlation for parallel implementation

Line	
7	b
9	1
10	$b\cdot\frac{3n-2}{a\cdot(2n-1)}$
11	1
14	$1+2n+(2n-1)$
15	$2b^2n$
17	$2b$

4.2.5.2 Reduce

Presented in Table 4.3 is the line by line complexity of the max reduce in parallel with the vector that the reduce function is performed on is of size n. Note that the code presented in section 4.2.3.1 does not completely match the real code, but does just represent the concept of a maximum reduce function. All diversities will be stated or explained.
For line 4 there are two memory transfers, one to keep track of the cross correlation value and one to keep track of the property corresponding to the correlation, meaning the $N_{\text{ID}}^{(2)}$, $N_{\text{ID}}^{(1)}$ or OFDM symbol. This means that the real code uses two actions. This is also the case for line 10-27, but here an additional `__syncthreads()` is added, making three actions in each one of the `if`-statement.

Note that for the reduce function to work n has to be a power of two. The number of threads reaching inside the `if`-statement on line 2 for all total loops equals

$$\sum_{i=0}^{\log_2(n)-1} 2^i - \sum_{i=0}^{\log_2(32)} 2^i = n - 1 - 63 = n - 64 \tag{4.8}$$

representing a probability $\frac{n-64}{n}$ that the program will jump into to loop for each iteration. The loop on line 4 is conditional to the value of the cross correlation, an approximation that the system reaches inside the `if`-statement with a 0.5 probability is thereby made. Presented in table 4.3 is the complexity of the max reduce function row-by-row for one thread. Adding all lines together becomes the expression

$$\begin{aligned} 4 \cdot \log_2(n) + 2 \cdot \frac{n-64}{n} + 4 + \frac{511}{n} &= \\ = 4 \cdot \log_2(n) + 6 + \frac{383}{n}&. \end{aligned} \tag{4.9}$$

As n goes to infinity

$$\begin{gathered} \lim_{n\to\infty} \left(4 \cdot \log_2(n) + 6 + \frac{383}{n}\right) = \\ 4 \lim_{n\to\infty} (\log_2(n)) + 6 + 0 \end{gathered} \tag{4.10}$$

the order of complexity is $\mathcal{O}(\log_2(n))$ for one thread. The complexity for all threads is on the other hand

$$\begin{gathered} \lim_{n\to\infty} \left(\left(4 \cdot \log_2(n) + 6 + \frac{383}{n}\right) \cdot n\right) = \\ 4 \lim_{n\to\infty} (n \cdot \log_2(n)) + 6 \lim_{n\to\infty} n + 383 \end{gathered} \tag{4.11}$$

giving the order of the total complexity $\mathcal{O}(n \cdot \log_2(n))$ for all joint threads.

Table 4.3: Line by line complexity equations for reduce algorithm

Line	
1	$1 + (\log_2(n) + 1) + \log_2(n) = 2 + 2 \cdot \log_2(n)$
2	$\log_2(n) \cdot 1$
3	$\frac{n-64}{n}$
4	$\frac{n-64}{n} \cdot 0.5 \cdot 2 = \frac{n-64}{n}$
7	$\log_2(n) \cdot 1$
9	1
10	$\frac{32}{n} \cdot 1$
11	$\frac{32}{n} \cdot 0.5 \cdot 3 = \frac{48}{n}$
13-27	$5 \cdot (\frac{32}{n} + \frac{48}{n}) = \frac{430}{n}$
29	1
30	$\frac{1}{n}$

4.3 Central Processing Unit

In this section the implementation for the serial decoding, for PSS, SSS and ESS is thoroughly explained. The code for the serial decoder is adapted to the CPU.

4.3.1 Time to Frequency Domain (FFT)

The FFT of the serial code was implemented using programs from FFTW which is a free library software which can be found online [22]. The programs of this library are much faster than any code that could have been implemented by the authors of this report during the time frame of this project.

4.3.2 Cross correlation

The cross correlation functions for the serial decoding are built very similarly to the kernels in the parallel decoding. There are two separate functions for the cross correlation of PSS and SSS here as well. The difference between the serial decoding program and parallel decoding program is that, since nothing is implemented on the GPU, there is no need to take consideration for threads and blocks and therefore is simpler to implement. In section 4.2.3 the padding for the signal was explained and the same method is used for the signal in serial decoding.
The implementations of the cross correlation for PSS, SSS and ESS can be seen in listing 4.3. The implementation difference in the correlations is that the function for the SSS does not handle imaginary parts and for the ESS correlation the sizes of the loops is longer due that ESS is 63 symbols long and the two other are 62 symbols long.

Listing 4.3: Cross correlation

```
int signalSize; // Size of original signal
int X; // # of reference signals

for (int nid = 0; nid < X; nid++){
    for(int n = 0; n < 2 * signalSize - 1; n++){
        corrValues = 0;
        for (int m = 0; m < signalSize; m++){
            corrValues = corrValues + refSig[nid * signalSize + m] * sigPad[m + n];
        }
        xCorr[nid * (2 * n - 1) + n] = sqrt(corrValues * corrValues);
    }
}
```

4.3.2.1 Finding the maximum cross correlation

To find the maximum cross correlation there is a value by value search that is the standard way of finding the maximum in an array which was implemented. There are other algorithms that can optimize time but none that can give the absolute max, making them unsuitable for this purpose [23].

Listing 4.4: Find maximum cross correlation value for serial decoder

```
max = xCorr[0][0];
for (int i = 0; i < NID2Size; i++) {
    for (int j = 0; j < xCorrSize; j++) {
        if (max < xCorr[i][j]) {
            max = xCorr[i][j];
            NID2 = i;
        }
    }
}
```

4.3.2.2 Alternative implementation possibilities

Except for the serial implementations presented in section 4.3 there are other implementation possibilities that might make the system more time efficient which are presented in this section.

4.3.2.2.1 Alternative 1 One alternative way of implementing the cross correlation is by the calculation technique presented in section 2.1.3 equation 2.7. This was implemented as a test using the FFTW tool, several `for`-loops was needed to do this since FFTW is not able to perform FFT calculations in batches. In each `for`-loop a new plan had to be created and later with a simple command the FFT plan is executed.

4.3.2.2.2 Alternative 2 Another alternative is to implement an time to frequency converter from the definition in section 2.1.2. This alludes to implementing and coding a 2048 FFT that is especially built for a system with 1200 subcarriers.

4.3.3 Complexity

For the serial implementations of the cross correlation the complexity is determined in the same way as for the parallel only difference being that there is only one calculated complexity due to the serial part.

4.3.3.1 Cross correlation

The complexity for the serial cross correlations is mostly dominated by the three `for`-loops since there are many tasks to be performed when looping through the signal but also checking the less than - statement and incrementing the looping variables. The complexity, for the three synchronization signals, is very similar and only differs on whether the first loop is corresponding to $N_{\mathrm{ID}}^{(2)}$, $N_{\mathrm{ID}}^{(1)}$ or OFDM-symbols, seen as the variable a. The complexity equations for each line can be found in 4.4.

Table 4.4: Line by line complexity for cross correlation in serial

Line	
4	$1 + (a+1) + a = 2a + 2$
5	$a \cdot \Big(1 + 2n + (2n-1)\Big) = 4an$
6	$a \cdot (2n-1) \cdot b = 2abn - ab$
7	$a \cdot (2n-1) \cdot (1 + (n+1) + n) = 4an^2 + 2an - 2a$
8	$a \cdot (2n-1) \cdot n \cdot 4b = 8abn^2 - 4abn$
10	$a \cdot (2n-1) \cdot 2b = 4abn - 2ab$

where a and b can be found in table 4.1. The variable b is equal to 2 if the signal is complex and 1 if the signal is real.
All of the complexity equations, in table 4.4, can be simplified to the expression

$$4an^2 + 8abn^2 + 6an + 2abn - 3ab + 2 \tag{4.12}$$

and when

$$\lim_{n\to\infty}\left(4an^2 + 8abn^2 + 6an + 2abn - 3ab + 2\right) = \\ (4a + 8ab)\lim_{n\to\infty} n^2 + (6a + 2ab)\lim_{n\to\infty} n - 3ab + 2 \tag{4.13}$$

the order of complexity is $\mathcal{O}(n^2)$ for all three synchronization signals.

4.3.3.2 Find maximum value

Complexity equations for the find maximum value function of the serial code is simple since the code checks all values one time in a linear search where n is the size of the vector searched through and a is a variable dependent on which synchronization signal is used, see table 4.1. In table 4.5 the complexity row wise for the find maximum value function can be found.

Table 4.5: Line by line complexity for maximum value search in serial

Line	
1	1
2	$1 + (a+1) + a = 2a + 2$
3	$a \cdot (1 + 2n + (2n-1)) = 4an$
4	$a \cdot (2n-1) \cdot 1 = 2an - a$
5	$a \cdot (2n-1) \cdot 0.5 = an - 0.5a$
6	$a \cdot (2n-1) \cdot 0.5 = an - 0.5a$

Adding all lines together becomes the expression $3 + 8an$. Hence, the order of complexity is $\mathcal{O}(n)$ for the find max function.

4.4 Timekeeping

In order to time the program correctly when computing with a GPU it is necessary to time CPU and GPU separately. This means that when the kernel for some function is launched a GPU timer should be set. For the rest of the program that is not executed on the GPU a regular CPU timer is valid. In order to make an as accurate reading as possible the functions is located in `while`-loops that loops 1000 times and the average is saved as the time is takes to execute some function. In the case of the serial program since there is no kernels the functions is only timed with a CPU timer. This while loop is on its hands run several time to get a wider spectra of what the timing of the system runs may look like.

5 Results

The resulting run time for the full system for both the parallel and the serial decoders and also the separate time for each cross correlation function for both decoders will be presented in this chapter.

5.1 Time measurements of the full system for both decoders

The system time measurement is the time it takes for the system to analyze 14 OFDM symbols finding $N_{\mathrm{ID}}^{(2)}$, $N_{\mathrm{ID}}^{(1)}$, $N_{\mathrm{ID}}^{\mathrm{cell}}$, the subframe and the OFDM symbol number for each OFDM symbol. In figures 5.1 - 5.3 three histograms are presented showing the distribution of timings of the average time over 1000 runs. Two histogram for the parallel case, one including CUDA streams and one without, and one histogram for the serial case. The results shows that the parallel program is significantly faster than the serial program.

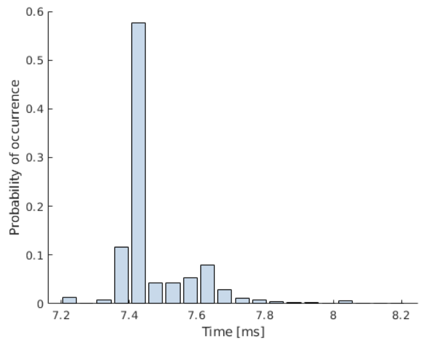


Figure 5.1: Measured time for one subframe running the whole parallel program with CUDA Streams

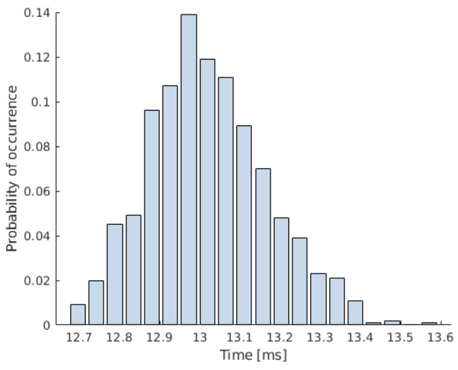


Figure 5.2: Measured time for one subframe running the whole parallel program without CUDA Streams

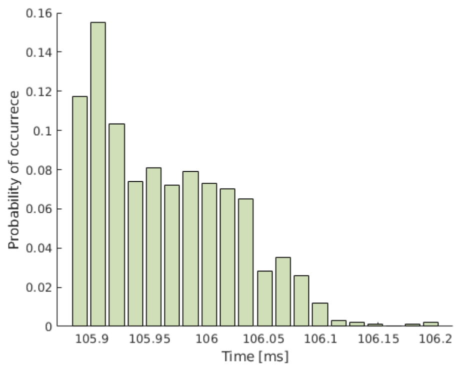


Figure 5.3: Measured time for one subframe running the whole serial program

In table 5.1, the statistics for the results when measuring the time for the full parallel decoder with and without CUDA streams as well as for the full serial decoder is shown. When using CUDA streams the system's average time becomes 42.6% faster then when not utilizing CUDA streams. The serial system is more then 10 times slower than the system when utilizing CUDA streams but has both lower standard deviation and lower coefficient of variance.

Table 5.1: Statistics for the measured run time of the full program for the parallel decoder with and without CUDA streams as well as the serial decoder

	Mean	Median	Standard deviation	Coefficient of variance
Parallel with CUDA streams	7.4668 ms	7.4150 ms	0.1177 ms	0.0158
Parallel without CUDA streams	13.0213 ms	13.0120 ms	0.1473 ms	0.0113
Serial	105.97 ms	105.97 ms	0.0597 ms	$5.6382 \cdot 10^{-4}$

5.1.1 Measured time for the FFTs of both decoders

The FFT in the parallel decoder is executed in batches and the FFT in serial (FFTW) cannot perform batches. Therefor the time is measured for two iterations of the FFTW and two batches for the FFT in the parallel decoder. The FFTW times are represented in figure 5.6 and its histogram in figure 5.8.

The parallel FFT, as can be seen in figure 5.4, has a transient time which means the first 500 values will not be considered, the remaining values are represented in figure 5.5 and the histogram in figure 5.7.

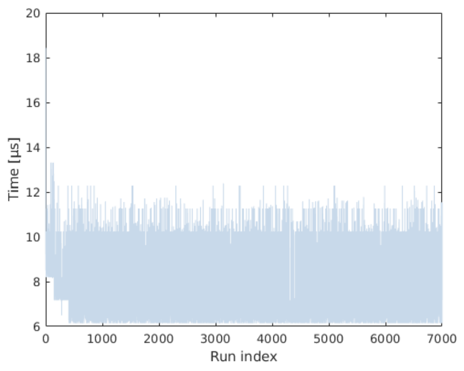


Figure 5.4: Measured time for each run in execution order for the parallel FFT

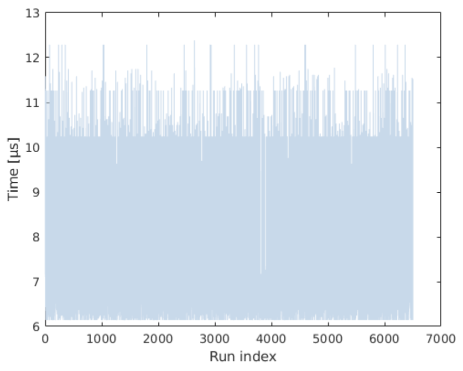


Figure 5.5: Measured time for each run in execution order for the parallel FFT without transient time

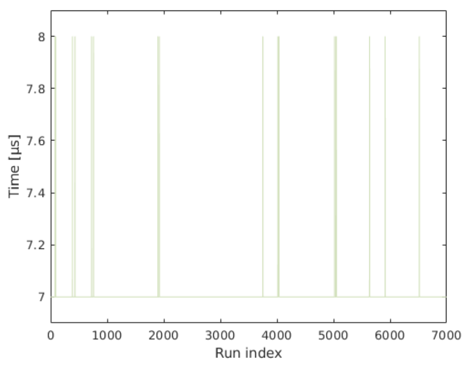


Figure 5.6: Measured time for each run in execution order for the serial FFTW

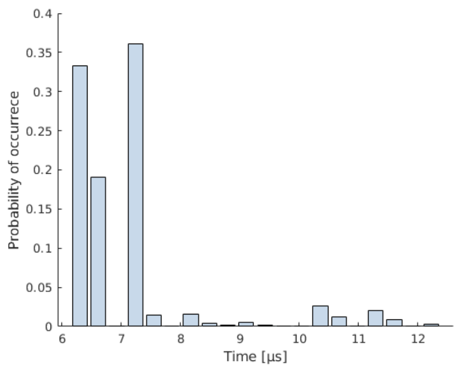


Figure 5.7: Histogram of the parallel FFT

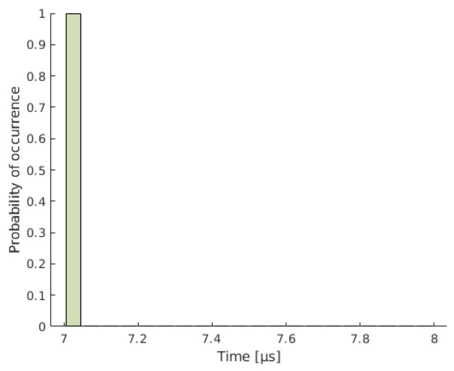


Figure 5.8: Histogram of the serial FFTW

Table 5.2: Statistics for the measured run time of the FFTs in both parallel and serial decoder

	Mean	Median	Standard deviation	Coefficient of variance
Parallel FFT	0.007 ms	0.007 ms	0.0012 ms	0.1718
Serial FFT	0.007 ms	0.007 ms	0.0518 μs	0.0074

5.2 Parallel decoder

In this section the time measurement are strictly over the kernels, including both the cross correlation and the max reduce function as mentioned in section 4.2.3, and does not include any data transfer or any other preparatory tasks.

5.2.1 Measured time for cross correlation in parallel decoder

The time measurement of the kernels are run 7000 times for each synchronization signal. As can be seen in figures 5.9, 5.10 and 5.11 the times are not fully random in their variation. There is a transient time before the time measurements becomes somewhat stable. Because of the behaviour on all three of the synchronization signals the first 500 runs will not be included in any of the remaining results for the parallel decoder.

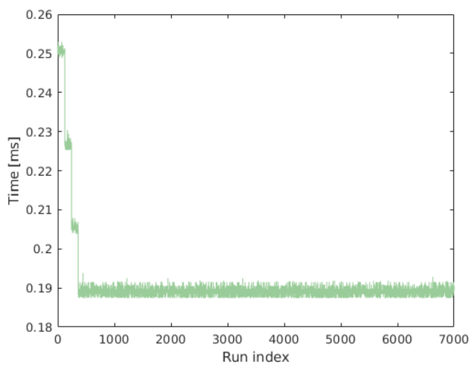


Figure 5.9: Measured time for each run in execution order for the PSS

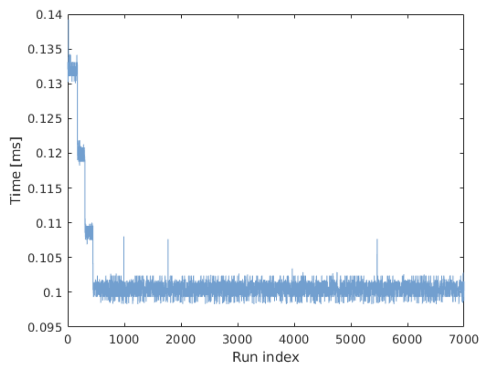


Figure 5.10: Measured time for each run in execution order for the SSS

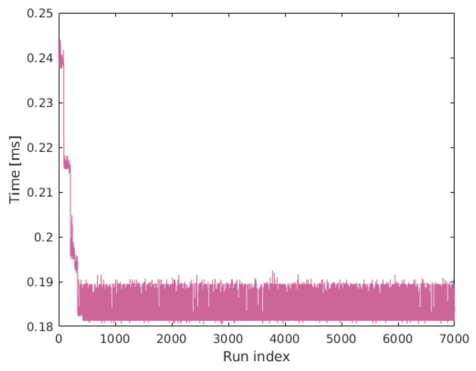


Figure 5.11: Measured time for each run in execution order for the ESS

The histograms and the time plots in figures 5.12 - 5.17 display when enough time has passed for the system to stabilize. There are a few outliers but the majority of the values stabilizes with a few microseconds in deviation. The averages, medians and variances are presented in table 5.3.

Table 5.3: Statistics for the measured run time of the cross correlations for the parallel decoder

	Mean	Median	Standard deviation	Coefficient of variation
Cross correlation and max reduce PSS	0.1891 ms	0.1894 ms	0.8618 μs	0.0046
Cross correlation and max reduce SSS	0.1005 ms	0.1004 ms	0.8299 μs	0.0083
Cross correlation and max reduce ESS	0.1827 ms	0.1894 ms	0.6886 μs	0.0038

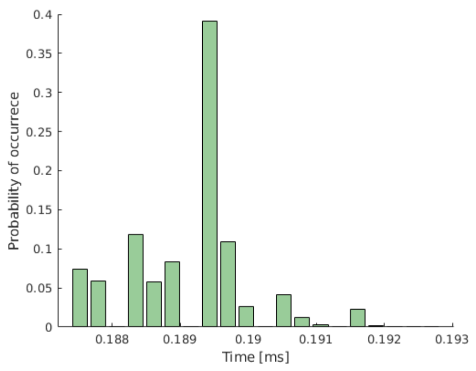


Figure 5.12: Histogram showing time measurements of the PSS cross correlation kernel

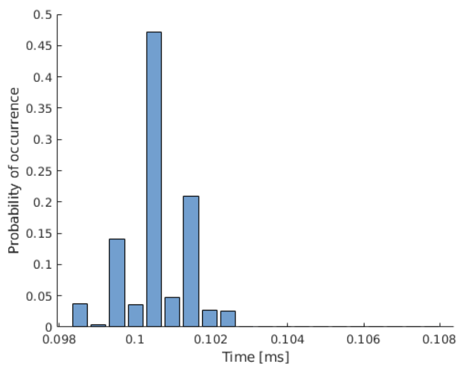


Figure 5.13: Histogram showing time measurements of the SSS cross correlation kernel

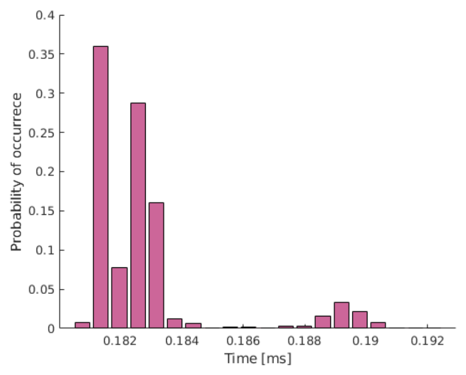


Figure 5.14: Histogram showing time measurements of the ESS cross correlation kernel

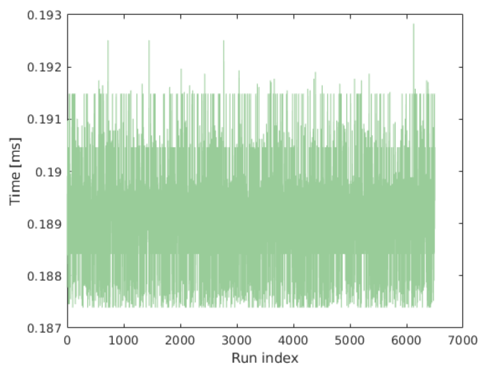


Figure 5.15: Measured time for each run in execution order for the PSS disregarding the first 500 runs

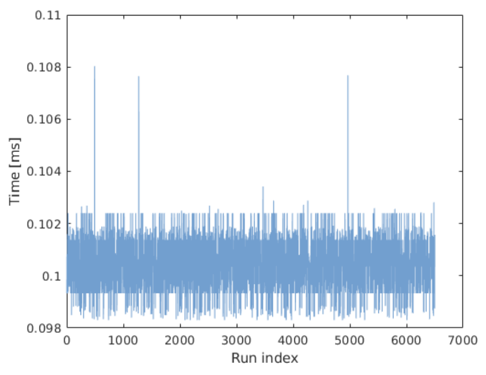


Figure 5.16: Measured time for each run in execution order for the SSS disregarding the first 500 runs

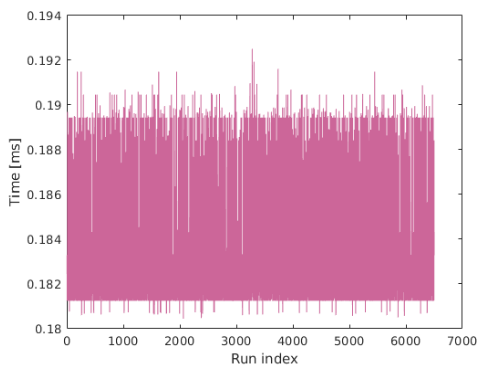


Figure 5.17: Measured time for each run in execution order for the ESS disregarding the first 500 runs

5.3 Serial decoder

In this section the time measurements for the functions calculating the different N_{ID}s and the OFDM symbol is shown. This includes cross correlation and finding a maximum function, see section 4.3.2.1, corresponding to the same calculations as in the parallel section.

5.3.1 Measured time for cross correlation in serial decoder

The normalized histograms for the cross correlations in the serial decoder is presented in figures 5.18 - 5.20 and as can be seen in figures 5.21 - 5.23, the serial code does not have a transient time and all of the collected data is thereby presented in the results in this section. The mean, median, standard deviation and the coefficient of variance for the time measurements are presented in table 5.4.

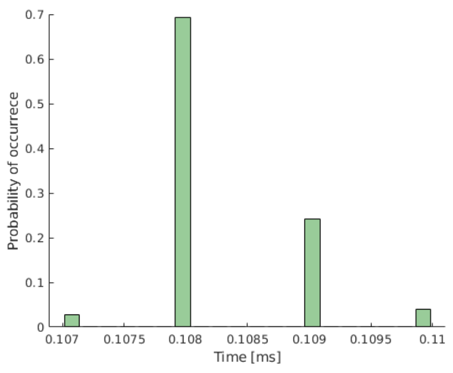


Figure 5.18: Histogram showing time measurements of the PSS cross correlation for the serial decoder

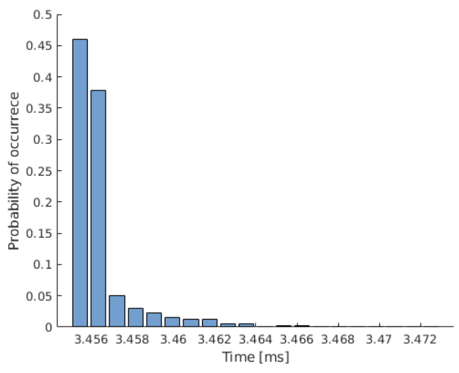


Figure 5.19: Histogram showing time measurements of the SSS cross correlation for the serial decoder

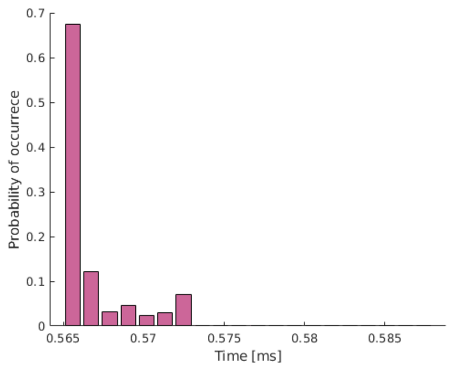


Figure 5.20: Histogram showing time measurements of the ESS cross correlation for the serial decoder

Table 5.4: Statistics for the measured run time of the cross correlations for the serial decoder

	Mean	Median	Standard deviation	Coefficient of variance
Cross correlation and find max PSS	0.108 ms	0.108 ms	0.5847 μs	0.0054
Cross correlation and find max SSS	3.456 ms	3.456 ms	1.9242 μs	0.0006
Cross correlation and max reduce ESS	0.567 ms	0.566 ms	2.1307 μs	0.0038

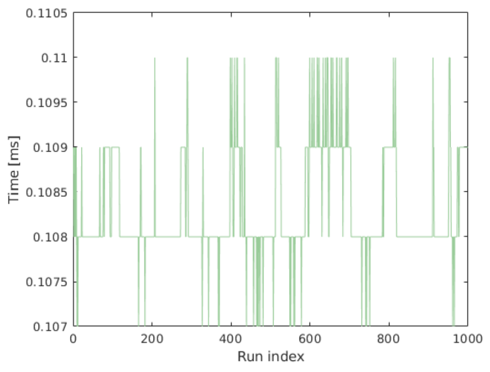


Figure 5.21: Measured run time for PSS cross correlation in the serial decoder in execution order

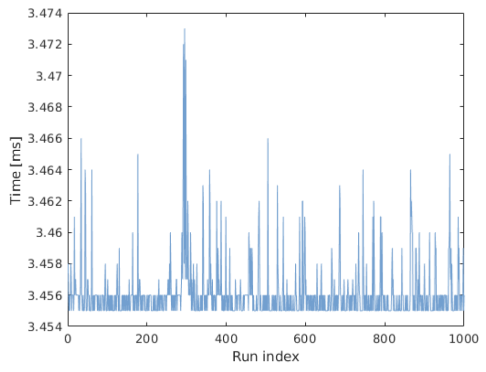


Figure 5.22: Measured run time for SSS cross correlation in the serial decoder in execution order

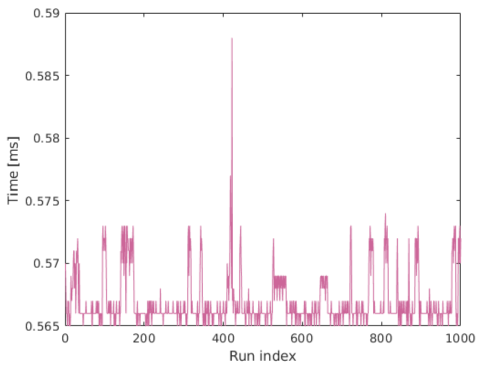


Figure 5.23: Measured run time for ESS cross correlation in the serial decoder in execution order

5.3.2 Alternative implementations

The alternative implementation of section 4.3.2.2.1, where the FFTW tool was used to do the cross correlation, slowed down the cross correlation significantly, more than double, and is thereby not a preferred solution time performance wise than the originally implemented cross correlation. The alternative implementation mentioned in 4.3.2.2.2, where the FFT was implemented by the authors, is as expected considerably slower implementation than the FFTW tool.

6
Discussion

In this chapter discussion about the project code implemented in chapter 4 and results from chapter 5 are presented together with potential improvements and changes that is relevant to the project.

6.1 Code

In this section the most important and critical parts of the decoders as well as some properties of the code is discussed. The code implemented in this project is as efficient as the authors could perform during the time period for this thesis with their knowledge of C programming.

6.1.1 Complexity

The complexity of the code is a way of determining how many times a function is executed. This means that a function with higher complexity would take longer time executing. As figure 6.1 shows, the complexity of $\mathcal{O}(n)$ is less complex than $\mathcal{O}(n^2)$ which implies that a function with complexity $\mathcal{O}(n)$ executes faster than $\mathcal{O}(n^2)$. For our system this is correct since one thread in the parallel cross correlation has the complexity of $\mathcal{O}(n)$ and the serial cross correlation has a complexity of $\mathcal{O}(n^2)$ and the latter is therefor not as fast. The figure derives that the maximum reduce function for one thread, $\mathcal{O}(\log_2(n))$, is less complex than the find maximum function, $\mathcal{O}(n)$.

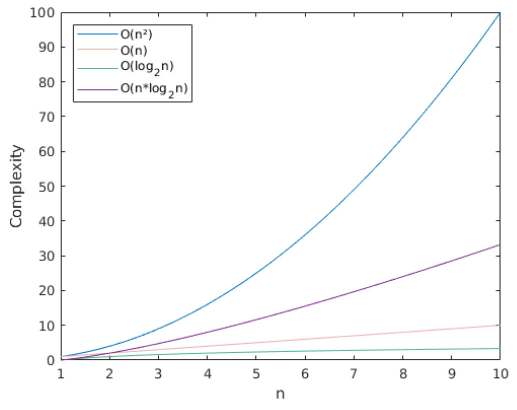


Figure 6.1: Comparison between the different orders of complexity over signal length n

One exception is that the complexity for the whole parallel cross correlator (all threads combined) has the same complexity as the serial cross correlator but is still much faster. This stems from the fact that the threads are executed in parallel leading to the run time for the parallel cross correlator to be faster even though it has the same complexity as the serial cross correlator. Another exception is that the whole maximum reduce function (all threads combined) has a higher complexity than the find maximum function, but the previous argument still holds in this case. Even though the maximum reduce function has a higher complexity the thread complexity is still lower and when executing the functions the maximum reduce function will perform faster.

6.1.2 Cross correlation

The major function in this project is the cross correlation. It is used to detect $N_{\mathrm{ID}}^{(2)}$, $N_{\mathrm{ID}}^{(1)}$ and OFDM symbol. For both the parallel and serial decoder three cross correlators are implemented, one for each synchronization signal. This could of course have been created as one cross correlation function being able to handle all three synchronization signals, but the synchronization signals are so different that the time to measure length of signal, whether or not the signal is complex and so forth would take a significant amount of time in our case and since the one of most important properties of the system is speed the three cross correlations was especially adapted after which synchronization signal it was correlating. As mentioned in section 6.1.1 there is a relation between complexity and time and if one cross correlation were to be implemented instead of three this cross correlator would have higher complexity which leads to time performance loss.

6.2 Measured time results

In this section the measured times will be compared against each other, serial to parallel, for all three synchronization signals in the decoders as well as the FFT and the whole program.
Since it is hard to measure such small time intervals as required for our system all time measurements are averages of 1000 values for the serial program as well as the full parallel program. This makes for some difference when comparing serial and parallel functions that are important to keep in mind. The serial functions will have a smoothing effect on all its results making outliers and smaller changes over time less prominent. This might make us miss patterned outliers. During the course of the project no better way of estimating the time for the CPU has been found. The GPU timing on the other hand is more precise and one run is enough to get a correct enough estimate of the kernel timings and outliers can thereby be spotted.

6.2.1 Code excluded from timekeeping

When stating that we are taking time measurements from the full program, the entire program is actually not timed. This is because we want to draw conclusions to what we believe would be the real implementation of this system. In a real system

the GPU would not re-allocate memory for each time it receives a signal, and the system would also not transfer all of the reference data to the GPU every time a synchronization signal is received. This is why all of the allocations and the majority of the data transfer happens before the start of the time measurement.
The only data transfer, `cudaMemcpy`, between CPU and GPU that occurs in the program is the received signal. This happens before every kernel is launched i.e., before the FFT and the three cross correlations. Only the relevant data for the specific kernel is transferred. When measuring the execution time for the kernels this transfer is excluded due to the fact that it is possible to transfer data directly into the GPU in real time, which would most likely be done if such a system was deployed. These data transfers are however not excluded when measuring the time for the whole system because there is no straight-forward way of of excluding them..

6.2.2 Performance of the FFT

Regarding the timing of the FFT, one important aspect to keep in mind is that the parallel program is done in two batches whilst the serial FFTW is done in two iterations. When looking at the time results, as can be seen in table 5.2, the mean as well as the median are very close to each other. The accuracy of the parallel FFT is higher and thereby there are some small, not in the table visible, differences in the results. Figure 5.6 implies that there is an accuracy issue for the serial whilst the parallel, presented in figure 5.5, gives no such indications. The standard deviation and the coefficient of variance on the other hand show some larger differences, where the serial program shows a more stable result. This stability difference can also be seen in the histograms for the two FFT's, see figures 5.7 and 5.8. This result might be due to the accuracy of the serial program or due to time variations of the GPU, or a combination of the two. A more in-depth analysis would be required to determine the exact reason. This, however, is out of the scope of this thesis.

6.2.3 Performance of the detection functions

The time measurements statistics of the decoding of the PSS, SSS and ESS are presented in tables 5.3 and 5.4 for the serial decoder.
For the decoding of the PSS the mean and the median are larger for the parallel than the serial decoder, meaning there is no real benefit from parallelizing the decoding of the PSS. An important aspect here is the fact that the system only uses 0.83% of the GPU's available resources, 1.66% when running two streams. Since a GPU is built on the technique of using many smaller processors instead one, or a few, larger ones the result might be due to three factors. The first being that the code could benefit from further optimization. The second factor is that the GPU performs worse than the CPU for the scope of this project, and the third is that the GPU is not utilized enough, meaning that the GPU would perform more efficiently for a more substantial set of data, using a larger share of the threads. The standard deviation is larger for the parallel but the coefficient of variance is greater for the CPU. The difference in standard deviations are not surprising since the mean and median are larger for the parallel functions which often entails a larger standard

deviation, but the fact that the coefficient of variance is greater for the serial than the parallel speaks against the earlier idea of the GPU having a naturally higher variance than the CPU. When studying the histograms 5.12 and 5.18 of the PSS, leaps in the time axis are visible. This is with high probability due to the accuracy of the timers. Disregarding this deformation of the histograms the two histograms do not have any extreme outliers and the standard deviation is also visible. The coefficient of variance on the other hand is harder to detect with the bare eye.
Concerning the SSS the mean and the median are smaller for the parallel than the serial decoder. This means that it is beneficial to use the GPU i.e., the data is large enough for the GPU to accomplish faster results than the CPU. The decoding of the SSS now occupy 35%, 70% with two streams, of the GPU's available resources. This strengthens the idea of the GPU not being utilized enough when decoding PSS. The GPU performs more calculations simultaneously and the CPU has a longer job queue, improving the time performance of the GPU compared to the CPU. The standard deviation and the mean derived from the parallel detection of the SSS are smaller than the corresponding part of the serial decoder, whilst the coefficient of variance shows larger time deviations for the GPU. When comparing the normalized histograms, see figures 5.13 and 5.19, one can see that the parallel decoding of SSS has a slightly less stable system in accordance to the coefficient of variance. This is contradicting the relationship between the standard deviation and the coefficient of variation for the PSS. The only factors that have changed code-wise from the kernel decoding PSS to the kernel decoding SSS is that the program runs in blocks, where each block outputs a suggested maximum, as well as the amount of threads occupied. Since the kernel outputs 21 values instead of the $N_{\mathtt{ID}}^{(1)}$ this leads to an offset of the timing for the SSS kernel since the search for the $N_{\mathtt{ID}}^{(1)}$ is not fully finished when the kernel is. One reason that makes the SSS faster than the PSS is the fact that the SSS only handles real values, in contrast to the PSS which has to handle complex values, adding complexity to the calculations.
The decoding of ESS has a lower parallel mean and median than the serial, making it beneficial to use a GPU for the detecting the ESS as well. The decoding of the ESS is the second largest calculation of the three, well noticeable in the time measurements for the serial decoder where it has the second longest measured time. In the parallel decoder it takes less time to decode the SSS than the ESS, due to the difference in complexity, and about the same as the PSS. The ESS uses 3.33%, 6.66% for two streams, of the GPU's available resources, which is more than for the PSS kernel but it is still not especially efficient. The standard deviation and the mean is smaller for the parallel decoding of ESS, while the coefficients of variance are equal. The equal coefficients of variance are hard to notice when comparing figures 5.14 and 5.20.

The means and the medians are similar to each other for all three synchronization signals both in the parallel and serial decoder. This implies that there are no uneven distributions in the run times.

6.2.4 Performance of the full program

For the time measurements of the full programs i.e., from time domain signal to $N_{\mathrm{ID}}^{(1)}$, $N_{\mathrm{ID}}^{(2)}$, subframe and OFDM symbol, the parallel decoder with CUDA streams are by far the fastest one. The parallel decoder without CUDA streams comes in second in performance, just below double the time. This since with CUDA streams, the vast majority of the time, the two streams are able to run consecutively, except for when two data transfers occur at the same time. The third and slowest is the CPU program. While we note that the CPU program is not yet optimized for speed, we believe that it overall order would remain the same, especially since there are some possibilities for optimizing the GPU program further.
The standard deviations of the three cases are almost proportional to the mean for the two parallel programs, but much smaller for the serial program. The coefficient of variance is thereby the absolute smallest for the serial code, second is the parallel program without streams and the parallel with streams is the one with most variance. Even though the mean of the two parallel programs are almost equal by a factor of two, the coefficient of variance is not. The three programs all show some variance but, as seen in histograms 5.1 - 5.2, do not indicate of any extreme outliers. This might be due to the averaging at the time measurement with the CPU clock.

6.2.4.1 CUDA streams

The use of CUDA streams has proven very efficient as can be seen in table 5.1. The whole system becomes 42.6% faster without loosing a significant amount of stability, expressed here as coefficient of variance. The system would have benefited from using more CUDA streams which was not possible due to the SSS using more than a third of the threads on the GPU. The SSS resource allocation would have to be less than a third to allow for one more CUDA stream. The number of CUDA streams could also have been increased if it was possible to perform the decoding of the PSS, SSS and ESS simultaneously. This is a constraint one can not get around since $N_{\mathtt{ID}}^{(1)}$ can not be found before or at the same time as $N_{\mathtt{ID}}^{(2)}$ due to $N_{\mathtt{ID}}^{(1)}$ being dependent on $N_{\mathtt{ID}}^{(2)}$. This holds for the OFDM-symbol as well. To find the correct OFDM symbol using ESS one needs to know what cell ID is used.

6.2.5 Difference in variation between CPU and GPU

The FFT, the decoding of SSS and the full program shows a larger variance in the GPU than in the CPU. The PSS on the other hand shows a larger variance in the CPU. However, the variance for decoding ESS is the same in the CPU and the GPU. The fact that an average is used for all CPU measurements could lead to a smaller variation for all CPU results, except for the measurements where the precision is low i.e, when the CPU timer can not measure with enough accuracy. However, we did not observe any influence of this on the results. The only program that uses the CPU timer for both parallel and serial is the full program time measurements. In these the results points towards the parallel program having larger variations. It is also important to remember that in the full parallel program the data transfers are included as well and could by themselves include variations.

When gathering the results of the thesis, it was discovered that the system would become significantly slower at what seemed to be random times. One apparent example is figure 5.16. These outliers could not be connected to anything that was implemented during the project and occurred for both tests related to CPU and GPU. Because of this the outliers are believed to be caused by the host computer performing background tasks that were not possible to switch off. This has to kept in mind when interpreting the results. However, the vast majority of the results was not affected by the background tasks. There are no clear directions on which of the CPU and the GPU makes for the most variations, a lot of the obtained results points in both directions.

6.3 Throughput and latency

Disregarding all of the steps reformatting the data and only considering the kernels cross correlation together with the maximum reduce functions the mean time it takes for the detection of one OFDM symbol for the parallel system is

$$\frac{0.007\text{ ms}}{2} + 0.1891\text{ ms} + 0.1005\text{ ms} + 0.1827\text{ ms} = 0.4758\text{ ms}$$

which is longer than the time of a subframe. This system would thereby not work without accumulating an infinite queue of jobs. The throughput and the latency of the system are 0.4793 ms whilst the throughput is

$$\frac{1}{0.4758\text{ ms/job}} \cdot 0.2\text{ ms/subframe} = 0.4203\text{ jobs/subframe}$$

also displaying its insufficiency. There is a way of making the system closer to functional though. Considering the kernels separate timings it shows how none of the four steps exceeds 0.2 ms by themselves. That means that by using more than one GPU it would be possible to line the four steps in different systems, making the bottleneck kernel the throughput. The bottleneck in this case is the PSS kernel and gives the throughput

$$\frac{1}{0.1891\text{ ms/job}} \cdot 0.2\text{ ms/subframe} = 1.0576\text{ jobs/subframe}$$

which is sufficient enough for the system to run without queue accumulation. The latency on the other hand still has the same value. This entails that the system would only detect one OFDM symbol per subframe. The system is far from being able to detect all 14 OFDM symbols. However, building such a system usually takes months and teams of experts. Still the parallel decoder performs substantially better than the serial decoder whose latency is

$$0.0035\text{ ms} + 0.108\text{ ms} + 3.456\text{ ms} + 0.567\text{ ms} = 4.1345\text{ ms}$$

and the throughput

$$\frac{1}{4.1345\text{ ms/job}} \cdot 0.2\text{ ms/subframe} = 0.0484\text{ jobs/subframe}$$

is much lower than for the parallel decoder. The bottleneck, of 3.456 ms, in the serial program is also much larger than for the parallel program.
For the parallel code the results also shows how much of the time that is spent in the CPU, the full timing is $\frac{7.4668 \text{ ms}}{14} = 0.5333$ ms/OFDM symbol, giving the fraction spent in the GPU

$$\frac{0.4758}{0.5333} = 0.8922$$

which is 89.22% and this shows that the system spends 10.78% of its time in the CPU. It might be possible to do more calculations in the GPU and thereby further speed up the system.

6.4 Parallelizing the CPU

One possible method to speed up the serial decoder could have been to use parallel programming and creating multiple threads for the CPU. The optimization would have been limited to the number of available cores, in our case 4. This is still by far lower than the number of available cores in the GPU. Hence, we conclude that we could close the gap to the GPU, but still not outperform it. Furthermore, the aim of the thesis was to see how the GPU performed parallel decoding compared to serial decoding and parallelizing the CPU would not lead us any closer to the answer.

6.5 Improvements and future work

The built programs only cover the synchronization of the 5G system. Future work would continue with the succeeding steps in the communication between the UE and the base station, both in uplink and downlink. Regarding the synchronization part of the project some constraints were set, stated in the limitations section of this report. One limitation being that the system does not include beamforming. This could be an extension of this project. Another improvement to this project is to include synchronization, the objective of this report was to investigate the use of CUDA could improve performance time-wise and the scope was thereby somewhat restricted. The natural next step would therefor be to implement so that the system could receive any part of a signal, using this data to synchronizes itself.

7
Conclusion

In this thesis parallelism of decoding the three 5G fiber synchronization signals was evaluated in comparison to decoding in serial. The hardware of the project limited the results and the measured times were much higher than expected in a real system but showed promising results. Under the right hardware conditions as well as more optimized software this could be a part of future 5G solutions.
The parallel system performed better for both latency and throughput for all synchronization signals except for the decoding of PSS, which used a too small fraction of the GPU resources to be able to reach the desired advantages of a GPU.
The use of CUDA streams for the GPU showed promising results, accelerating the system up to 42.6%. In this thesis only two streams were utilized due to hardware restraints. However, the results could possibly reach an even higher percentage if improved hardware, more processors, and additional streams were used.

Bibliography

[1] (2018). 5g open for business, Ericsson, [Online]. Available: https://www.ericsson.com/en/5g?_t_id=1B2M2Y8AsgTpgAmY7PhCfg%3d%3d&_t_q=5G&_t_tags=language%3aen%2csiteid%3a621ab6dc-927f-42e5-92e7-e47cbe5f58f3&_t_ip=81.236.111.59&_t_hit.id=Corporate_Web_Cms_ContentTypes_Pages_SectionStartPage/_3f093b9a-c3e5-4e9c-b682-eed0d972654f_en&_t_hit.pos=1 (visited on 02/27/2018) (cit. on p. 1).

[2] *High band spectrum - the key to unlocking the next generation of wireless*, ctia. [Online]. Available: https://api.ctia.org/docs/default-source/default-document-library/5g-high-band-white-paper.pdf (visited on 05/18/2018) (cit. on p. 1).

[3] S. Buzzi, C. L. I, T. E. Klein, H. V. Poor, C. Yang, and A. Zappone, "A survey of energy-efficient techniques for 5g networks and challenges ahead", *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 4, pp. 697–709, 2016, ISSN: 0733-8716. DOI: 10.1109/JSAC.2016.2550338 (cit. on p. 1).

[4] (2018). Verizon 5g technical forum, Verizon, [Online]. Available: http://www.5gtf.net/ (visited on 02/27/2018) (cit. on p. 1).

[5] *Hur ser det ut i sverige? - fakta och statistik*, ssnf. [Online]. Available: https://www.ssnf.org/globalassets/sveriges-stadsnat/fakta-och-statistik/informationsblad/ssnf_fakta_statistik_hres_ot.pdf (visited on 05/18/2018) (cit. on p. 1).

[6] *Låt oss leka med siffror för fiber*, Byafiber. [Online]. Available: http://www.byafiber.se/lat-oss-leka-med-siffror-for-fiber/ (visited on 05/18/2018) (cit. on p. 1).

[7] *Priser & villkor*, Telia. [Online]. Available: https://www.oppenfiber.se/anslut/villa/priser-och-villkor (visited on 05/18/2018) (cit. on p. 1).

[8] E. Dahlman, S. Parkvall, and J. Sköld, *4G LTE-Advanced Pro and The Road to 5G*, Third Edition. Oxford: Academic Press, 2016, pp. 1–4, 31–33, 285–288, 527–530, ISBN: 978-0-12-804575-6 (cit. on pp. 3, 5, 6, 8).

[9] *About international telecommunication union (itu)*, ITU. [Online]. Available: https://www.itu.int/en/about/Pages/default.aspx (visited on 05/23/2018) (cit. on p. 5).

[10] H. Rohling, *OFDM: Concepts for Future Communication Systems*, First Edition. Springer, Berlin, Heidelberg, 2011, pp. 5–9, ISBN: 978-3-642-17495-7. DOI: https://doi-org.proxy.lib.chalmers.se/10.1007/978-3-642-17496-4 (cit. on p. 6).

[11] (2018). Verizon 5g tf; air interface working group; verizon 5th generation radio access; physical channels and modulation (release 1), Verizon, [Online]. Avail-

able: http://www.5gtf.net/V5G_211_v1p7.pdf (visited on 03/06/2018) (cit. on p. 8).

[12] *What's the difference between a cpu and a gpu?*, nVIDIA, 2009. [Online]. Available: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/ (visited on 04/16/2018) (cit. on p. 15).

[13] *Nvidia*, nVIDIA. [Online]. Available: https://www.nvidia.com/ (visited on 04/16/2018) (cit. on p. 16).

[14] *Titan xp*, nVIDIA. [Online]. Available: https://www.nvidia.com/en-us/titan/titan-xp/ (visited on 04/16/2018) (cit. on p. 16).

[15] S. Cook, *CUDA Programming - A developers guide to parallel computing with GPUs*. 225 Wyman Street, Waltham, MA 02451, USA: Morgan Kaufmann Publishers, 2013, pp. 28, 38–39, 44–46, ISBN: 978-0-12-415933-4 (cit. on pp. 16, 18).

[16] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*. 10475 Crosspoint Boulevard, Indianapolis, IN 46256: John Wiley & Sons, Inc., 2014, ch. 3, ISBN: 978-1-118-73932-7 (cit. on pp. 17, 18).

[17] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer", *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 134–144, 2011. DOI: 10.1109/ISPASS.2011.5762730 (cit. on p. 18).

[18] J. Sanders and E. Kandrot, *Cuda by example - An Introduction to General-Purpose GPU Programming*. 501 Boylston Street, Suite 900, Boston, MA02116, USA: Addison-Wesley - Pearson Education, Inc., 2010, pp. 192–205, ISBN: 978-0-13-138768-3 (cit. on pp. 18, 19).

[19] A. Butterfield and G. E. Ngondi, *A Dictionary of Computer Science*, Seventh Edition. Oxford: Oxford University Press, 2016, ISBN: 978-0-19-968897-5 (cit. on p. 20).

[20] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems - Queueing Theory in Action*. 32 Avenue of the Americas, New York, NY 10013-2473, USA: Cambridge University Press, 2013, pp. 18, 24–25, ISBN: 978-1-107-02750-3 (cit. on p. 20).

[21] *Optimizing parallel reduction in cuda*, nVIDIA. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf (visited on 04/30/2018) (cit. on p. 24).

[22] *Fft library*, FFTW. [Online]. Available: http://www.fftw.org (visited on 05/18/2018) (cit. on p. 30).

[23] L. M. Barone, E. Marinari, G. Organtini, and F. Ricci-Tersenghi, *Scientific Programming—C-Language, Algorithms and Models in Science*. 5 Toh Tuck Link, Singapore 596224, Singapore: World Scientific Publishing Co. Pte. Ltd., 2013, ch. 18, ISBN: 978-9814513401 (cit. on p. 30).