Eindhoven University of Technology

MASTER

Strain visualization of ultra sound signals processed by general purpose graphic process unit

Alvarez Loya, L.E.

Award date:
2012

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 20. Jan. 2018
Thesis Report

Strain Visualization of Ultra Sound Signals Processed by General Purpose Graphic Process Unit

Luis Alvarez

email: l.e.alvarez.loya@student.tue.nl

Student ID 0754555

Technische Universiteit Eindhoven

Prof. Dr. Gerard de Haan

Technische Universiteit Eindhoven

Philips Research Eindhoven

Dr. Harm Belt and Dr. Octavian Soldea

Philips Research Eindhoven

August 27, 2012
Abstract

Medical Imaging is a technique aimed to develop tools to improve medical diagnosis and procedures. Medical imaging applications use different types of signals to create images of the inside of the human body. It is a very important technique that decreases the complication rate during procedures and provides with more information for more accurate diagnosis. One characteristic of image processing algorithms is that there are large data sets to be processed in order to provide visualization. In consequence, some diagnose imaging techniques take too long before their results are ready and the doctor can interpret them. A different approach can be used to decrease the time consumed by medical imaging algorithms.

In this thesis, it has been explored the possibility of using Graphic Unit Processors (GPUs) for implementing real time medical imaging applications, more specifically ultrasound imaging for catheter ablation. The reader can find in the report the analysis of a set of medical imaging algorithms and its implementation on a GPU. There is a comparison with a CPU implementation and conclusions about the benefits and implications of developing a medical imaging application on top of a GPU.

Introduction

The objective of this thesis is to implement a real time (RT) medical imaging application on a graphic processing unit and compare it with a single core CPU implementation in order to measure the benefits of using General Purpose GPUs (GPGPU) for medical imaging algorithms. The greatest value of this document is: i) the analysis of the algorithms to be implemented on a GPGPU; ii) the challenges for achieving an optimal GPGPU implementation of such algorithms and iii) the comparison with an implementation on a CPU.

Graphic cards have the characteristic of using many processing cores (up to hundreds) for image rendering. The main idea is that all cores in a GPU work independently on different data exploiting the absence of data dependencies. NVIDIA General Purpose GPUs provide a programming language called CUDA C (see [1]) which allows the developers to use the power of the GPGPUs for running intensive algorithms.

GPGPUs are very efficient to implement algorithms where large data sets have to go through the same computations. Examples of this are: physic phenomena simulations and medical imaging (the reader can refer to [2] for an example of a MRI application accelerated with GPGPU). The reader finds in Section 1 a brief introductory explanation of how medical imaging applications can get benefits by being implemented on a GPGPU.

For this thesis, the RT application implemented on a GPGPU is a visualization tool for catheter ablation which is a medical procedure for treating arrhythmia. In Section 2, the reader can find more information about arrhythmia and the procedure of catheter ablation; and the motivation for using a visualization tool for the procedure of catheter ablation.

The visualization tool receives and processes ultrasound signals coming from the heart tissue. Such ultrasound signals are the reflection of an electric pulse sent by transducers mounted on the Catheter used in the procedure. This process is explained in the Subsection 2.2. The ultrasound signals are processed and displayed in order to provide "eyes" inside the human body, to the physicians doing the procedure. Therefore, the procedure can be easier, faster and more accurate.

The input ultrasound signals are processed by two different algorithms. One is contrast enhancement where the input is converted into a gray scale that can be visualized. This algorithm is described in Section 3. The other algorithm is strain imaging where the Young's modulus is used to visualize the relative stiffness on the tissue. This algorithm is described
in Section 4. Furthermore, it is explained that cross correlation function is used to derive the strain visualization. The algorithm for contrast enhancement was implemented on a GPGPU in a previous phase to this thesis. The reader can see the details in [3].

The core of the algorithm to generate a strain visualization consists of the computation of a cross correlation function. The algorithm for computing the cross correlation function can be implemented in time domain or frequency domain. The time domain implementation is first analyzed and optimized for decreasing the number of computations required. Such analysis is presented in Section 6, where it is also explained how the algorithm was implemented on a CPU and on a GPGPU. At the end of Section 6 the reader can find the benchmark of the different implementations in a CPU and a GPGPU.

The cross correlation is also implemented in frequency domain. For this a FFT and IFFT were implemented on a GPGPU. The analysis of the frequency domain approach for strain visualization is analyzed in Section 7. Furthermore, in this section the design of the FFT implementation on a GPGPU is explained. At the end of the section the reader can find a benchmark of the of the new GPGPU FFT implementation, an open-source CPU FFT and the FFT developed by NVIDIA.

The resulting cross correlation function derives the strain visualization. The strain imaging visualization tends to be too noisy where the displacement of the tissue is small. Therefore the signal after the strain algorithm goes through a step of post-processing. The purpose of this is to give the physicians a more meaningful visualization. The decisions taken for the post processing step are described in Section 8.

The contrast enhancement visualization and the strain visualization contain different information about the heart tissue. The physician doing the procedure may find more meaningful one or the other visualization at certain stage of the procedure. The physician then, may want to take a look to the other visualization in order to have more information. He may want to often toggle from contrast enhancement to strain visualization (and backwards) during the procedure. Thereby, both processes should be run in parallel to allow toggling between visualizations.

In Section 9 the reader will find the benchmark of CPU and GPGPU to compare the execution time when processing both contrast enhancement and strain algorithms. In this Section there is an analysis of the quality opportunities provided by both CPU and GPGPU implementations.
Contents

1 Medical Imaging Context 6

2 Visualization Tool 6
   2.1 Catheter Ablation Project ...................................... 7
   2.2 Ultrasound Input Processing ................................... 9
   2.3 M-Mode Imaging ............................................... 13

3 Contrast Enhancement Visualization 14

4 Strain Visualization 17
   4.1 Strain Computation (Local Displacement Estimation) .......... 18

5 Motivation for GPGPU Implementation 21

6 Strain Visualization Implementation 24
   6.1 CPU Implementation .......................................... 24
      6.1.1 Naive Implementation .................................. 24
      6.1.2 First Optimization: Avoid Redundant Multiplications .... 26
      6.1.3 Second Optimization: Reducing Additions ................. 29
   6.2 GPGPU Implementation ........................................ 32
      6.2.1 GPGPU Implementation Properties ......................... 32
      6.2.2 Naive Implementation .................................. 37
      6.2.3 First Optimization: Loading on Shared Memory .......... 38
      6.2.4 Second Optimization: Reduce Multiplications .......... 40
      6.2.5 Third Optimization: Reduce number of Additions ....... 42
      6.2.6 Fourth Optimization: Increment GPGPU Usage .......... 43
      6.2.7 Fifth Optimization: Increment GPGPU Usage and Reduce Operations (Cycles) ............................................. 45
      6.2.8 Comparison of Optimizations .............................. 47
   6.3 Benchmark Time Domain: GPGPU Implementation .......... 50

7 Strain Implementation on Frequency Domain 55
   7.1 Overview of the algorithm ................................... 55
   7.2 GPGPU Implementation ........................................ 56
   7.3 FFT: GPU Implementation .................................... 58
   7.4 Benchmark FFT: GPU Implementation, FFTw, cuFFT ........... 63
   7.5 Frequency Domain vs Time Domain .......................... 65

8 Post-Processing 69

9 System Benchmark: CPU vs GPU 74
   9.1 Conclusions ................................................ 75

10 Thesis Conclusions 78

A Functions used in FFT Strain Visualization Algorithm 79
   A.1 Complex Multiplications .................................... 79
   A.2 Find the Lag of the Maximum Cross Correlation Value ....... 80
1 Medical Imaging Context

Throughout the years, medical science has evolved towards providing a better and longer life for every human being. Medicine has taken profit of the technological advances which make available new tools and methods. Medical imaging is one important field fruitfully used nowadays. Doctors can now have visualization equipment that gives them more information for diagnosis. Ultrasound visualization and magnetic resonance imaging are broadly used examples of medical imaging. Such techniques require intensive computation power that may imply trade-offs on quality for achieving a reasonable performance. However, the surfacing of general purpose graphic processing units leads to a new software paradigm that can handle a larger bulk of intense computation requirements.

Medical imaging is broadly used as a diagnosis tool. X-rays and MRI give doctors a better understanding of the condition of the patient and help them decide the best way to treat them. There is still a lot of research currently focused on improving these techniques. Engineers are working on new and improved algorithms for processing the information and providing doctors with better visualization tools.

An important amount of the available algorithms require very intense computations over huge amounts of data. When these algorithms are processed by CPU implementations, the time consumed is too long because of the sequential nature of the CPU architecture. I.e. consecutively all computations are executed on each piece of data even if such pieces of data have no dependencies between them. This approach may still be useful for procedures like MRI visualization where the time to get results is not so critical. However, when we talk about interventions, the response time plays an important role.

A fresh approach consists of taking advantage of the intrinsic parallel nature of the processing (Since same computations may be done on different pieces of data at the same time). GPGPU implementation is a promising approach for achieving medical imaging with a better time performance (with no quality sacrifices) by profiting from the parallel nature of image processing.

The improvement in performance due to the usage of GPGPUs may allow to get results in shorter periods of time. The processing time reduction may give room for larger data sets which contain information at a higher resolution. In conclusion, the introduction of a GPGPU implementation for the medical imaging algorithms can produce better quality images in less time. Furthermore, medical imaging for interventions require a real time (RT) application with hard real-time constraints and low latency.

RT medical imaging implies a major challenge for engineers. While diagnosis visualizations (e.g. Radiographies) are softly constrained in time, visualization tools for interventions must be very precise. Visualization tools that help doctors on medical procedures have very tight constraints as human lives are at stake during the procedures. There is no room for delays or imprecise timing and doctors expect a good visualization quality. In this scenario, a GPGPU implementation becomes more than an alternative for getting results in a shorter time; it becomes the option for meeting the real-time constraints that medical interventions require while keeping a good quality.

2 Visualization Tool

As stated in the Introduction, the purpose of this thesis is to explore the possibility of using a GPGPU implementation for a real time visualization tool meant to be used during a medical procedure. This section is intended to provide the reader with background information to have a clear view of the thesis purpose. First, the medical procedure and its objective along
with the motivation for using a visualization tool during this procedure are briefly described. Furthermore, the origin of the visualization tool project is introduced including the role of this thesis on such project. The visualization tool uses ultrasound signals, the origin of such signals is explained and the algorithms to process them are introduced in this section. The visualization tool uses M-Mode Imaging which is explained at the end of the section.

The algorithms used by the visualization tool introduced in this section are the algorithms to be implemented in a GPGPU. As a reminder, a more detailed explanation of the contrast enhancement visualization algorithm is presented in Section 3 and for the strain visualization algorithm in Section 4. And the implementation for the strain visualization algorithm is explained in Section 6 for the time domain and in Section 7 for the frequency domain. A list of all the variables used throughout the document is shown in Table 1. The reader may refer to the table to look for information of any variable introduced.

2.1 Catheter Ablation Project

Catheter ablation is a medical procedure intended for curing arrhythmia. Arrhythmia is originated by malign tissue that causes the heart to malfunction. The normal behavior of the heart is based on electrical pulses produced by special tissue known as the sinoatrial (SA) node, that makes the heart contract and relax at certain rhythm, any irregularity on the behavior is called arrhythmia. Such irregularities may be cause because the electrical pulse is not started in the SA node and the electrical pulses spread over the heart in a fast and disorganized way. Such is the case of atrial fibrillation (AF) which is a very common type of arrhythmia. More information about arrhythmia, AF and other types of arrhythmia is in [4].

The medical intervention catheter ablation consists on bringing to the heart a narrow tube (the catheter) that is inserted in the human body normally by the groin. Such catheter contains a device that is used for locally treat the malign tissue and hence solving the problem of beat rhythm irregularities. This procedure is nowadays monitored by rather rudimentary methods and most of them are based on analysis after the ablation procedure. This means that there is a chance of a non-successful procedure which will cause the patient to go over the ablation procedure more than once. There is even the possibility of going to surgery because the procedure is not working.

Philips Research Eindhoven has the incipient and innovative approach of a real-time visualization tool for catheter ablation [5]. It offers a real time visualization of the heart that provides feedback to the surgeon while the procedure is still on. In the Philips approach, the catheter used has mounted several ultrasound transducers that give information about the tissue which can be visualized. In the context of this thesis four ultrasound transducers are considered.

The information provided by the ultrasound transducers can be used to give the physician vision of the area he is working at during the ablation procedure. When the physician is able to see the heart’s tissue he can reduce the complication rate and increase the success rate of the intervention. Since he can see what he is doing, the procedure can be shorter. That reflects in less expensive procedures and more patients can be candidates to it because it is more effective and easier for the patient. Furthermore, less experience physicians can also do it. In Table 2 the reader can find a summary of the benefits of a visualization tool for the Catheter Ablation procedure.

Philips Research’s first target type of Arrhythmia is Atrium Fibrilation (AF). AF is the most common cardiac arrhythmia and it is usually treated with medication. However, depending on the type of AF, medication may not be enough and a medical procedure or even surgery need to be applied. Only when other types of treatment are not effective, heart surgery used. Therefore, medical procedures like catheter ablation are applied.
Table 1: References to used notation

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
<th>Remarks</th>
<th>Typical Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t$</td>
<td>Continuous time index</td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>$n$</td>
<td>Discrete time index</td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>$p_m(t)$</td>
<td>Electric Pulse</td>
<td>Electric Pulse for the ultrasound transducer</td>
<td>N/A</td>
</tr>
<tr>
<td>$v_m(t)$</td>
<td>Ultrasound Response</td>
<td>Impulse reflected from the Tissue</td>
<td>N/A</td>
</tr>
<tr>
<td>$f_a$</td>
<td>Axial Frequency</td>
<td>&quot;Sampling frequency of the reflection&quot;</td>
<td>200MHz</td>
</tr>
<tr>
<td>$f_h$</td>
<td>Pulse Repetition Frequency</td>
<td>Defines how often an ultrasound pulse is directed</td>
<td>1KHz</td>
</tr>
<tr>
<td>$T_h$</td>
<td>Pulse Period, $T_h = \frac{1}{f_h}$</td>
<td>It is the time elapsed between U.S. pulses.</td>
<td>1ms</td>
</tr>
<tr>
<td>$L_{depth}$</td>
<td>Depth monitored</td>
<td></td>
<td>15.7mm</td>
</tr>
<tr>
<td>$T_d$</td>
<td>Sampling Period</td>
<td>Defines the time to listen to the Ultrasound Response. $T_d = 2 \times \frac{L_{depth}}{c}; \ T_d \leq T_h$</td>
<td>20.48µs</td>
</tr>
<tr>
<td>$V_m(n)$</td>
<td>Stored US Response</td>
<td>This is called the input vector throughout this document and stores an RF line</td>
<td>N/A</td>
</tr>
<tr>
<td>$N$</td>
<td>Size of the Stored Response</td>
<td>The size $N$ is defined as $N = T_d \times F_a$</td>
<td>4096</td>
</tr>
<tr>
<td>$A_m(n)$</td>
<td>RF line after envelope detection</td>
<td>The range of values is $[0,1]$</td>
<td>N/A</td>
</tr>
<tr>
<td>$w$</td>
<td>Square Window Size (expressed in samples)</td>
<td>Defines the size of the segments to detect displacement</td>
<td>64</td>
</tr>
<tr>
<td>$s$</td>
<td>Step Size (expressed in samples)</td>
<td>Defines the sliding step of the windows</td>
<td>4</td>
</tr>
<tr>
<td>$L_{max}$</td>
<td>Maximum Lag</td>
<td>Defines the Max Lag on the Cross Correlation Function</td>
<td>20</td>
</tr>
<tr>
<td>$L$</td>
<td>Number of Cross Correlation Values</td>
<td>It is computed as $L = 2 \times L_{max} + 1$</td>
<td>41</td>
</tr>
<tr>
<td>$I$</td>
<td>Number of Windows within the Vector</td>
<td>It is computed as $I = \lfloor (N - w)/s + 1 \rfloor$</td>
<td>1009</td>
</tr>
<tr>
<td>$p_i(l)$</td>
<td>Cross Correlation function</td>
<td>Contains for every window $i$ the cross correlation function with $L$ values corresponding to the $L_{max}$ defined</td>
<td>N/A</td>
</tr>
<tr>
<td>$V_{dis}(i)$</td>
<td>Displacement Array</td>
<td>Contains for every segment $i$ the local displacement computed.</td>
<td>N/A</td>
</tr>
<tr>
<td>$R_{size}$</td>
<td>Size of the Real Vector to apply FFT. It has to be a power of 2</td>
<td>It is defined as: $R_{size} = 2^{\log_2(2w-1)}$</td>
<td>256</td>
</tr>
<tr>
<td>$F_{size}$</td>
<td>It is the size of the frequency domain vector.</td>
<td>It is computed as $F_{size} = R_{size}/2 + 1$</td>
<td>129</td>
</tr>
</tbody>
</table>
Table 2: Summary of the benefits of a Visual Tool for Ablation

<table>
<thead>
<tr>
<th>Benefits for EP Physicians</th>
<th>Benefits for Patients</th>
<th>Benefits for Society</th>
</tr>
</thead>
<tbody>
<tr>
<td>* Increase Success Rate</td>
<td>* Fewer Callbacks</td>
<td>* Less Costs</td>
</tr>
<tr>
<td>* Reduce Complication Rate</td>
<td>* Shorter Procedure = less discomfort</td>
<td>* Higher productivity (The</td>
</tr>
<tr>
<td>* Faster Procedures</td>
<td></td>
<td>patients only go through the</td>
</tr>
<tr>
<td>* Less Experienced EP Physicians can do ablation</td>
<td></td>
<td>procedure once and by</td>
</tr>
<tr>
<td></td>
<td></td>
<td>consequence they go back to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>work faster.)</td>
</tr>
</tbody>
</table>

There are more than one type of Catheter Ablation procedure but Philips has focused on RF Ablation. This method uses an electric pulse to locally burn the maligned tissue. RF Ablation is usually safe but it can also derive certain complications such: blood cots, bleeding, heart-attacks, nerve injuries or thrombophlebitis. The reader can refer to [6] for further information about the complications mentioned. The success rate of the procedure and the avoidance of complications could be reduced if the doctor could see what it is happening in the heart during the intervention.

This is an ambitious project which is planned to have important levels of intense computations with high parallelism. It is my assignment to create a benchmark set of this real time tool that helps determining the performance difference between a CPU and a GPGPU implementation. My contribution is exploring the possibility of using GPGPU for medical imaging applications in order to have better time performance. Decreasing the time for executing the algorithms would enable the system to work with larger data rates which will improve the quality of the product. I.e. the application can be a high precise hard real-time visualization system with a better quality due to high resolution on the processed data.

2.2 Ultrasound Input Processing

In order to produce a visualization, the visualization tool receives an ultrasound signal from the heart tissue. These signals contain information about the tissue on different depths. The visualization tool processes the signals and derive a visualization. The origin of the ultrasound signals is explained in this subsection.

The general view of the origin of the ultrasound signals is depicted in Figure 1: The transducers emit an electric pulse \( p_m(t) \) at a pulse repetition frequency \( f_h \) where \( m \) is the pulse index \( (0 \leq m < \infty) \) and \( t \) is the time index \([0, T_e] \), the period \( T_e \) is the duration of the electric pulse. The pulse touches the tissue and a reflection \( v_m(t) \) is sent back to the transducer. During a period of time \( T_d \) the transducer listens to the reflection \( v_m(t) \). Then the signal is sampled at a frequency \( f_a \) and digitized. The digitized signal is the vector \( V_m(n) \). At a rate determined by \( f_h \) the system will receive a vector \( V_m(n) \) which is the one to be processed in order to generate the visualization. There is more analog circuitry involved in this process that is not explained since it is not part of the scope of this thesis.

The transducers do not emit the electric pulse and sense at the same time. Therefore, the period of time between electric pulses \( T_h \), defined as \( T_h = \frac{1}{f_h} \), is longer than time to listen to the reflection: \( T_d \leq T_h \). Thus, the transducers have enough time to sense the reflection \( v_m(t) \) before they have to emit another pulse \( p_m(t) \).
The reflection $v_m(t)$

It has been explained that at a frequency $f_h$ an electric pulse is directed to the heart tissue and a reflection is sensed, sampled and digitized. And it is the digitized signal what serves as an input to the visualization tool to generate the visualization. How the tissue reflects the electric pulse and how it provides information about the tissue is explained as follows:

When the pulse $p_m(t)$ is emitted, it touches the tissue and due to tissue properties part of the pulse spectrum is reflected. As the pulse continue traveling deeper in the tissue, different parts of its spectrum will be reflected at different depths due to differences in tissue properties. Therefore, the reflection $v_m(t)$ which is sensed by the transducer at different times $t$, contains information of different layer of the tissue. The earliest reflections sensed correspond to shallower layers and the latest reflections correspond to the deeper layers. Therefore, the longer the transducers sense the reflections, information from deeper layers will be recorded. A graphic sketch of the electric pulse traveling into tissue and its reflections at different times is shown in Figure 2.

The maximum depth to be monitored by the visualization tool is a constant defined as $L_{\text{depth}}$ and represents how deep into the heart tissue (mm) the visualization tool can provide information. The $L_{\text{depth}}$ is decided to cover the deepest layers in the heart tissue that provide important information to the physician doing the catheter ablation. In order to allow the transducers to sense the whole depth $L_{\text{depth}}$, the time $T_d$ the transducers is defined as $T_d = 2 \times L_{\text{depth}}$ where $c$ is the speed of sound. Thus, the reflection $v_m(t)$ sensed, sampled and digitized (stored in $V_m(n)$) contains information of the heart tissue at a depth $L_{\text{depth}}$.

The size of $V_m(n)$ is determined by the sampling frequency $f_a$ and the period of time $T_d$ when the reflection is sensed by the transducers. The size is defined as $N = f_a T_d$. This size correspond to a number of samples that represent the maximum depth $L_{\text{depth}}$ that is monitored by the system. The relation of size of the vector $V_m(n)$ with the maximum depth monitored is expressed in Equation (1).

$$N = \frac{2f_a \times L_{\text{depth}}}{c}, \quad \text{where } c \text{ is the speed of sound.} \quad (1)$$

At each period $T_h$ a reflection is received and sampled for a period of $T_d$ at a sampling frequency $f_a$. The sampling frequency depends on the spectrum range of the transducers where the central frequency is $f_c$. In order to cover all the spectrum of the transducers a Nyquist frequency $f_{\text{Nyquist}}$ is defined where $f_{\text{Nyquist}} \geq f_c$. Hence, the sampling frequency $f_a$ is simply: $f_a = 2f_{\text{Nyquist}}$.

The transducers used have a central frequency $f_c$ equal to $30\text{Mhz}$. And the Nyquist frequency was chosen to be $f_{\text{Nyquist}} = 100\text{Mhz}$. In Figure 3 there is an example of the frequency
Figure 2: Pulse $p_m$ traveling through the heart tissue.
★ Black arrow represents the electric pulse emitted by the transducer.
★ Blue arrows represent the sensed reflection at times $t, t', t''$ where $t < t' < t''$.
★ Gray arrows represent reflections not sensed by the transducer.

spectrum of 20 $V_m(n)$ vectors sensed by the transducer.

Figure 3: Frequency spectrum of 20 Inputs $V_m(n)$

Visualization Algorithms

It was previously introduced the source of the signals that are received by the visualization tool. The next point is to briefly introduce the algorithms applied to the input signals that derive the visualization. First, it is important to remind that the sampling frequency $f_a$ determines
the resolution of the ultrasound signal and the size of the input vector $V_m(n)$. I.e. the higher the sampling frequency, the larger the input vectors to be processed. And the pulse repetition frequency (PRM) $f_h$ identifies how often a new input vector $V_m(n)$ will arrive. I.e. the higher the PRM the higher the input rate of the system which implies less time to process every input vector $V_m(n)$.

Every vector $V_m(n)$ $0 \leq m < \infty$ is a radio frequency (RF) line with information about the tissue. They are received by the visualization tool and every single RF line goes through a number of processing steps before they can be displayed. In Figure 4 the reader can see an example of how the RF lines inter arrival can be seen in terms of depth and time.

![Figure 4: RF Lines arriving at $f_h$ and sampled at $f_a$](image)

During the ablation there are changes on the tissue which modify the physical properties. The purpose of the algorithms is to give meaningful visualization about these properties. Some of these properties are: echogenicity, contractability, blood perfusion and stiffness.

In this thesis, two algorithms which produces different visualizations are considered for the visualization tool. It is possible that more algorithms or modification on the current algorithms can be developed in the future. One of the algorithms computes a contrast enhanced visualization of the echogenicity of the tissue. We call this the contrast enhancement visualization algorithm. The second algorithm computes a visualization of stiffness changes on the tissue, this is done by computing a strain imaging. We call this the strain visualization algorithm. In Figure 5 the reader can see an image that shows the different processing steps to compute the strain and the contrast enhancement visualizations.

First, the vectors $V_m$ containing the RF lines arrive to the system. Then the values are normalized to a range $-1 < V_m(n) < 1$ and the output goes to strain visualization processing and to contrast enhancement visualization processing. The results are stored in a buffer, the input rate of the buffer must be equal to the input rate of the tool. This means, before an input vector $V_{m+1}$ arrives the vector $V_m$ should be processed and stored in a buffer. Finally, the signals are taken from the buffer, scaled and gamma corrected to be visualized in a display. Scaling is required because the information stored in the buffers is too big to be fit in a display. Because of the human eyes properties, the human perception of brightness is not linear therefore Gamma correction is apply for mapping into a non linear proportion. Then the values are quantized to luminance values in the range $0 - 255$ that are ready to be displayed. The frequency in which the buffer is being read depends on the desire frame rate of the visualization. A normal input rate $f_h$ is $1kHz$ and a usual display frame rate is not higher than $30Hz$.

In Figure 5 it is already introduced that the strain visualization requires the cross correlation
function to be computed; and the contrast enhancement visualization requires first the envelope of the signal and then modify the dynamic range of the signal with clipping and stretching. More details about contrast enhancement visualization are given in Section 3 and details about strain visualization are given in Section 4.

2.3 M-Mode Imaging

In this Section, the technique used to display the processed ultrasound signals is explained. This technique is M-Mode imaging.

M-Mode Imaging is a visualization of the changes in time of the input ultrasound images. In this project, that means the visualization of the ultrasound signals reflected by the heart tissue. The different ultrasound signals are displayed from left to right along the time axis (horizontal axis). The generated visualization presents the changes over time of the analyzed object.

In this project, the processed lines are displayed vertically from left to right. Hence, vertically the user can see the layers of heart’s tissue at different depths; and horizontally the user can see the history of the procedure. An example of how the contrast enhancement visualization looks like is given in Figure 6.
Figure 6: Depth Visualization. The input lines are displayed from left to right. I.e. the most right line is the most recent and the most left one is the oldest. In the vertical way, the bottom part of the image is the deepest of the tissue.

3 Contrast Enhancement Visualization

In Section 2.1 the general processing of the Visualization Tool was introduced. This Section is focused on the Contrast Enhancement Visualization. In Figure 7 there is a diagram with the processing steps required to generate the contrast enhancement visualization.

First, the envelope function of the normalized RF Lines is computed. The normalized RF lines have values in the range $(-1, 1)$ and the derived envelope functions are within the range $[0,1)$. The enveloped functions are no longer a RF lines but they are known as A Lines. In the Second step, the A Lines are clipped and stretch. On the third step, the result is stored in a buffer.

Envelope Detection

Envelope detection consists on finding a function that covers the input signal. An example of what the envelope detection does is shown in Figure 8.

In order to find such cover function, as depicted in Figure 9, the input signal is split in two: one is multiplied by a $sine$ function and the other is multiplied by a $cosine$ function. Then the two signals are filter by a Low Pas Filter (LPF). After, the square root of the sum of the square
signals. The *sine* and *cosine* functions have the center frequency of the spectrum of interest which in this case is the spectrum range of the transducers. Therefore, the frequency of the *sine* and *cosine* functions is the center frequency of transducers $f_c$. Normally a low pass filter is used at the end of the process as depicted in the dashed block in Figure 9, however this filter does not provide an important improvement on the visualization and it was neglected.

The envelope function covers the maximum values of the input function and yields the characteristic shape of the input. The visualization of such signal is visually more meaningful than the RF line. In Figure 10 two M-Mode Images are shown. The left image is the visualization of an RF line and the right image is the visualization of an A line (RF line envelope function). For visualizing an RF line, the range of its values (-1,1) was mapped into a range [0,1].
Dynamic Range Transformation

The purpose of modifying the dynamic range of the signal is to have a better contrast image when the gamma correction is applied. The algorithm needs an upper threshold $ut$ and a lower threshold $lt$. And a minimum signal value $minV$ and maximum signal value $maxV$. The dynamic range to be stretch is in the range $(lt, ut)$ and it will be stretched to fit in the range $(minV, maxV)$. E.g. when the A line $A_m(n) > ut$ then it is converted to $A_m(n) = maxV$. That is possible using the equation 2. The values of the A line that are outside the range $(lt, ut)$ are set to $maxV$ if $A_m(n) > ut$ or to $minV$ if $A_m(n) < lt$. This process of clipping and stretching the dynamic range is called histogram stretching, the histogram of the values of the signal is clipped on the extremes and stretch.

$$A_m(n) = \begin{cases} 
\frac{(A_m(n) - lt) \times maxV}{ut - lt} & (A_m(n) < lt) \& (A_m(n) > ut) \\
maxV & A_m(n) > ut \\
minV & A_m(n) < lt 
\end{cases}$$

The contrast enhancement is derived from the clipping and stretching the dynamic range of the A lines. In Figure 11 the reader can see the difference with and without stretching the dynamic range. It is easily noticeable that when the dynamic range is not modified, the gray color is dominant. But when the signal limits are clipped out and the rest of the signal is stretch, the dark colors become darker and the light color become lighter. That gives a better view of the ultrasound signals.
4 Strain Visualization

This section concerns the algorithm for strain visualization, intended for showing local stiffness changes inside heart tissue during ablation. The motivation for developing this visualization is introduced. First, the definition of stiffness is introduced as well as the means to measure it. Then we explain the reasons that bore the development of a strain visualization instead of directly developing an (absolute) stiffness visualization, as well as the reason why a strain visualization can derive meaningful information about the stiffness.

After the argumentation behind a strain visualization, the algorithm to compute it is explained. The algorithm is based on finding the relative displacement of the tissue. Since the interest is on the local relative displacement of the different layers of the tissue, the algorithm is applied to different segments. The approach to manage this is explained in this section. The cross correlation function is used to estimate the relative displacement. For this reason, the concept of the cross correlation function is introduced and analyzed. And finally the latest process to get meaningful displayable values is described.

Motivation

The contrast enhancement visualization (previously introduced) shows information about the property of echogenicity. The motivation of adding another visualization is to give options to the physicians of viewing different properties of the tissue. During the intervention, it is possible that one visualization does not show certain phenomena during the ablation while in the other it is very clear. The purpose of strain visualization is to derive information about the stiffness of the tissue. The stiffness may vary during the ablation process on different regions of the tissue. Having feedback about this changes is useful for the physicians because it provides information about the progression of a lesion during the ablation.

Stiffness measurement

In order to measure the stiffness a scale is required. Young’s modulus is used as a measure of the stiffness on materials and it is defined as the ration between the Stress ([\textit{Force} \over \textit{Area}]) on certain material and its Strain ([\textit{elongation} \over \textit{length}]). The definition is written in equation 3. In simple words it is the relation between the force applied to certain material and the change...
of shape produced by such force.

\[ Y = \frac{F/A}{\Delta l/l} = \frac{stress}{strain} \]  \hspace{1cm} (3)

The definition of stress is as simple as the ratio of the force applied to certain object and the area where it is applied. The Strain \( \Delta l/l \) is the relative change of shape of an object. Therefore, by Young’s Modulus the stiffness is defined as the relative change of shape of a body due to an external force applied to it.

In medical imaging young’s modulus is computed by applying a known force over a known area and measuring the Strain. E.g. this method is used to obtain the stiffness of liver tissue. The physician applies a force via Acoustic Radiation pulses and then the local displacement of the tissue is measured. Hence, it is possible to obtain the Young’s modulus.

However, this approach cannot easily be used for measuring the stiffness in the heart. The inherent movement of the heart introduces an important challenge on distinguishing from the natural movement of the heart and the actual consequence of the external force. Furthermore, the currently available catheters do not have means to produce a force to push the heart tissue.

The absolute Young’s modulus cannot be obtained because it is not possible to introduce a force. Philips is exploring the possibility of using acoustic waves to generate the force [7]. But today, another approach is being taken. It is possible to measure the strain of the tissue \( \Delta l/l \) and by measuring the changes of strain in time, a measure of stiffness change over time (Young’s modulus change) is obtained.

For the physicians, it does not make a difference to see the absolute stiffness at every unit of time; or to see the relative stiffness over time. What they really need is to see the changes on tissue properties when they are ablating. Therefore, computing the strain over time gives the required feedback about stiffness in the tissue.

The strain is calculated by estimating the changes in shape of a body. Differences in strain over time imply changes on the stiffness of the tissue. Thereby, a visualization of the relative stiffness of the tissue over time is derived. Therefore, having a strain visualization where they can see the relative changes on stiffness of the tissue, fulfills the objective of a stiffness visualization.

The strain is calculated by measuring the local displacement of the tissue. Such displacement is derived by the contractions and relaxations of the heart tissue due to the heart beat. In order to measure the local displacement at certain point in time, a reference is required. Therefore, the local displacement of every input RF line \( V_m \) is calculated taking as a reference the previous RF line \( V_{m-1} \). This process is explained in the following subsection.

4.1 Strain Computation (Local Displacement Estimation)

In this subsection, the process to measure the local displacement of the input RF lines is described. This requires to introduce the cross correlation function and the analysis to understand it and later on implement it.

The Cross Correlation Function

In order to estimate the local displacement of the tissue \( (\Delta l/l) \) between two consecutive RF Lines, Cross Correlation is applied. Cross correlation is a measure of the similarity between two signals shifted with different time lags. The time lags are expressed in number of samples. The cross correlation values of the different time lags derive the cross correlation function. The maximum cross correlation value of this function yields the relative displacement of the subject signal.
For computing each cross correlation value, the signal is shifted a number of positions determined by the lag. The lag can be positive or negative and the sign represents the direction of the shift. The maximum lag is defined as $L_{\text{max}}$ and determines the maximum number of positions that the signal will be shifted (in the positive or in the negative direction). Therefore, the maximum lag $L_{\text{max}}$ corresponds to the maximum detectable relative displacement between to signals. The lags are in the range $-L_{\text{max}} \leq l \leq L_{\text{max}}$. The cross correlation function is described in equation 4. This equation shows the cross correlation function of two signals: $X(n)$ and $Y(n)$.

$$\rho(l) = \frac{1}{N} \sum_{n=0}^{N} \left( X(n) \ast Y(n + l) \right) \quad \text{Where:} \quad (-L_{\text{max}} \leq l \leq L_{\text{max}}) \quad (4)$$

In this equation, it is understood that:

$$Y(n + l) = \begin{cases} 
0 & n + l < 0 \\
0 & n + l \geq N 
\end{cases}$$

Every lag $l$ is an index of the cross correlation function $\rho(l)$ which represents the cross correlation value when the signal is shifted $l$ positions. The index of the maximum cross correlation value in the function is the lag when the two signals are more similar. In order to find the lag of the maximum cross correlation value equation 5 is used.

$$V_{\text{dis}} = \text{argmax}(\rho(l)) \quad (5)$$

From the lag that corresponds to the maximum cross correlation value we can interpret the relative displacement of the signal. The interpretation of the lags when cross correlation is applied to estimate $\Delta l/l$ of the tissue is depicted in figure 12.

$$\text{Relative Displacement} \begin{cases} 
\text{Downwards Displacement} & l < 0 \\
\text{No Displacement} & l = 0 \\
\text{Upwards Displacement} & l > 0 
\end{cases} \quad (6)$$

**Cross Correlation Functions of segmented RF lines**

The relative displacement is calculated over the vectors $V_m$ and $V_{m-1}$ when $m > 0$. These vectors represent the subject RF line to compute the strain and the reference RF line. Nevertheless, the requirement is not one global displacement estimation but multiple local displacement estimations. That is because the main purpose is to account for movement on the different layers in the tissue. Therefore, segmentation is required. This implies that multiple cross correlation functions must be computed for each RF line.

The segment length is defined to estimate the displacement of any layer on heart’s tissue. The segment size is expressed in number of samples which represent a number of positions of the input vector $V_m$. The Cross Correlation function is applied to each of the segments. The space between segments is referred to as the step between segments. The step size is defined as $s$ and it is expressed in number of samples. The segment size is defined as $w$. Provided that the size of $V_m$ is $N$, the number of segments to considered is $I = N/s$ when $w \leq s$ and $I = (N - w)/s + 1$ when $w > s$. 
Equation (7) shows the formula to compute the cross correlation function over the different segments. For each cross correlation function to be computed, \( w \) values are taken from \( V_m \) and \( V_{m-1} \). As shown in Equation (7), the positions of values taken from the reference \( V_{m-1} \) depend on the lag. That is how the cross correlation values are calculated between the signal and the reference shifted at different lags. The maximum lag \( L_{max} \) is limited by the size \( w \): \( 0 < L_{max} < \frac{w}{2} \).

\[
\rho_i(l) = \sum_{n=1}^{w-1+i+s} (V_{m-1}(n+l) * V_m(n)) \quad \text{where} \quad \begin{cases} 
(0 \leq i < I) \\
(-L_{max} \leq l \leq L_{max}) \\
(m > 0)
\end{cases}
\] (7)

In this equation, it is understood that:

\[
V_m(n+l) = \begin{cases} 
0 & n+l < 0 \\
0 & n+l \geq N
\end{cases}
\]

The number of elements of the cross correlation function is the number of time lags to shift the signals. There can be positive as well as negative shifts, as shown in equation 7 where it is shown that the lags are in the range \(-L_{max} \leq l \leq L_{max}\). Hence, the number of elements of the cross correlation function is \( L = 2 \times L_{max} + 1 \). The maximum lag \( L_{max} \) corresponds to the maximum detectable relative displacement between to consecutive RF Lines.

For each segment considered, the time lag \( l \) that derives the maximum cross correlation value represent the local relative displacement of the tissue on that segment. The Equation 8 finds the maximum cross correlation value per segment and produces the array \( V_{dis}(i) \) which contains the local relative displacement.

\[
V_{dis}(i) = \arg\max_i(\rho_i(l))
\] (8)

For every segment \( i \) in \( V_m \) the relative displacement with respect to \( V_{m-1} \) is stored in \( V_{dis}(i) \). The estimated displacement is in the range of the lags \( l \) which is \( L_{max} \leq l \leq L_{max} \). The values in \( V_{dis}(i) \) can be normalized in order to have a more meaningful scale of the relative displacement. The range is changed from \( L_{max} \leq V_{dis}(i) \leq L_{max} \) to a range \( 0 \leq V_{dis}(i) \leq 1 \). E.g. when for certain segment \( i \), the lag is \( l = 0 \) then \( V_{dis}(i) = 0.5 \). With the new range, the values can easily be changed to a luminance scale where the displacement can be seen. No displacement or relatively small displacement will yield a gray image; upwards displacement will turn the image darker and downwards displacement will turn the image lighter. An example of a Strain Image is shown in figure 13. The reader can see a luminance image that indicated the local displacement of the tissue at different times.
5 Motivation for GPGPU Implementation

This section is intended for providing reasons for implementing the Philip’s visualization tool in a GPGPU. For this, the main advantages and disadvantages of GPGPUs in comparison with single core CPU, multi-core CPU and FPGAs are explained.

The general purpose GPUs are designed to process large data sets. GPGPUs exploit the parallelization level of the algorithms using hundreds of processing cores. Implementations on a GPGPU have optimal performance when the algorithms require processing over the elements of a large data set without any data dependency. But GPGPU architectures allow the contribution of blocks of threads for processing segments with data dependencies. Therefore, implementations on a GPGPU of algorithms with certain data dependencies may also be optimal with an efficient usage of GPGPU resources.

Algorithms with multiple accesses to RAM memory represent a bigger challenge for a GPGPU implementation. That is because of the high latency of global memory access. This drawback can be partly solved by the usage of cache memory within the multiprocessors of the GPGPUs and by the usage of hundreds or thousands or threads that hide the latency of the global memory accesses. Thereby, algorithms with multiple accesses to RAM memory with low computation power requirements are not suitable for GPGPUs. Only when the algorithms are sufficiently challenging on computation power requirements then the GPGPU implementations can overcome the high latency of memory access.

Sequential implementations are better than a GPGPU implementation when the data dependencies are high and the computation power require is not high enough. An implementation on a GPGPU for such type of algorithm requires redundant operations for avoiding the data dependencies and let all the GPU cores work in parallel. Therefore with a low computation power demand, the GPGPU cannot compensate for the redundant operations made and a sequential implementation is better. Furthermore, the CPU architectures are optimized for branching controlling and can perform better with algorithms requiring many conditions. On the other hand,
the GPGPU architectures execute instructions in groups of 32 threads which can be executed in one cycle in later architectures, however, if conditions in the algorithm require each of the 32 threads to execute different code then it would take 32 cycles to execute 32 threads. Hence, GPGPU is not suitable for algorithms with high branching requirements.

The main advantage of GPGPUs in comparison with multi-core implementations is scalability. Multi-core implementations usually split data, process or both into the number of cores available. However, if the quantity of data increases the number of cores should also increase for keeping the same performance. This requires modifications on the implementation according to the number of cores. It is the same case if more or new algorithms are included. On the GPGPU implementations, the code can be written in such a way that the program is enabled to detect the resources available of the GPGPU and to launch the GPGPU functions to exploit as much as possible the GPGPU resources. When a more powerful GPGPU is required, it can simply be replaced and the program will simply use more resources. E.g. more threads can be released when a GPGPU with more available cores is installed.

Another advantage of the GPGPUs over the multi-core CPUs is on performance due to the number of cores available. The GPGPUs use hundreds of cores and have a sophisticated cache memory architecture that allows for faster computations over large data sets. Furthermore, the different cores within a GPGPU multiprocessor can communicate via shared memory. Multi-core CPUs are limited in the number of cores because the algorithm implemented must somehow be split to utilize the different cores. Therefore, the algorithm should either have many independent steps easily parallelized or large data sets without data dependencies to be split over the cores. Besides, power consumption and price for a multi-core CPU with more than 8 cores may be too high.

Mainly, the advantage of the CPUs is the simplicity of the implementations. A GPGPU efficient implementation requires the developer to understand about the GPGPU architecture. This creates important learning overhead before a GPGPU implementation is fruitful. An inefficient GPGPU implementation can easily be slower and a sequential CPU implementation.

The tradeoff for deciding a processing unit may be over implementation simplicity and performance. In that case a CPU implementation is easier to design than a GPGPU implementation, but the performance on a GPGPU (depending on the algorithm) can be much better. The same comparison can be done between FPGAs and GPGPUs. However, in this case FPGAs can have a better performance but the implementation is harder to design.

In [8], [9] and [10] the reader can find benchmarks concerning performance in number of GFLOPS and power consumption comparing single core CPU, multi-core CPU, GPGPU and FPGA implementations. In the end the same conclusions are given: implementation on FPGA would have a better performance and power consumption with the drawbacks of expensive and limited scalability [9] [8]; and difficult debugging and long waits for synthesis [10]. Thereby, the main advantage of the GPGPU is that it remains with good performance even for very large data sets the the design of the implementation is not as challenging as for the FPGAs.

In conclusion, an implementation on a GPGPU is recommended for algorithms on large data sets with high levels of parallelization. In the remainder of the section, it will be explained the reason for using a GPGPU for the Philip’s visualization tool.

Let’s start with the input rate and the size of the data sets. Every second the visualization tool receives 1000 RF lines of 8Kb per transducer and since we consider 4 transducers the data to be processed per second is 31.25Kb (in 4000 RF lines with 4096 shorts each).

The data set goes through a number of processing steps. Most of these steps are applied element by element independently e.g. gamma correction and contrast enhancement. There are two steps that contain data dependencies but with high computation power requirements. These are envelope detection which requires per RF LINE a FIR filter and the cross correlation
function on several segments of the input vector. These processes implies a considerably large amount of FLOPS which makes them suitable for a GPGPU implementation.

The high level of parallelization and the large data sets encourage the exploration of a parallel implementation. It was decided to explore the GPGPU over the multi-core option because it is expected that the computation power demand is high enough to present a challenge for a GPGPU and it is my contribution to implement, test and benchmark the visualization tool algorithms on a GPGPU.
6 Strain Visualization Implementation

In this section the reader finds the implementation of the strain visualization algorithm. In Subsection 6.1 the implementation in CPU is explained. The algorithm is analyzed in order to reduce the number of computations done on the CPU implementation. In subsection 6.2, the implementation on a GPGPU is explained along with the decisions taken in order to get an efficient implementation.

6.1 CPU Implementation

This subsection shows the work done for implementing the strain visualization algorithm on a single core CPU. The implementation is developed in steps. Each step is an optimization of the previous one being the first step the Naive version. The different optimization steps are explained along with a comparison between them in terms of number of computations and time consumption. While the scope of the thesis is not focused on the implementation on CPU, it is important to understand the implementation in order to compare it with the GPGPU implementation.

The CPU implementation covers Equations (7) and (8). These Equations are rewritten in Equations (9) and (10). In the first (Naive) implementation the formulas are directly implemented without any modification. The second implementation is an optimization of the naive implementation. In this first optimization, the algorithm is described in a different way in order to reduce the multiplications required to compute the strain visualization algorithm. Finally, in the third and last implementation, the algorithm is described in such way that the number of additions required is reduced in addition to the previously reduced multiplications.

\[
\rho_i(l) = \sum_{n=0}^{w-1+iws} (V_{m-1}(n+l) * V_m(n)) \quad \text{where} \quad \left\{ \begin{array}{l}
(0 \leq i < I) \\
(-L_{\text{max}} \leq l \leq L_{\text{max}}) \\
(m > 0)
\end{array} \right.
\]

(9)

In this equation, it is understood that:

\[
V_m(n+l) = \left\{ \begin{array}{ll}
0 & n+l < 0 \\
0 & n+l \geq N
\end{array} \right.
\]

(10)

6.1.1 Naive Implementation

The naive implementation of the strain visualization algorithm is a straight forward codification of the algorithm, directly implemented as shown in the pseudo-code in Figure 14. One cross correlation function must be calculated for every segment \( i \), that is covered by the for loop in line 22. The number of cross correlation values per function is determined by the maximum lag. This is shown in the for loop in line 15 where the range of \( l \) is shown as: \(-L_{\text{max}} \leq l \leq L_{\text{max}}\). Finally, every cross correlation value is the summation of \( w \) multiplications of \( V_{m-1}(n+l) * V_m(n) \). This is implemented in lines 20 and 21 with a for loop. In lines 18 and 19 the variables \textit{init} and \textit{end} are computed, these two variables are the offset on the input vector \( V_m \). This two variables also skip the multiplications when \( V_m(n+l) < 0 \) or \( V_m(n+l) > N \) which according to Equation (9), produce a zero. Furthermore, according to Equation (10) an extra loop for finding the maximum value would be required. However, it is possible to simply keep track of the maximum Cross Correlation value and avoid further operations. This is shown in lines 22 through 25 in the naive pseudocode.
Figure 14: Naive Pseudocode for Cross Correlation

```plaintext
w = 64; // Segment Size
s = 4;  // Step Size
N = 4096 // Vector Size
I = (N-w)/s +1; // Number of segments
MaxLag = 20; // Maximum Lag

float MaxCorr;
int nOutputPos;
float tmp;
int init;
int end;
int nOffset = 0;
for(int nl=0; nl < I; nl++)
{
    MaxCorr = 0;
    for(int nL=MaxLag; nL>=-MaxLag; nL--)
    {
        tmp = 0;
        init = nOffset + max(0, -nL - nOffset);
        end = min(nOffset + w, N - nL);
        for(int nJ = init; nJ < end; nJ++)
            tmp += Vm[nJ] * Vm-1[nJ+nL];
        if(tmp>MaxCorr)
            nOutputPos = nL;
        MaxCorr = tmp;
    }
    apdbOutput[nI] = (double)(nOutputPos)/(2.0*MaxLag);
    nOffset+=s;
}
```

25
In the naive implementation the number of multiplications is shown in Equation (11). The term \( ILw \) comes from the loops required for every segment \( i \), every lag \( l \) and each value of the segment \( w \); the term \( \sum_{i=1}^{L_{\text{max}}/s}(i \ast s) \) describes the multiplications skipped. However, in a normal implementation where \( N = 4096, w = 64 \) and \( L_{\text{max}} = 20 \) the skipped multiplications is insignificant (e.g. \( ILw = 2647616 \) and \( \sum_{i=1}^{L_{\text{max}}/s}(i \ast s) = 60 \)). Therefore, the Equation (11) can be re-written into Equation (12).

\[
\# \text{Mul}_{\text{Naive}} = ILw - 2 \sum_{i=1}^{L_{\text{max}}/s}(i \ast s) \quad (11)
\]

\[
\# \text{Mul}_{\text{Naive}} = \left(\frac{N}{s} + 1 - \frac{w}{s}\right) wL \quad (12)
\]

It is compulsory to detect the lag \( l \) that produces the maximum cross correlation value for every segment \( i \). However, it is not necessary to compute the cross correlation function of every segment independently because the information of the entire vector is known beforehand. Taking advantage of this fact and considering in a normal context the segments are overlapping, a reduction on the number of operations can easily be done. That is shown in the following subsubsection.

### 6.1.2 First Optimization: Avoid Redundant Multiplications

Reducing multiplications is the first optimization done for the CPU implementation. The reasoning comes from the overlapping segments when \( w > s \) that according to Equation (9), produce multiplications over the same values to be repeated. Therefore, there are redundant multiplications which can be avoided.

The new approach is multiply the input vectors \( V_m \) and \( V_{m-1} \) without segmentation. As presented in Equation (13), a matrix defined as \( T_l(n) \) is computed. This matrix contains the \( N \) multiplications of \( V_m(n) \) and \( V_{m-1}(n+l) \) at every lag \( l \).

\[
T_l(n) = V_{m-1}(n+l) \ast V_m(n) \quad \text{Where} \quad \begin{cases} 
-L_{\text{max}} \leq l \leq L_{\text{max}} \\
(m > 0) 
\end{cases} \quad (13)
\]

In this Equation, it is understood that:

\[
V_m(n+l) = \begin{cases} 
0 & n+l < 0 \\
0 & n+l \geq N 
\end{cases}
\]

As shown in Equation (14), the cross correlation function is computed for each segment \( i \) by summing the corresponding values from \( T_l(n) \). Equation (14) is a simplification of Equation ((9)). The multiplications \( V_{m-1}(n+l) \ast V_m(n) \) are substituted by the matrix \( T_l(n) \), provided that all the multiplications required are computed and stored in \( T_l(n) \).

\[
\rho_i(l) = \sum_{n=L_{\text{max}}}^{L_{\text{max}}} (T_l(n)) \quad \text{Where} \quad \begin{cases} 
0 \leq i < l \\
-L_{\text{max}} \leq l \leq L_{\text{max}} \\
(m > 0) 
\end{cases} \quad (14)
\]

The implementation of equations (13) and (14) is described in the pseudocode on figure 15. According to Equation (13), the matrix \( T_l \) stores the multiplications for every lag \( l \). However, in this implementation the outer loop shown in line 8 operates on the different lags and at lag \( l+1 \) the values of \( T_l \) from the previous lag are not required anymore. Therefore, the matrix \( T_l \)
Figure 15: Pseudocode: Multiplication Reduction

```plaintext
w = 64; // Segment Size
s = 4; // Step Size
N = 4096 // Vector Size
I = (N-w)/s +1; // Number of segments
MaxLag = 20; // Maximum Lag
n = 0;
nMaxCorr[I];
for (int nL=-L_max; nL<=L_max; nL++)
{
    nMaxCorr[I]=0;
    init = max(0,-nL); // Avoid Negative Index when lag<0
    end = min(N, N - nL); // Avoid Out-of-bound Index when lag>0
    for (int nJ = init; nJ<N; nJ++)
    {
        T[l][nJ] = Vm−1[nJ+l]*Vm[nJ];
    }
    tmp=0;
    nI = 0;
    nOffset = 0;
    for (int nI=0; nI<I; nI++)
    {
        init = max(nOffset,-nL - nOffset); // Avoid Negative Index when lag<0
        limit = min(nOffset + w, N - nL); // Avoid Out-of-bound Index when lag>0
        for (int nJ = init; nJ<limit; nJ++)
        {
            tmp += T[l][nJ];
        }
        if(tmp>nMaxCorr[nI])
        {
            V_Displacement[nI] = nL;
            nMaxCorr[nI] = tmp;
        }
        nOffset += s;
    }
}
```
is represented by a one-dimension array with \( N \) positions that is re-written at every lag. The computation of \( T_l \) is shown in lines 11 through 16. Once \( T_l \) is computed for a lag \( l \), the cross correlation value for such a lag is computed for every segment \( i \). This is shown in the lines 20 through 27. The maximum cross correlation value for every segment \( i \) is kept in an array with \( I \) positions as shown in lines 28 through 32.

This approach reduces the number of multiplications to \( \# Mul_{Op1} = NL \). The ratio between the number of multiplications on the naive implementation and the multiplication reduced implementation is shown in Equation (15).

\[
\frac{\# Mul_{Naive}}{\# Mul_{Op1}} = \frac{(\frac{N}{s} + 1 - \frac{w}{s}) wL}{NL}
\]

In order to simplify the ratio we can make two assumptions based on a normal configuration of the system. An example of such normal configuration is shown in left hand Table 3. In the right hand table, the number of multiplications required with this set of values is shown for both i) the naive implementation and ii) the first optimization (multiplication reduced). As the reader can see, the vector size \( N \) is considerably larger than the segment size \( w \). And the step size \( s \) is smaller than the segment size \( w \) in any sensible configuration, implying overlapping segments in normal configuration. Hence, the two assumptions are introduced: i) \( w > s \) and ii) \( N \gg w \).

These two assumptions allow to simplify Equation (15) into Equation (16). Thus, we can conclude the number of redundant multiplications depends on the ratio between the segment size \( w \) and the step size \( s \). In Equation (17) it is shown that when \( w = s \) then there is no difference between the naive and the multiplication reduced implementations. On the other hand, Equation (17) shows when \( s = 1 \) the difference on multiplications is the highest.

\[
\frac{\# Mul_{Naive}}{\# Mul_{Op1}} = \left( \frac{w}{s} + \frac{w}{N} - \frac{w^2}{s} \right)
\]

Considering \( N \gg w \) then:

\[
\frac{\# Mul_{Naive}}{\# Mul_{Op1}} \approx \frac{w}{s}
\]

\[
\frac{\# Mul_{Naive}}{\# Mul_{Op1}} = \begin{cases} 
1 & s = w \\
\frac{w}{s} & s = 1 
\end{cases}
\]

A normal value for the step size \( s \) is 4 and for the segment size \( w \) is 64 (see Table 3). Therefore, the naive implementation has around 16 (\( \frac{w}{s} \)) times more multiplications than the first optimization (reduced multiplication). The number of additions remains the same in both implementations. Another optimization can be applied to reduce the number of additions computed. This is covered in the following subsection.
6.1.3 Second Optimization: Reducing Additions

In this subsection it is shown how to reduce the number of additions required by the first optimization. The number of additions can be reduced because Equation 14) implies redundant additions that can be omitted when \( w > s \) (Same case as it happens with the multiplications). This optimization is applied over the first optimization. As stated in the first optimization, \( w \) values from \( T_l(n) \) are summed together to produce a cross correlation value. The additions from segment to segment share \( w - s \) values. Only the first \( s \) positions of every segment are not used on the following segment and then the subsequent \( w - s \) values can be used for the next segment. Therefore, by storing the result of the sum of the \( w - s \) values, it can be reused in the following segment and the redundant additions are reduced.

Storing a temporal sum result required a new variable \( \text{body}_i \) which stores the result of the sums of \( w - s \) values for the \( i_{th} \) segment. Then, Equation 14) can be rewritten as shown in Equation 18. The value for the first segment \( \text{body}_0 \) sums \( w - s \) values starting at index \( s \) and \( \rho_l(0) \) is defined as the sum of \( \text{body}_0 \) plus the first \( s \) values of \( T_l(n) \). When \( i > 0 \), the value \( \text{body}_{i-1} \) (corresponding to the previous segment of \( i \)) can be used to calculate the segment \( i \) without computing \( w - s \) additions again. Then \( \rho_l(i) \) is defined as \( \text{body}_{i-1} \) plus \( s \) values of \( T_l(n) \). This is possible by sequentially updating \( \text{body}_i \) after every segment \( i \) is computed. The update consists of adding to \( \text{body}_i \) the \( s \) new values used for calculating \( \rho_l(i) \), and subtracting the \( s \) values not required for \( \rho_l(i+1) \).

\[
\rho_l(i) = \begin{cases} 
\text{body}_i + \sum_{n=0}^{s-1} (T_l(n)) & i = 0 \\
\text{body}_{i-1} + \sum_{n=(i-1)s+w}^{(i-1)s+w+s-1} (T_l(n)) & i > 0 
\end{cases} \tag{18}
\]

Where:

\[
\text{body}_i = \begin{cases} 
\sum_{n=s}^{w-1} (T_l(n)) & i = 0 \\
\text{body}_{i-1} + \sum_{n=(i-1)s+w}^{i(s+w-1)} (T_l(n)) - \sum_{n=i}s (T_l(n)) & i > 0 
\end{cases}
\]

The implementation of Equation 18 is shown in the pseudo-code in figure 16. In this pseudocode, the reader can see the computation of \( T_l(n) \) is the same as the first optimization. In the lines 17 through 31 is implemented the computation of \( \text{body}_0 \) and \( \rho_l(0) \). The loop in line 33 implements the sequential computation of the cross correlation values on the different segments \( i \) when \( i > 0 \).

The number of sums required with the optimization is \#Sum\text{Op2} = N + s(I - 1) + (I - 1). The \( N \) additions are because all the elements in \( T_l(n) \) must be added at least once. Then, on the segments where \( i > 0 \) \((I - 1)\), \( s \) extra sums and one substraction must be done for maintaining the value \( \text{body}_i \) updated. In Equation 19 it is described the ratio between the sums required on the naive implementation and the sums required on the second optimization. In this Equation it is considered \( N = (I - 1)s + w \).

\[
\frac{\#Mul_{\text{Naive}}}{\#Mul_{\text{Op2}}} = \frac{IwL}{(I)(2s + 1) + w - 2s - 1} \tag{19}
\]

For summarizing the section dedicated to the CPU it is shown in Table 4 the comparison in number of operations of the three different implementations along with the time consumed to
Figure 16: Pseudocode: Addition Reduction

```
1 w = 64;  // Segment Size
2 s = 4;  // Step Size
3 N = 4096  // Vector Size
4 I = (N-w)/s +1;  // Number of segments
5 L_max = 20;  // Maximum Lag
6 dbMaxCorr [1];
7 Tl[N];
8 for(int nL=-L_max; nL>=L_max; nL++)
9 {
10    init = max(0 , -nL);
11    end = min(N, N - nL);
12    for(int nJ = init; nJ < end; nJ++)
13    {
14        Tl[nJ] = V_m[nJ] * V_m-1[nJ+nL];
15    }
16    tmp;
17    nOffset = w;
18    body = 0;
19    for(int nJ=m_nStepSize; nJ<m_nWindowSize; nJ++)
20    {
21        body+=Tl[nJ];
22    }
23    tmp = body;
24    for(int nJ=0; nJ<m_nStepSize; nJ++)
25    {
26        tmp+=Tl[nJ];
27    }
28    if(tmp>dbMaxCorr[0])
29    {
30        V_Dispacement[0] = nL;
31        dbMaxCorr[0] = tmp;
32    }
33    for(int nI=1; nI<I; nI++)
34    {
35        end = nOffset + s;
36        for(int nJ=nOffset; nJ<end; nJ++)
37        {
38            body +=Tl[nJ];
39        }
40        if(body>dbMaxCorr[nI])
41        {
42            V_Dispacement[nI] = nL;
43            dbMaxCorr[nI] = body;
44        }
45        for(int nJ=nI*s; nJ<nI*s+s; nJ++)
46        {
47            body -= Tl[nJ];
48        }
49        nOffset+=s;
50    }
```
Table 4: Comparison of computations and time consumed on the three implementations (Sections 6.1.1, 6.1.2 and 6.1.3) on a Intel Xeon CPU (3.33GHz)

<table>
<thead>
<tr>
<th>Operation Type</th>
<th>Implementations</th>
<th>Naive</th>
<th>1st Opt</th>
<th>2nd Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplications</td>
<td>2,647,556</td>
<td>167,936</td>
<td>167,936</td>
<td></td>
</tr>
<tr>
<td>Sums</td>
<td>2,647,556</td>
<td>2,647,556</td>
<td>372321</td>
<td></td>
</tr>
<tr>
<td>ConsumedTime(4000)</td>
<td>11.3s</td>
<td>5.9s</td>
<td>2.1s</td>
<td></td>
</tr>
</tbody>
</table>

computed them. This results are done using the configuration of Table 3. The time consumed corresponds to a single thread implementation on a Intel Xeon CPU which runs at 3.33 GHz.

Now the reader have a feeling on the number of computations required by an implementation on CPU and how fast it can be executed. The purpose of this is to have a comparison point where to look before the implementation on a GPU is explained. In the next section the details of the GPGPU implementation are explained.
6.2 GPGPU Implementation

In this section, the GPGPU implementation of the strain visualization algorithm is presented. However, before getting to the GPGPU implementation specifically, it is important to understand the peculiarities that GPGPU implementations raise. In Section 6.2.1, it is presented an overview of specific properties to take into account when developing software on top of GPGPUs. These characteristics are related to the multicore architecture of the GPGPUs and its resource allocation properties. Once the context of GPGPU implementation is set, the different steps and decisions, taken for developing an efficient GPGPU implementation of the strain visualization will be explained. A subsubsection is dedicated to each step of the development of the implementation. First in section 6.2.2 the reader will find a very straightforward implementation (naive) which implements the algorithm without further consideration on the resources of the GPGPU nor the utilization of the GPGPU. In the rest of the subsections, each subsection is about one implementation decision, its motivation and the implementation procedure itself. For example, section 6.2.4 covers the modification on the implementation to reduce the number of multiplications, similar to that in Section 6.1.2 where this is explained for the CPU implementation.

At the end of this section the reader will find a general comparison of the different implementation steps. The comparison is judge by the time taken on processing the strain visualization for 4000 input vectors, \( V_m(n) \) for \( 0 \leq m < 4000 \).

6.2.1 GPGPU Implementation Properties

There are new parameters to take into account when working implementations GPGPUS. Those parameters will be introduced in this section, but the reader may find a reference and explanation of these parameters in Table 5. The power of GPGPUs comes from the ability of executing at the same time many threads (up to hundreds). This is possible because of the number of cores a GPGPU has. Furthermore, a GPGPU can allocate thousands of threads (21,504 threads in a Tesla C2075) and a hardware scheduler selects the threads to be executed by the available cores (448 cores in a Tesla C2075). All the threads execute the same code, GPGPU each of them has a individual ID which can be used to work on different data. Therefore, GPGPU holds a Single Instruction Multiple Data approach (SIMD). One GPGPU function is written to be executed for all the threads released, such function is called kernel. Calling this function requires a special function call which is known as kernel call.

Each kernel call requires two important parameters i) the number of threads that will be released on each GPGPU call and ii) how those threads will be organized in the GPGPU blocks. GPGPU blocks are groups of threads that cooperate sharing memory and synchronizing execution. Each thread within a block have its own registers and executes independently from other threads within the block and within the GPGPU device. On the other hand, all threads within a block can access the same piece of shared memory and all the threads on the GPGPU device can access global memory.

Therefore, for every kernel call (GPGPU function call) two parameters are compulsory: The number of blocks to be released \( \#Blocks \) and the number of threads on each block \( \#Threads \). These parameters are sent in square brackets \( << \) param1, param2 \( >>> \) right before the round brackets where normally the parameters are sent (param1, param2, param3). An example of a kernel call is shown below:

\[
\text{kernelDummyCall} \left( \left<< \#Blocks, \#Threads \right>> \right) \left( DOutput, DInput, prm1, prm2 \right);
\]

A third parameter may also be sent for indicating the amount of shared memory to be al-
Table 5: References to used notation

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
<th>Remarks</th>
<th>Typical Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>#MP</td>
<td>It is the number of Multiprocessors in the GPU Device</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>#MAXThreads</td>
<td>It is maximum number of threads on flight in a Multiprocessors</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>#MAXThreadsblock</td>
<td>It is maximum number of threads that can be assigned to a block</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>#Blocks</td>
<td>Number of Blocks used on a Kernel Call</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>#Threads</td>
<td>Number of Threads in a Block</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>SMem_{MP}</td>
<td>Amount of Shared Memory per MultiProcessor</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>SMem_{Total}</td>
<td>Amount of Shared Memory on the GPU Device</td>
<td>It is defined as $SMem_{Total} = SMem_{MP} \times #MP$</td>
<td>48K</td>
</tr>
<tr>
<td>SMem_{Block}</td>
<td>Amount of Shared Memory per Block</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>overlap</td>
<td>It is the region needed by more than one block because of the $Lag_{Max}$</td>
<td>It is defined as $overlap = Lag_{max}$</td>
<td>20</td>
</tr>
<tr>
<td>SMA_A</td>
<td>Shared Memory, it contains a segment of $V_m$</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>SMA_B</td>
<td>Shared Memory, it contains a segment of $V_{m-1}$</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Tmp_{size}</td>
<td>The number of input values needed per Vector in every block</td>
<td>It is the size of Arrays SMA_A and SMA_B</td>
<td>N/A</td>
</tr>
<tr>
<td>#XCval_{block}</td>
<td>Number of Cross Correlation Values per block</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>MaxXC</td>
<td>Variable that stores the current biggest Cross Correlation value</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>
located per Block. However, this parameter is optional.

GPGPU Architecture
Some characteristics of the GPGPU architectures will be briefly introduced in this space.

The GPGPUs are comprised by a number of Multiprocessors (MP). Each MP has a limited amount of resources and a number of processing cores depending on the capability of the GPGPU. The GPGPU architecture allows each MP to divide its resources at most to 8 blocks at the same time. Each MP can hold a limited number of threads, has a limited number of shared memory and a limited number of registers. The most recent architectures allow at maximum 1024 threads per block ($\#\text{MAXThreads}_{\text{block}}$) while the maximum number of threads is 2048 ($\#\text{MAXThreads}$). In other words, at least 2 blocks with 1024 threads must be released to achieve full capacity. For more information significant to the GPGPU resources the reader can refer to [11].

The programming language used to write code for Nvidia GPGPUS is CUDA C. This is a very flexible language that provides the GPGPU programmer with tools to use the kernel calls without knowing much about the inside architecture of the GPGPU. However, using full capacity of the GPGPU devices is a bigger challenge that requires the programmer to understand more about the resource limitations of the GPGPU he is using. For efficient implementations it is compulsory to understand about the resource limitations of the GPGPU and how are those resources allocated. The reader can refer to [12] for an overview of the GPGPU Architecture and for more detail about the different architectures the reader can refer to [13].

GPGPU Implementation Example
Let’s analyze an example. We have a four Multiprocessor GPGPU device that can hold 6144 threads concurrently (1536 each MP). Each MP has 48Kb of shared memory and 32768 registers. With this GPGPU device we give examples of resource allocations that lead to inefficient use of the capacity of the GPGPU device.

Example:
Program Objective: An array of 10,000 positions is to be processed. The algorithm takes 100 values and produces one; every produced value is independent from the others and use completely different data. The result is a vector with 100 positions.

A) Using too many blocks
A kernel function defined in such way that every block produces 1 value using the cooperating 100 threads. Therefore, a kernel call with 100 Blocks and 100 Threads is necessary.

Only 32 blocks can be hold on flight in the GPGPU device ($8 \times \#\text{MP}$). Since every block uses 100 threads, only 3400 threads are hold on flight in the device when 6144 threads is the maximum capacity. Thus the GPGPU device is working at 55% of its capacity.

The rest of the blocks are kept in a waiting list. When the first 32 blocks are finished then the next 32 blocks will be allocated in the device. An option to avoid this problem is to reduce the number of blocks and let each block produce more than one value. That option is covered in the following point.

B) Using too many threads
Continuing with the example, a way to solve the issue of the number of blocks is increasing the number of threads. The kernel function should be change so each block produces 10 values instead of one, using 10 times more threads. Then the kernel call can be done with 10 blocks and 1000 threads. Each block will use 1000 threads to produce 10 values.

The problem with this context is that each MP can only hold 1536 threads. Since each
blocks has 1000 threads, only one block can be allocated on each MP at the time. Therefore the GPGPU is working at 65% of the capacity. The big idea to solve the problem is to constraint the kernel call to have a configuration that uses properly the resources of the GPGPU device. And at the same time, to have a flexible kernel function that works with any configuration of #Blocks and #Threads.

For efficiency, a kernel call should use a #Blocks in the range \(2 \times \#MP \leq \#Blocks \leq 8 \times \#MP\). The limits are set to 2 and 8 blocks per MP. The reason to have a maximum of 8 blocks is due to the capacity of the GPGPUs. The reason to set the lower limit to 2 blocks is because the number of threads a MP can hold in flight is larger than the maximum number of threads a block can have.

The number of threads should be in the range \(#Threads \leq \lfloor \frac{\#MAXThreads}{\#Blocks} \rfloor\) in order to allocate all the blocks in the MP without having a waiting list. E.g. 32 blocks with 192 threads where every MP allocates 1536 threads.

On the other hand, the kernel function must be able to compute the algorithm with the decided number of blocks and threads. If a configuration with 32 blocks and 192 threads is selected, then the kernel function must use 32 blocks for producing 100 values, using its 192 threads. That step may implies a bigger effort on writing the code for the kernel function. The tradeoff is simplicity on kernel function vs efficiency.

In this point I deliberately omitted two very important limitations: shared memory and registers. Those are covered in the following points.

C) Inefficient Register Allocation

In this point we reuse the implementation discussed in clause B) with 32 blocks and 192 threads.

If we now consider the number of registers we will find an important limitation. Let’s say the kernel function uses 24 registers, then each of the 192 threads need 24 + 1 registers. Therefore, every block would require 4800 register. The device can hold 32768 registers per MP. That means each MP can hold \(\lfloor \frac{32768}{4800} \rfloor = 6\) blocks. Hence, the GPGPU device is working at 75% of its capacity.

A way around the problem is reducing the number of threads. In order to keep 8 blocks, we can calculate the maximum number of threads that can be allocated without running out of registers, that is: \(\lfloor \frac{32768}{25} \rfloor = 163\). A configuration with 8 blocks and 163 threads would yield a 85% usage of the GPGPU. Nevertheless, there is an important property of the GPGPUs that I have not told yet: the warp execution.

A warp is a group of threads that execute in lock step on the GPGPU. That means the threads in a warp execute the same line of code with no fetching in between. In latest architectures, the threads may actually be executed at the same time since MPs have 48 cores. All current technologies work with a warps of 32 threads.

The warp constraint is completely transparent to the CUDA C programmer but it has very real consequences. In the previous example, we found out that 163 threads would be better than using 192. However, because of the warp groups, the actual number of threads released is 192. Therefore, the GPGPU will allocate only 6 blocks per MP and each blocks has 163 working threads plus 29 dummy threads. Yielding not even the 75% usage but it goes down to 63%.

The right way to do it is to count in number of warps. Then the number of threads to released is, \(\left\lfloor \left( \frac{32768}{25} \right) / 32 \right\rfloor \times 32 = 160\). That will release exactly 5 warps and we can be sure the number of registers in the MP will be enough. Therefore, the usage of the GPGPU device
goes up to 83%.

Yet another way to increase the usage is working on the kernel function and limit the number of registers required. In this example, by only reducing the number registers by 4, it is possible to use 192 threads per block and the GPGPU device Usage goes back to 100%.

In the following clause it will be cover the issues coming when shared memory is used.

D) Inefficient Shared Memory Allocation

By now we have a flexible kernel function that can use 32 blocks with 160 threads to compute 100 positions. In this clause the requirement of shared memory is introduced. Certain amount of shared memory is required by the threads within each block to cooperate and produce an output. The input vector has 10,000 positions, which means every block must have 400 positions available in shared memory. Then every MP will hold 3200 positions (8 * 400). If the input values are doubles, then the shared memory required per MP is 25.600bytes. Provided that each MP has 48Kb of shared memory if the input values are doubles, the 83% usage holds.

However, if the input values were a structure type with 4 double values, the story is different. Then, each MP would need 100Kb of shared memory to allocate 8 blocks. Since each MP only has 48Kb this is not possible and only 3 blocks can be allocated per MP. That is 37.5% of the capacity.

A manner to solve this problem is divide the shared memory of each MP over the number of blocks selected. In this case, dividing 48Kb over 8 blocks. Hence, every block has memory to store 192 positions. However, for each output value, 100 input are required. Thereby, at every block, 92 positions in shared memory are not used. Besides, only 100 threads will be required to execute because only 100 positions of memory are available. This means 60 threads are wasted per block. Hence, 32 blocks with 100 threads are used on the kernel call yielding a 52%. This is a big improvement in comparison with the 37.5%.

There is still room for optimization. With 32 blocks and 100 threads there is too much shared memory and too many threads not used. This can be changed by selecting a different number of blocks. If we used 6 blocks per MP instead of 8 we can allocate more shared memory per block and increase the number of threads without going over the number of registers. With 6 blocks, the number of positions per block can be 256. Then, the number of threads increases to 200. Hence, the usage of the GPGPU goes up to 78.1%. The number of registers used per MP is 30,000 which is still below the limit.

Without using shared memory, a configuration of 8 blocks with 192 threads and a configuration of 4 blocks with 384 are equivalent. This is because the blocks and threads are within the limits of the resources of the device. And the number of registers used is still the same. However, when shared memory is used, the number of blocks plays an important role for achieving high usage of the device.

In this example I covered the most common challenges that GPGPU implementation take. Having maximum usage of the GPGPU by analyzing the resource allocation is a key point for developing and I use it during the implementations of this project.

In the remaining of the section each step to reach an optimal GPGPU implementation is explained in a different subsection. At the end of each subsection it will be presented: i) the usage of the GPGPU and ii) the time consumed for executing 4000 input lines \( V_m(n) \). The summary of usage and time consumed of every subsection is in Table 15. The GPGPU device used on the experiments is a QUADRO 600, see [14] for more information about the characteristics of this GPGPU. It provides 2 Multiprocessors, each MP can hold 1536 threads, has 48Kb of shared memory and 32768 registers that can be divided at most to 8 blocks. The GPGPU properties mentioned and the configuration of the strain visualization algorithm used for the benchmark is shown in Table 6.
6.2.2 Naive Implementation

The first implementation in GPGPU is a straight forward approach similar to the naive implementation of the CPU. However, in the GPGPU implementation the local displacement $V_{dis}(i)$ for each segment $i$ is calculated by an independent GPGPU thread. In Figure 17 the reader can see the pseudocode used by every thread released. The number of threads per block and the number of blocks is configured as shown in Equation (20). For the naive version the number of blocks used is simply determined by the number of Multiprocessors on the GPGPU device and the number of threads per block is determined by the number of segments each block computes. Every single thread processes the Cross Correlation function and captures the maximum value therefore computing the local displacement of each segment.

$$
\#Blocks = \#MP
$$

$$
\#Threads = \lceil I/\#Blocks \rceil
$$

With the configuration shown in Table 6, the GPGPU works at 32.84% of its capacity and the time consumed for computing 4000 input vectors is 3.7s. The number of registers required per multiprocessor is 14336 which is within the limits of the GPGPU (32768 registers). This implementation does not require shared memory. The low usage is exclusively due to the low number of threads released which is 1009 since the number of segments $I = 1009$. And the capacity of the GPGPU is 3072 threads in flight. A summary of the resources spent by the naive version is shown in Table 7.

<table>
<thead>
<tr>
<th>GPUProperty</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPUName</td>
<td>Quadro600</td>
</tr>
<tr>
<td>#MP</td>
<td>2</td>
</tr>
<tr>
<td>#MAXThreads</td>
<td>1536</td>
</tr>
<tr>
<td>SMemMP</td>
<td>48Kb</td>
</tr>
<tr>
<td>#RegistersMP</td>
<td>32768</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>4096</td>
</tr>
<tr>
<td>w</td>
<td>64</td>
</tr>
<tr>
<td>s</td>
<td>4</td>
</tr>
<tr>
<td>$L_{max}$</td>
<td>20</td>
</tr>
<tr>
<td>L</td>
<td>41</td>
</tr>
<tr>
<td>I</td>
<td>1009</td>
</tr>
</tbody>
</table>

Table 6: Example of System’s Settings

<table>
<thead>
<tr>
<th>Naive Implementation</th>
<th>TimeConsumed</th>
<th>Usage</th>
<th>SharedMemory</th>
<th>Registers</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.7s</td>
<td>32.84%</td>
<td>0Kb</td>
<td>14336</td>
<td>1009</td>
</tr>
</tbody>
</table>

Table 7: Summary of results and resources (Naive Version)
Figure 17: Pseudocode: GPU Naive

```c
// s = step_size
// w = window_size
XCId = threadIdx + blockIdx*(\#Threads-(I)%\#Blocks));
if(XCId < I){
  offset = threadIdx*s;
  l = L_max;
  nWinSize = w + offset;
  nMaxCorr = 0.0;
  nMaxCorrPos = 0;
  for (l = -L_max; l<L_max; l++)
  {
    tmp=0;
    nInit = max(0,-1 - offset) + offset;
    int end = min(nWinSize, N - 1);
    for (int nl = nInit; nl<end; nl++)
    {
      tmp +=Vm[nl] Vm-1[nl+nLag];
    }
    if(tmp>nMaxCorr){
      nMaxCorrPos = l;
      nMaxCorr = tmp;
    }
    nOutputPos++;
  }
  apdbOutput[XCId] = nMaxCorrPos;
}
```

The idea of showing the naive implementation is simply for understanding purposes. The main algorithm is exactly the same as the CPU Naive version but instead of iterating over the windows, GPGPU processes all of them at the same time with different threads. Although, there are two major flaws in this implementation besides the redundant multiplications. The most important one is the multiple access to GPGPU global memory which has a considerably high latency. The second one is the waste on the GPGPU’s capacity, GPGPUs can have thousands of threads in flight and if the number of windows \( I \) required is small then the full-capacity of GPGPU is not used.

### 6.2.3 First Optimization: Loading on Shared Memory

In this section it is explained the implementation done for avoiding the problem of multiple unnecessary access to GPGPU global memory. The problem on GPGPU global memory access can be easily solved by using shared memory (see [15] for more detail about Global Memory and efficient usage). All the threads within a block can use the same piece of shared memory. Therefore, the information can be first copied to the shared memory of different blocks and then be processed by the threads within that block. Each block will contain a different piece of the input vectors \( V_m \) and \( V_{m-1} \) on its shared memory. However, on the edges of the ranges of the blocks there are overlaps because of the lags.

The vectors \( V_m \) and \( V_{m-1} \) are divided over the number of blocks released. The amount of shared memory per block is defined as \( SMem_{Block} \) and it must store two arrays of shared memory, one for each Input Vector. The vector part of \( V_m \) of the blocks \( bl \) is stored in \( SMA^A_{bl} \) and \( V_{m-1} \) in \( SMA^B_{bl} \) where \( bl \) is the index of the block. Evidently the sum of both shared memory arrays must be less or equal than \( SMem_{Block} \). The number of positions of the arrays and the
shared memory required for each block to store the two shared memory arrays is calculated as shown in Equation 21. The memory allocated in one block should fit in one multiprocessor. In case this condition does not hold, a different number of blocks must be used.

\[ SMA_{\text{size}} = (\lceil I / \# \text{Blocks} \rceil - 1) \times s + 2 \ast \text{overlap} + w \]

\[ SMem_{\text{Block}} = 2 \times SMA_{\text{size}} \times \text{sizeOf(type input)} \]

where, \( SMem_{\text{Block}} \leq SMem_{\text{MP}} \) (21)

In this implementation, the amount of blocks determines the number of divisions of the input data as well as the limit on data every block can store (\( SMem_{\text{Block}} \)). The number of blocks and threads released is calculated according to equation (22). The number of blocks is decided to be only one per multiprocessor for avoiding overlaps when loading the input in Shared Memory. Each block has a number of threads enough to read the assigned positions from input vectors and store the data on the shared memory arrays \( SMA_{B} \) and \( SMA_{B} \).

\[ \# \text{Blocks} = \# \text{MP} \]
\[ \# \text{Threads} = \min(SMA_{\text{size}}, \# \text{MAXThreads}) \] (22)

In Equation 23 it is shown how the Input Vectors are stored in shared memory. There is a different vector \( SMA_{A}^{bl}(th) \) and \( SMA_{B}^{bl}(th) \) for each block \( 0 \leq bl < \# \text{Blocks} \). And each thread \( th \) loads a value from \( V_{m} \) and \( V_{m-1} \) and stores it in \( SMA_{A}^{bl}(th) \) and \( SMA_{B}^{bl}(th) \). The offset between blocks is defined by the number of cross correlation values to be processed by each block \( #XVal_{\text{block}} = \lceil I / \# \text{Blocks} \rceil \) and the step size \( s \). The offset \( bl \times s \times XVal_{\text{block}} \) determines the start of the segment that block \( bl \) uses to compute cross correlation values. However, because of the maximum lag, the start point takes \( L_{\text{max}} \) positions before this point. This is the overlap between blocks as shown in Equation 23.

\[ SMA_{A}^{bl}(th) = V_{m}(th + bl \times s \times XVal_{\text{block}} - \text{overlap}) \]
\[ SMA_{B}^{bl}(th) = V_{m-1}(th + bl \times s \times XVal_{\text{block}} - \text{overlap}) \]

where

\[ 0 \leq bl < \# \text{Blocks} \]
\[ 0 \leq th < \# \text{Threads} \]
\[ \text{overlap} = L_{\text{max}} \]

In this equation, it is understood that:

\[ V_{m}(n) = \begin{cases} 
0 & n < 0 \\
0 & n \geq N 
\end{cases} \]

Once the data is loaded in shared memory, only \( #XVal_{\text{block}} \) threads need to run for computing the Cross Correlation Values (exactly the same as the naive implementation, but this time reading from shared memory), and the rest of the threads \( \# \text{Threads} - XVal_{\text{block}} \) will simply wait.

The result of this implementation is a reduction on the accesses to GPGPU global memory from \( ILw \) to \( N + 2 \ast L_{\text{max}} \ast \# \text{Blocks} \). According to the configuration of Table 6, there is reduction of global memory access of the 99.84%. The consequence of this reduction in memory access is a time consumed of 2.15s which is only 60% of time consumed by the naive implementation.

The size of \( SMem_{A} \) and \( SMem_{B} \) is 2096, and the total shared memory required for block is 16.37Kb thus the condition \( SMem_{\text{Block}} \leq SMem_{\text{MP}} \) holds. The number of threads per block is equal to \( \# \text{MAXThreads}_{\text{block}} \) because \( SMem_{\text{size}} > \# \text{MAXThreads}_{\text{block}} \). Therefore the implementation reuse threads for storing the input from global memory to shared memory. The registers required per block 18,432 are within the limit of the multiprocessors. Despite the 66% of the threads are released and used (2048 threads from the 3072 available), it is not possible to say the usage of the GPGPU is 66% since 520 threads (\( \# \text{Threads} - XVal_{\text{block}} \)) of each block are used only to store in shared memory.
For calculating the GPGPU usage, only the code for data transfer from global to shared memory was run. Then, the complete implementation was run. The data transfer takes much less than 1% of the total time consumed. Therefore we consider the GPGPU usage to stay in 32.84%. A summary of the resources spent by the naive version is shown in Table 8.

<table>
<thead>
<tr>
<th>First Optimization</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TimeConsumed</strong></td>
<td>2.15s</td>
</tr>
<tr>
<td><strong>Usage</strong></td>
<td>32.84%</td>
</tr>
<tr>
<td><strong>SharedMemory</strong></td>
<td>16.37Kb</td>
</tr>
<tr>
<td><strong>Registers</strong></td>
<td>18.432</td>
</tr>
<tr>
<td><strong>Threads</strong></td>
<td>1009</td>
</tr>
</tbody>
</table>

Table 8: Summary of results and resources (First Optimization)

This optimization makes a better use of the resources of the GPGPU but computations done to derive the strain visualization is exactly the same as for the naive implementation. The only difference is the usage of shared memory instead of accessing global memory. The following optimizations are based in changes on the manner the computations are being done but they all use shared memory to access the data as explained in this section. First in section 6.2.4 it is explained how the number of multiplications can be reused. Then in section (sums) it is explained how the optimization of Section 6.2.5 is improved by reducing the number of additions. In section 6.2.6, it is explored how to increase the usage of the GPGPU. For this, the optimizations of computation reduction are not used. Finally, the optimization in section 6.2.6 is improved by reducing additions and that is explained in Section 6.2.7.

### 6.2.4 Second Optimization: Reduce Multiplications

In this section, the second optimization step is covered. This optimization is based on reducing the number of multiplications with the same reasoning applied in subsection 6.1.2. For this, Equation (13) can be rewritten into Equation (24). A matrix similar to $T_l(n)$ used in subsection 6.1.2 is used to store $N$ multiplications for every lag $l$.

However, in the GPGPU implementation, the multiplications are computed reading the values of the input vectors from the shared memory arrays $SMA_A^l$ and $SMA_B^l$. Since every GPGPU block has a different segment of the input vectors, there is a different matrix for each block. This is shown in Equation (24) where $T_{bl}^l(th)$ represents the matrix allocated in shared memory for each block $bl$. The size of $T_{bl}^l(th)$ is $SMA_{size}$.

When the multiplications are computed and stored in $T_l$, the additions are computed as done in the naive version. For this, Equation (14) can be rewritten into Equation (25). A single thread computes the $w$ additions at every lag $l$ for deriving the cross correlation function for one segment $i$. For this, $\#XCval_{block}$ threads per block complete the cross correlation function of all the segments.

After the input vectors are stored in shared memory (as explained in Equation (23), the multiplications element by element of the entire arrays $SMA_A$ and $SMA_B$ are computed and store in a new array $T_l$ in shared memory. According to Equation 24, at every lag $l$ in $-L_{max} \leq l \leq L_{max}$ every thread $th$ will produce a value by multiplying an element of array $SMA_B$ and array $SMA_B$ and store it in shared memory $T_{bl}^l(th)$.

$$T_{bl}^l(th) = SMA_B(th + l + overlap) * SMA_A(th + overlap) \quad \text{where} \quad \begin{cases} 0 \leq th < \#Threads \\ -L_{max} \leq l \leq L_{max} \end{cases}$$

40
To produce the Cross Correlation function only $\#XCval_{block}$ threads add the required values from $T^b_l$ for every lag $l$. According to equation 25, $\#XCval_{block}$ threads of every block will be in charge to add up $w$ positions of $T^b_l$ for each lag.

$$\rho_{in}(l) = \sum_{n=\text{th}}^{w-1+\text{th}} (T^b_l(n)) \quad \text{where} \quad \begin{cases} \text{in} = th + bl \times \#XCval_{block} \\ -L_{\text{max}} \leq l \leq L_{\text{max}} \\ 0 \leq th < \#XCval_{block} \\ 0 \leq bl < \#Blocks \\ n < Tmp_{\text{size}} \end{cases}$$

The lag that yield the largest value for certain position is stored in the $V_{\text{dis}}$ which holds the Relative Displacement between $V_m$ and $V_{m-1}$. For this, Equation (10) can be rewritten into Equation (26), with slight differences caused by the indexing.

$$V_{\text{dis}}(\text{in}) = \text{argmax}(\rho_{in}(l)) \quad \text{where} \quad \begin{cases} \text{in} = th + bl \times \#XCval_{block} \\ 0 \leq th < \#XCval_{block} \\ 0 \leq bl < \#Blocks \end{cases}$$

In the actual implementation, on each segment $i$ the cross correlation values generated for every lag $l$ are compared with a temporal variable that stores the Maximum Value $MaxXC$. The first value that $MaxXC$ stores is $\rho_{in}(0)$ then on the following lags $l$ only a bigger value than $MaxXC$ will replace it. That is also done in CPU implementations.

According to the configuration of Table 6, two blocks are used for the call of this implementation. The size of $S\text{Mem}_A$ and $S\text{Mem}_B$ and $T^b_l$ is 2096. Thereby, the total shared memory required for block is 24.56Kb thus the condition $S\text{Mem}_{\text{Block}} \leq S\text{Mem}_{\text{MP}}$ holds. The number of threads per block does not differ from the first optimization, it is equal to $\#\text{MAXThreads}$ because $S\text{Mem}_{\text{size}} > \#\text{MAXThreads}$. In the first optimization, the code allows the reutilization of threads to cover the $S\text{Mem}_{\text{size}}$ positions of memory. In this second optimization, the threads are also reused for computing the multiplications of the $S\text{Mem}_{\text{size}}$ positions. The registers required per block are within the limit of the multiprocessors 20, 480.

The time consumed after the second optimization is 2.04s, 1.83s for computing the additions and 0.27 for computing the multiplications. In this optimization, 66% of the threads load from global memory and execute the multiplications and 32.84% of the threads compute the additions and find maximum cross correlation value. For calculating the global usage of the GPGPU, first we measure the time consumed only in loading memory and computing the multiplications. That step represents only 1.5% of the total time consumed, during this time the GPGPU works at 66%. The rest 98.5% of the time, the GPGPU works at 32.84% of its capacity. Hence, the usage of the GPGPU is only 33.32%. A summary of the resources spent by this optimization is shown in Table 9. The difference between the first two implementations is small. We can see that the number of multiplications does not play a big role in the GPGPU, on the other hand the usage

<table>
<thead>
<tr>
<th>Second Optimization</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeConsumed</td>
<td>2.04s</td>
</tr>
<tr>
<td>Usage</td>
<td>33.32%</td>
</tr>
<tr>
<td>SharedMemory</td>
<td>24.56Kb</td>
</tr>
<tr>
<td>Registers</td>
<td>20,480</td>
</tr>
<tr>
<td>Threads</td>
<td>1009</td>
</tr>
</tbody>
</table>

Table 9: Summary of results and resources (Second Optimization)
of the GPGPU is very important. Therefore, in subsection 6.2.6 an optimization to increase the utilization of the GPGPU is described. The major issue is that increasing the utilization of the device implies increasing the level of parallelization of the algorithm which means avoidance of any dependency. Therefore, the implementation with reduced multiplications would not be used.

Before analyzing the optimization for increasing GPGPU usage, a last optimization for reducing number of computations is used. The number of additions computed can be decreased in addition to the reduction of multiplications. Such optimization is described in the following subsection.

### 6.2.5 Third Optimization: Reduce number of Additions

In this subsection it will be explained how by using a property of the visualization tool configuration, the number of additions computed can be decreased. This idea origins an implementation which assume the multiplication reduction of the second optimization in subsection 6.2.3. This optimization is based on the fact that $w > s$ but also $\frac{w}{s} \% s = 0$.

This approach introduces a new array in shared memory $tSum(m)$ of size $N/s$ that stores the temporary additions. Equations (27) and (28) show how this approach can be computed. The additions are done in group of $s$ sums and stored in $tSum(m)$, then the cross correlation function is computed by computing $w/s$.

$$tSum(m) = \sum_{n=m \cdot s}^{m \cdot s + s} T_i(n) \quad where \quad \begin{cases} -L_{\max} \leq l \leq L_{\max} \\ 0 \leq m < \frac{N}{s} \end{cases}$$  \quad (27)

$$\rho_i(l) = \sum_{j=i}^{i + \frac{w}{s}} (tSum(n))$$  \quad (28)

The parallel implementation on a GPGPU is driven by Equations (29) and (30). These equations show that an array $tSum^{bl}(th)$ is declared for every block $bl$ and a different thread $th$ executes in parallel $s$ additions. When the batches of additions are computed and stored, the cross correlation function of each segment can be computed using only $w$s additions per lag. The number of additions required for this implementation is shown in Equation (31).

$$tSum^{bl}(th) = \sum_{n=th \cdot s}^{th \cdot s + s} T_i^{bl}(n) \quad where \quad \begin{cases} 0 \leq th < \frac{SMA_{size}}{s} \\ 0 \leq bl < \#Blocks \end{cases}$$  \quad (29)

$$\rho_{in}(l) = \sum_{n=th}^{th + \frac{w}{s}} (tSum^{bl}(n)) \quad where \quad \begin{cases} in = th + bl \cdot \#XCval_{block} \\ 0 \leq th < \#XCval_{block} \\ 0 \leq bl < \#Blocks \end{cases}$$  \quad (30)

$$\#SUMS = L(N + I(\frac{w}{s}))$$  \quad (31)

The comparison in number of additions with the naive version is shown in Equation (32). Evidently, this optimization is not efficient when $s = 1$ and $s = w$. In this cases the number of additions actually is increased. The best addition reduction is when $s \approx \sqrt{w}$. On the other
hand the reduction of multiplications is maximum when \( s = 1 \) and the reduction decreases as \( s \to w \). Therefore, the optimal range for this optimization is when \( 2 \leq s \leq \sqrt{w} \).

\[
\frac{\#\text{Sum}_{Naive}}{\#\text{Sum}_{Opt}} = \left( \frac{IwL}{L ((I - 1)s + w + I \frac{w}{s})} \right)
\]

(32)

Using the configuration of Table 6, the number of additions is decreased 3 times. With this optimization the time is reduced to 0.71s, for computing 4000 thousand input vectors: 0.27s for the multiplications and 0.44s for the additions. The overall time is a third of the time consumed when only the multiplications are reduced. The time consumed only for computing the additions in this optimization is 0.44s which is 4 times less than only reducing the multiplications (1.83s).

The reduction is not only due to the reduction of the additions but also to the reduction of the iterations the threads have to execute. This is, for every lag \( l \) the number of iterations was reduced from 64 to 16.

The number of threads released is 1024 for each block which means 66% of the utilization of the GPGPU. However, for computing the additions not all the threads are used. The number of threads used to compute the batches of sums is 529 and that process takes 0.16s and the number of threads that compute the cross correlation function and determine the maximum value is 504 and that process takes 0.28s. This gives a total usage of 44.43% as shown in Table 10.

<table>
<thead>
<tr>
<th>Step</th>
<th>Time</th>
<th>% of time</th>
<th>GPGPU Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplications &amp;</td>
<td>0.27s</td>
<td>38%</td>
<td>62.5%</td>
</tr>
<tr>
<td>Shared Memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Batch sums</td>
<td>0.16s</td>
<td>22.53%</td>
<td>34.44%</td>
</tr>
<tr>
<td>Cross correlation</td>
<td>0.28s</td>
<td>39.43%</td>
<td>32.81%</td>
</tr>
<tr>
<td>Total</td>
<td>0.71s</td>
<td>100%</td>
<td>44.43%</td>
</tr>
</tbody>
</table>

Table 10: GPGPU Usage Calculation

Since a new array is required to store the temporal additions, the shared memory per block adds \( \frac{SMA_{size}}{s} = 524 \) positions. Thus the shared memory used per block is 26.6Kb. As for the registers, 20,480 registers are required per block. A summary of the resources used in this implementation is in Table 11.

<table>
<thead>
<tr>
<th>Third Optimization</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Consumed</td>
<td>0.71s</td>
</tr>
<tr>
<td>Usage</td>
<td>44.43%</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>26.6Kb</td>
</tr>
<tr>
<td>Registers</td>
<td>20,480</td>
</tr>
<tr>
<td>Threads</td>
<td>1009</td>
</tr>
</tbody>
</table>

Table 11: Summary of results and resources (Third Optimization)

The following optimization does not use any of the computation reductions discussed and focuses more on increasing the parallelization of the algorithm.

6.2.6 Fourth Optimization: Increment GPGPU Usage

This subsection is dedicated to explain the fourth optimization of the implementation of the strain visualization algorithm. This optimization does not apply the reduction of computations
and instead it is focused on increasing the usage of the GPGPU. The approach taken is by computing all the lags required by each segment in parallel.

The implementation uses the segment vectors $SMA_A$ and $SMA_B$ defined as shown in subsection 6.2.3. Since all cross correlation values correspondent to each lag are computed in parallel, the optimizations explained in subsections 6.2.4 and 6.2.5 are not used. Reducing the number of multiplications would imply store the complete matrix $Tl(th)$ and with the normal configuration there is not enough memory storage.

According to Equation (33), every block $bl$ executes a number of segments $J$ where for each $j$ all the lags $l$ are computed in parallel. Each lag $l$ is computed by an independent thread $th$ and stored in the matrix $Lags^{bl}_j(l)$ int shared memory. Thus, every block $bl$ has a matrix in shared memory $Lags^{bl}_j(l)$ of dimensions $JxL$, which contains the cross correlation functions of the segments assigned to the block. The number of segments processed per block depends on the available threads, as shown in Equation (34). By consequence, depending on the segments processed per block is the number of blocks to be released.

$$Lags^{bl}_j(l) = \sum_{n=js}^{js+J} (SMA_B(n+l+\text{overlap}) \times SMA_A(n+\text{overlap}))$$

where

$$\begin{cases} l = (L_{\text{max}} - th\%L) \\ j = \frac{th}{T} \\ 0 \leq th < \#\text{Threads} \\ 0 \leq bl < \#\text{Blocks} \end{cases}$$

(33)

$$J = \frac{\#\text{Threads}}{L}$$

$$\#\text{Blocks} = \frac{1}{J}$$

(34)

In practice, it is only a limited number of blocks that is released. Each block actually covers more than one index $bl$ by iterating over the segments. Once the matrix $Lags^{bl}_j(l)$ is ready for each index $bl$ we can use Equation (35) to find the lag $l$ correspondent to the maximum cross correlation value. Each block derives a range of segments that finally composed the local displacement of the input vector $V_m$.

$$V_{dis}(in) = \text{argmax}(Lags^{bl}_j(l)) \text{ where } \begin{cases} in = bl \times J + j \\ 0 \leq bl < \#\text{Blocks} \end{cases}$$

(35)

The number of multiplications and additions required by this implementation is shown in Equation 36. The group of $JL$ threads compute the $L$ lags of $J$ segments in parallel by executing $w$ multiplications and additions. This is done $\left\lceil \frac{J}{T} \right\rceil$ times for computing all the segments.

$$\#\text{Mul} = \#\text{Sum} = J \times L \times \left\lceil \frac{J}{T} \right\rceil \times w$$

(36)

In this optimization, the number of threads used is considerably larger than previous implementations but the number of multiplications computed is the same of the naive implementation. Using the configuration of Table 6, the number of threads released per block is 1024, all the threads are used for loading global memory into shared memory. But in this implementation, the number of threads used for the computations is 943 per block. That gives a GPGPU usage of 61.4% and the time consumed for processing 4000 input vectors is 1.8s.

The shared memory requires 2096 positions for each array SMAA and SMAB plus the array 943 positions for the matrix $Lags^{bl}_j$. Thus, the shared memory required per block is 20.05Kb. And it its 22,528 registers. The summary of the resources used is in Table 12.
This optimization shows how by increasing the utilization yields better performance than decreasing the number of computations to be executed. The time consumed for processing 4000 reducing multiplications 16 times is 2.1s and by doubling the GPGPU usage the time consumed is 1.46s. However, if we compare this optimization with the third optimization (reduction of additions and multiplications), the story is different. The third optimization reduces by 16 times the number of cycles done for computing every cross correlation value and that makes it twice faster than an implementation increasing the usage. That implies that reducing the number of cycles of the threads and incrementing the number of effective threads pays better than only reducing the load of work.

Therefore, in the next optimization the reduction of additions will be implemented to this optimization.

<table>
<thead>
<tr>
<th>fourth Optimization</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeConsumed</td>
<td>1.46s</td>
</tr>
<tr>
<td>Usage</td>
<td>61.4%</td>
</tr>
<tr>
<td>SharedMemory</td>
<td>20.05Kb</td>
</tr>
<tr>
<td>Registers</td>
<td>22,528</td>
</tr>
<tr>
<td>Threads</td>
<td>1886</td>
</tr>
</tbody>
</table>

Table 12: Summary of results and resources (Fourth Optimization)

6.2.7 Fifth Optimization: Increment GPGPU Usage and Reduce Operations (Cycles)

In the previous optimization the computation of the L cross correlation values for a group of segments are compute in parallel and stored in the matrix $Lags^{bl}$. This is done according to Equations (33). In this section that approach is used but instead of computing $w$ multiplications and additions for generating one cross correlation value, the operation will be first executed in batches of $s$ operations and stored in a temporal array. And then, compute the cross correlation values by executing only $w/s$ multiplications and additions. It is the same approach of section 6.2.5. However, this approach has two major differences. First, the multiplications are not previously computed. And second, the batches of $s$ multiplications and additions are done in parallel for every lag.

A new matrix is introduced for storing the intermediate operations. We can rewrite Equation (33) into Equations (37) and (38). The number of batches of $s$ operations required to compute $J$ segments is defined as $M = J + \frac{w}{s} - 1$. Therefore the matrix $MSum^{bl}_{m}(l)$ has a size $M \times L$. For each batch $m$ and every lag $l$ the computations are executed and stored. Then as shown in (38), the cross correlation values are computed for each segment $j$ by executing $\frac{w}{s}$ operations for every lag $l$.

$$MSum^{bl}_{m}(l) = \sum_{n=m+s}^{m+s+s} \left( SMA_{B}(n+l+\text{overlap}) \ast SMA_{A}(n+\text{overlap}) \right)$$

where

$$\begin{align*}
    l &= (L_{\text{max}} - t\%L) \\
    0 &< m \leq J - 1 + \frac{w}{s} \\
    0 &\leq th < \#\text{Threads} \\
    0 &\leq bl < \#\text{Blocks}
\end{align*}$$

(37)
\[ Lags_j^b(l) = \sum_{n=j}^{j+w} \left( MSum^b(n) \ast MSum^b(n) \right) \]

where \( l = (L_{\text{max}} - th\%L) \)
\[
\begin{align*}
    j &= \frac{th}{L} \\
    0 &\leq th < \# \text{Threads} \\
    0 &\leq bl < \# \text{Blocks}
\end{align*}
\] (38)

The reduction of operations is done by first computing \( s \) multiplications and additions and then \( \frac{w}{s} \) additions instead of \( w \) multiplications and additions. And the optimal reduction is when \( s \approx \sqrt{w} \).

The number of multiplications is reduced \( \frac{w}{s} \) times in comparison to the previous optimization. This is shown in Equation 39. The group of \( JL \) threads computes \( M \) batches of \( s \) multiplications and additions and store the results. This is done \( \lceil \frac{L}{J} \rceil \) times to cover all the segments. The number of additions then is equal to the number of multiplications plus the additions require to sum the batches. This is \( JL \) threads compute \( \frac{w}{s} \) additions for every segment. This is shown is Equation 40.

\[ \# \text{Mul} = J \times L \times \lceil \frac{J}{L} \rceil \times \lceil \frac{M}{J} \rceil \times s \] (39)

\[ \# \text{SUMS} = J \times L \left( \lceil \frac{L}{J} \rceil \times \frac{w}{s} \right) + \# \text{Mul} \] (40)

Using the configuration of table 6, the number of threads and usage of GPGPU stays the same as the previous optimization. The shared memory required increases in 1517 positions for the matrix \( MSum_m^b(l) \) introduced with size \( M \times L \). And the registers required are 23,552. A summary of the resources used in this implementation is shown in Table 13.

<table>
<thead>
<tr>
<th>fourth Optimization</th>
<th>TimeConsumed</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.64s</td>
<td>61.4%</td>
</tr>
<tr>
<td></td>
<td>SharedMemory</td>
<td>25.98Kb</td>
</tr>
<tr>
<td></td>
<td>Registers</td>
<td>23,552</td>
</tr>
<tr>
<td></td>
<td>Threads</td>
<td>1886</td>
</tr>
</tbody>
</table>

Table 13: Summary of results and resources (Fifth Optimization)

The time consumed by this implementation is 1.64s for 4000 lines. With this configuration, reducing the number of additions yields a slower response time. By analyzing each step it is found that for the fourth implementation it takes 0.78s for computing the matrix \( Lags^b_j(l) \) and 0.38s for finding the maximum cross correlation value. In the fifth optimization, the time for finding the maximum cross correlation value is the same 0.38s, the only change is when filling \( Lags^b_j(l) \) which is done in two steps: i) first the batches of 4 computations are executed and store in \( MSum_m^b(l) \) and ii) taking this matrix the cross correlation values are computed and stored in \( Lags^b_j(l) \). The time spent in these two steps is 0.81s: 0.378s for computing the batches and 0.432s for computing the cross correlation values.

The time consumed for computing the cross correlation values is reduced by half but the time to compose the batch makes it not efficient. On the other hand, there is a clear increment on the time spent on loading from global memory. The reason is the usage of cache memory. Since the fifth optimization makes more use of shared memory, the arrays that store the input
vector $SMA_A$ and $SMA_B$ have less chance of being cached. This was discovered by analyzing the time spent for computing fewer segments which implies less access to shared memory. The difference in time consumed between the two implementations is smaller as the number of segments computed is decreased.

This optimization can be more useful when the ratio between the segment size and the step size is larger. Then the number of additions saved by this optimization is larger and more time is saved. In the following subsection is the analysis and comparison of all the optimizations worked out. The bottom line is that depending on the configuration selected, a different optimization may be the best.

6.2.8 Comparison of Optimizations

This subsection is intended to compare the optimizations of the GPGPU implementation on different configurations. The first comparison will be on the configuration of Table 6. Then the extremes of the ratio $\frac{w}{s}$, when $\frac{w}{s} = 1$ and when $\frac{w}{s} = w$, are compared on the optimizations with different segment sizes $w$ and vector sizes $N$. With this benchmark it is intended to prove what is the best implementation for different cases.

Table 14 shows a summary of the optimizations, a brief explanation and the number of additions and multiplications that execute for computing the strain visualization. During the comparison, the optimizations will be refereed as OP1, OP2, OP3, OP4 and OP5, as shown in Table 14.

<table>
<thead>
<tr>
<th>Implementations</th>
<th>Explanation</th>
<th>#Multiplications</th>
<th>#Sums</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>Straight forward implementation of the strain visualization</td>
<td>$IwL$</td>
<td>$IwL$</td>
</tr>
<tr>
<td>1st Opt</td>
<td>Optimization using GPGPU shared memory (Section 6.2.3)</td>
<td>$IwL$</td>
<td>$IwL$</td>
</tr>
<tr>
<td>2nd Opt</td>
<td>This optimization reduces the number of multiplications to be executed (Section 6.2.4)</td>
<td>$NL$</td>
<td>$IwL$</td>
</tr>
<tr>
<td>3rd Opt</td>
<td>This optimization reduces the number of additions and multiplications to be executed (Section 6.2.5)</td>
<td>$NL$</td>
<td>$IL(s + \frac{w}{s})$</td>
</tr>
<tr>
<td>4th Opt</td>
<td>This optimization increases the utilization of the GPGPU(Section 6.2.6)</td>
<td>$JwL \times \left(\left\lceil \frac{I}{J} \right\rceil\right)$</td>
<td>$JwL \times \left(\left\lceil \frac{I}{J} \right\rceil\right)$</td>
</tr>
<tr>
<td>5th Opt</td>
<td>This optimization reduces the number of computations and increases the usage of GPGPU(Section 6.2.7)</td>
<td>$JsL \times \left(\left\lceil \frac{I}{J} \right\rceil \times \left\lceil \frac{M}{J} \right\rceil\right)$</td>
<td>$JwL \times \left\lceil \frac{I}{J} \right\rceil + #Mul$</td>
</tr>
</tbody>
</table>

Table 14: Comparison of computations and time consumed on the five implementations (Sections 6.2.4, 6.2.3, 6.2.5, 6.2.6 and 6.2.7) on a Quadro 600

In Table 15 the reader can find the summary of the time consumed and GPGPU usage of each implementation with the configuration of Table 6. The best implementation for a configuration when $\frac{w}{s} = 16$ is the OP3. However, it is possible that using different configuration another optimization would be best. For starters, we can see by looking at the equations for calculating the number of multiplications and additions to be executed that for any configuration with $s = 1$, OP3 and OP5 execute more operations than the naive implementation. And OP2 is
minimal when \( s = 1 \). Therefore we can expect OP2 to be the best optimization when \( s = 1 \).

Other assumptions can be done: i) OP2 is only better than the others when \( s = 1 \). It is obvious that OP3 and OP5 are better because the number of additions is decreased when \( s > 1 \). But even when OP4 executes the same amount of operations than the naive implementation, OP4 is better than OP2 when \( s > 2 \). The reason is the utilization of the GPGPU. By having more threads executing the latency produced by memory access is better hidden. ii) It is expected that with certain ratio \( \frac{w}{s} \) OP4 and OP5 implementations are the fastest. The reason is that the reduction of additions decreases as \( \frac{w}{s} \to 1 \) and \( \frac{w}{s} \to w \) and the reduction of multiplications is reduced as \( \frac{w}{s} \to w \). iii) It is expected that for \( \frac{w}{s} = 1 \) and \( \frac{w}{s} = w \) OP4 is faster than OP5 because the reduction of additions is not efficient in those cases. OP5 includes extra shared memory access and a larger amount of shared memory, in addition to the extra logic for storing temporal sums. Hence, the number of additions reduced should be large enough to compensate for the extra overhead. The optimal point is when \( s \approx \sqrt{w} \).

<table>
<thead>
<tr>
<th>Implementations</th>
<th>Results Consumed Time (4000)</th>
<th>GPGPU Usage</th>
<th>#Multiplications</th>
<th>#Sums</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>3.68s</td>
<td>32.84%</td>
<td>2,647,556</td>
<td>2,647,556</td>
</tr>
<tr>
<td>1st Opt</td>
<td>2.36s</td>
<td>32.84%</td>
<td>2,647,556</td>
<td>2,647,556</td>
</tr>
<tr>
<td>2nd Opt</td>
<td>2.04s</td>
<td>33.3%</td>
<td>167,936</td>
<td>2,647,556</td>
</tr>
<tr>
<td>3rd Opt</td>
<td>0.71s</td>
<td>44.43%</td>
<td>167,936</td>
<td>829,840</td>
</tr>
<tr>
<td>4th Opt</td>
<td>1.47s</td>
<td>61.4%</td>
<td>2,676,480</td>
<td>2,676,480</td>
</tr>
<tr>
<td>5th Opt</td>
<td>1.63s</td>
<td>61.4%</td>
<td>501,840</td>
<td>1,170,960</td>
</tr>
</tbody>
</table>

Table 15: Comparison of computations and time consumed on the five implementations (Sections 6.2.3, 6.2.4, 6.2.5, 6.2.6 and 6.2.7) on a Quadro 600.

For proving the assumptions a configuration with \( w = 64 \) and \( N = 4096 \), which are common values of the tool, is used for comparing the optimizations at different step sizes \( s \). The values used for \( s \) are not common for the visualization tool but are used to analyze the impact of having different amounts of computations. This analysis provides information about the tradeoffs between number of computations and usage of the GPGPU.

The results in time consumed are in Table 16. The expectations hold: i) When \( s = 1 \) the best optimization is OP2. When \( s > 1 \) the OP3 is better than OP2. ii) When \( \frac{w}{s} \geq 8 \) OP4 and OP5 are better than OP3. iv) OP4 is faster on any \( \frac{w}{s} \) and not only on the extremes. However, this expectation still hold when the segment size is increased. In Figure 18 the reader can see a graph with the number of computations that each optimization executes for these configurations.

More experiments were run with different configurations and the results are shown in Table 22 of the Appendix B. The experiments corroborate the expectations for the best optimizations according to the configurations and the analysis yield a criteria for selecting a optimization for a given configuration. There are three characteristics of the configuration to be analyzed in order to select the best optimization. And these characteristics defined the criteria for selection the best optimization for each configuration: i) When \( s = 1 \) OP2 is used. ii) OP3 is not selected when the factor of addition reduction is larger than 6. If it is smaller than 6, OP3 is selected only if the reduction factor of the multiplications is larger than 15. Otherwise, OP4 or OP5 are selected. iii) OP5 is selected when it has a reduction of 4 times the number of additions compared to OP4. Furthermore, the number of additions saved must be larger than 100. This condition is introduced because for small segments sizes \( w < 64 \), OP5 is not faster than OP4.
Table 16: Comparison of computations and time consumed on the five implementations at different configurations (Sections 6.2.3, 6.2.4, 6.2.5, 6.2.6 and 6.2.7) on a Quadro 600.

<table>
<thead>
<tr>
<th>Step size</th>
<th>1st Opt</th>
<th>2nd Opt</th>
<th>3rd Opt</th>
<th>4th Opt</th>
<th>5th Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.3s</td>
<td>3.37s</td>
<td>5.61s</td>
<td>6.9s</td>
<td>9.43s</td>
</tr>
<tr>
<td>2</td>
<td>2.10s</td>
<td>1.86s</td>
<td>1.7s</td>
<td>2.92s</td>
<td>3.67s</td>
</tr>
<tr>
<td>4</td>
<td>2.10s</td>
<td>2.11s</td>
<td>0.71s</td>
<td>1.46s</td>
<td>1.64s</td>
</tr>
<tr>
<td>8</td>
<td>3.10s</td>
<td>3.28s</td>
<td>0.81s</td>
<td>0.74s</td>
<td>0.78s</td>
</tr>
<tr>
<td>16</td>
<td>5.48s</td>
<td>5.75s</td>
<td>1.7s</td>
<td>0.39s</td>
<td>0.47s</td>
</tr>
<tr>
<td>32</td>
<td>10.44s</td>
<td>10.5s</td>
<td>5.5s</td>
<td>0.22s</td>
<td>0.33s</td>
</tr>
<tr>
<td>64</td>
<td>10.40s</td>
<td>5.8s</td>
<td>5.81s</td>
<td>0.12s</td>
<td>0.19s</td>
</tr>
</tbody>
</table>

The reason is the overhead introduced for storing temporal additions.

This conclusions are the result of an analysis of various configurations, the analysis of the number of computations of each optimization and the number of execution cycles per thread. The number of cycles was used to find a criteria to discriminate OP5 from OP3. Since OP5 uses more threads, the number of cycles per thread is less and so are the conditions executed. Therefore, the discrimination is a combination on the reduction of number of cycles due to the reduction of additions and the reduction of multiplications.

In the remainder of the section two graphs are explained. The graphs contain results in time consumed of different optimization at different configurations. In Figure 19, there is a graph of OP2 and OP3 at different configurations, showing how OP2 is always better when $s = 1$. In Figure 20 there is a graph with OP3, OP4 and OP5. This graph shows how OP3 is more efficient when $s = 2$ and $s = 4$, for larger values of $s$ the other optimizations are better. However, it also depends on the size of the segment, when $w = 128$ OP3 is best also when $s = 8$. In this case the addition reduction is 5.3 and the multiplication reduction is 15 times. Hence, the criteria for selecting the best optimization holds.

The visualization tool is able to select from the different optimizations according to the configuration specified. In the following section, the comparison between the GPGPU and the CPU implementations is done. In each configuration explored, the best implementation for GPGPU is used for the benchmark and it is the same for the CPU.
This section is intended to provide a comparison on performance of an implementation in GPGPU and CPU. In this section, only the algorithm for strain visualization is considered. Memory transfers and other processes are analyzed in Section 9 where the whole process of the visualization tool is analyzed and compared in both implementations.

First, in Figure 21 the reader can see the behavior of performance of the CPU implementation. In this graph, different configurations are shown. In the x axis it is shown the steps size used. It is important to understand that a smaller step size yields a larger overlap between segments. The maximum overlap comes when $s = 1$ which creates an overlap of $w_s = w$. As shown in Figure 21 the time consumed by the CPU implementation decreases with a smaller step size. At the same time, smaller steps sizes implies large overlaps which allow the CPU implementation to save more computations. However, the number of segments to be computed is inversely proportional to the step size $(I = (N - w)/s)$. Despite the fact that the percentage of computations saved increases, the overall number of computations ends up increasing. A large segment size decreases the number of segments to be computed, therefore a configuration with larger segment sizes decrease the time consumed.

For the GPGPU implementation it also holds that smaller step sizes and segment sizes
Figure 21: CPU implementation at different configurations

increases the number of computations. However, the ratio in which the number of computations increases due to the decrement of the step size is very different between the CPU and GPGPU implementations. The large of the overlaps between segments due to the usage of small step sizes give an important advantage to the CPU. The advantage of the CPU implementation is on the number of multiplications and additions saved due to the avoidance of redundant operations. With larger overlaps $w_s \rightarrow w$ the number of saved computations increases. Specifically, the number of multiplications only depend on the size of the vector and the lags to computed $\#MUL_{CPU} = NL$. Therefore, the usage of small step sizes do not change the number of multiplications. On the other hand the number of redundant additions increases with large overlaps.

Before the analysis of the comparison of GPGPU and CPU we would like to remind the reader that four GPGPU optimizations (OP2, OP3, OP4 and OP5) are used at different configurations. Therefore the comparison with the CPU is done on a different GPGPU implementation according to the configuration. Let’s first state the configurations to be analyzed and the GPGPU optimization used on them.

The configurations analyzes are with an input size $N = 4096$ and 5 different segment sizes $w$: $w = 64$, $w = 128$, $w = 512$, $w = 256$ and $w = 1024$. Each of the segment sizes is analyzed with overlaps: $w_s = 64$, $w_s = 32$, $w_s = 16$, $w_s = 8$ and $w_s = 4$. In Figures 22 and 23, the segment sizes are identify by different colors in the top faces of the bars and for every segment, the overlaps $w_s$ are shown in descendent order in the $x$ axis. The $y$ axis is the number of multiplications in millions.

In Figure 22 the number of computations required for both CPU and GPGPU implementations are shown in a 3D graph. Two of the optimizations developed in this Section, namely OP2 and OP3 avoid the redundant multiplications. These optimizations are used when $w = 64$, $w = 128$ and $w = 256$ with overlaps $\frac{w}{s} = 64$, $\frac{w}{s} = 32$ and $\frac{w}{s} = 16$. Hence, in those configurations the number of multiplications required is equal to the number of multiplications required by CPU. OP4 does not include any reductions of multiplications and OP5 reduces the additions and multiplications by a small factor but instead both implementations have a higher utilization of the GPGPU.

As stated in Section 6.2.8, the optimization OP3 which avoids redundant multiplications and reduces additions has a lower performance than OP4 and OP5 when the percentage of additions reduced is not high enough to compensate for the low GPGPU utilization. In Figure 23 the number of additions are shown with the same configurations displayed in Figure 22. It is
shown how in the configurations where OP3 is used are the configurations with higher number of additions. For the rest of the configurations, OP4 is only used when \( w = 64 \) with overlaps \( \frac{w}{8} = 8 \) and \( \frac{w}{8} = 4 \). The remaining configurations are executed by OP5.

Now we can proceed with the comparison between GPGPU and CPU. Taking a closer look to Figure 22 we can see the number of multiplications in the CPU is constant in 167,936 multiplications while in some configurations the GPGPU required up to above 1 million multiplications. This represents an enormous different in multiplications which is expected to be reflected in the time consumed analysis. We can conclude that the CPU maintains a regular number of multiplications and additions independently of the overlapping whereas the GPU implementation is strongly affected.

Taking a look now on Figure 23, where the number of additions required is display, it seems like the CPU implementation is also constant. This is not true, the least number of additions required is when \( w = 1024 \) and \( \frac{w}{8} = 4 \) with 252,396 and the largest number is 495,936 when \( w = 64 \) and \( \frac{w}{8} = 64 \). The changes in the GPGPU are counted in millions, therefore we can consider to be constant the number of additions of the CPU.

It is expected that the GPGPU implementation does not have a good performance when the number of additions is too high, despite the number of multiplications. I.e. the GPGPU can have
a good performance when the number of additions is low even if the number of multiplications is too high. The reason is rather simple. The number of additions implies cycles or iteration over the values of the segments for computing the cross correlation values. In all the GPGPU implementations, the number of cycles depend on the number of additions. When there is no reduction on additions, the number of additions is \( IwL \) and it is the same number of cycles. In OP2 the number of multiplications are extracted from the cycles which makes them faster but the number of cycles remain. In OP4, all the cycles are comprised with a multiplication and an addition but more threads are used. OP3 and OP5 reduce the number of cycles in the same way, the difference is OP3 extracts the multiplications from the cycles and OP5 uses more threads instead. In any case the cycles compute additions, either of multiplications stored (OP3) or multiplications being computed in the cycles (OP5). The additions are computed in groups of \( s \) and stored. Then \( \frac{w}{s} \) more additions are done for every segment for computing the cross correlation value. This implies two sets of cycles, the first one with \( s \) iterations and the second one with \( \frac{w}{s} \) iterations. The best performance is achieved when \( s = \sqrt{w} \) because the number of cycles is reduced to the minimum possible. In configurations with \( \frac{w}{s} \to w \) or \( \frac{w}{s} \to 1 \), both the number of additions and the number of cycles increase. In those scenarios the GPGPU performs poorly.

In figure 24 there is a graph of the time consumed by the CPU and GPU implementations at different configurations. Despite the difference in number of operations, the GPGPU implementation has a better performance for overlaps \( \frac{w}{s} \leq 64 \). The only exception is when \( w = 128 \), where an overlap of \( \frac{w}{s} = 64 \) makes more efficient the CPU implementation. In Figure 24 we included the case of an overlap of \( \frac{w}{s} = 128 \) in order to provide a scenario where the CPU is better. In Figure 25 is a graph showing the number of additions and multiplications required by GPGPU and CPU implementations for the configurations where CPU has a better performance.

![Figure 24: Time consumed by CPU and GPGPU implementations](image)

A configuration with a large overlap \( w/s \) yields a process for the CPU where only \( s \) additions is done for computing a segment. In an implementation on a GPGPU the additions to be computed at every segment are \( w/s \). E.g. \( w = 128 \) and \( s = 2 \), for every segment the CPU computes 2 additions whereas the GPGPU computes 64 additions. This hypothetical configuration is not feasible for useful strain visualization but it highlights the limitations that a GPGPU implementation has. The larger \( w \) and the smaller \( s \) the larger the overlap and by consequence the larger the difference in computations between CPU and GPU. However, the threshold where the GPGPU implementation has a better performance is not large. For small segment sizes \( 64 < s < 256 \), when the overlap is \( w/4 \) or less the GPGPU is faster. For large segment sizes
s \geq 256, the large data sets yield a better performance on the GPGPU implementation when the overlap is less than w/2.

On the bright side, the configurations for useful strain visualization fall in the range where the GPGPU is better. Common configurations are $w = 64$ with $s = 4$ or $s = 8$; or $w = 128$ with $s = 4$ $s = 8$ or $s = 16$. These values may be larger if the sample rate of the US signals is increased. However, in that case the input vector size increases by the same factor. E.g. in the example when $w = 64$ and $s = 4$ the vector size is $N = 4096$ and the sample frequency is $F_s = 200 \text{MHz}$. If the sampling frequency increases to $F_s = 400 \text{MHz}$ then the configurations values are doubled: $w = 128$ and $N = 8192$. It is not likely that the sampling rate would increase but it is possible that the visualization tool would be required to see deeper in the tissue which increases the input vector size but the segment and step size remain constant. In order to cover the scenario where the input vector size increases we made an analysis with $w = 64$ and $s = 4$ with different input vector sizes. The results are shown in Figure 26. The ratio in which the time consumed increases is larger on the CPU implementation. This settles that for the strain visualization with feasible configurations the GPGPU has a better performance even when the number of total computations is larger on the GPGPU implementation than the CPU implementation. The conclusion of the benchmark is that the difference in performance between CPU and GPU depends on the algorithm to be implemented. If it can be optimized by CPU using dependencies then a disadvantage for the GPGPU is introduced. However, there is a threshold depending on the number of computations saved by the CPU in comparison with the GPGPU where the GPGPU has a better performance.

A GPGPU implementation has a better performance than a CPU implementation in the case where the number of computations is the same. However, for algorithms that can be improved adding data or process dependencies, a CPU implementation may save so much computations that can end up with a better performance than a GPGPU implementation. A GPGPU implementation exploits parallelism and cannot easily implement algorithms or optimizations with data dependencies. However, as shown in this benchmark, a GPGPU implementation can still have a better performance even executing 10 times more computations.

As a final remark, the GPGPU used (Quadro 600) has only 2 multiprocessors with 48 processing cores each (96 cores) whereas there are in the market GPGPUs with 448 cores. Therefore, the performance of the GPGPU implementation can easily improved 4 times by changing the device. The flexibility of Cuda programming makes it possible to simply replace one GPGPU for the other with out changes in the code.
7 Strain Implementation on Frequency Domain

The strain visualization algorithm can also be implemented by computing the complete cross correlation function in the frequency domain. The method for implementing the cross correlation function in frequency domain is explained in this section. It is also explained how the Fast Fourier Transform (FFT) is implemented in GPGPU along with the comparison with other FFT implementation in a CPU and the NVIDIA GPGPU implementation.

Equation (41 shows the formula for computing the cross correlation function of two vectors $X$ and $Y$. However, the cross correlation function is obtained for each segment $i$ in the input vectors in order to obtain the strain.

$$\rho(t) = \text{ifft}\left(\text{fft}(X) \ast \text{fft}(Y)^*\right)$$ (41)

7.1 Overview of the algorithm

The segment vectors to apply fourier transform are defined as $segV_A^i$ which is a segment of $V_m$ and $segV_B^i$ which is a segment of $V_{m-1}$. For computing the complete cross correlation function in the frequency domain, the vectors $segV_A^i$ and $segV_B^i$, are doubled in size by padding. Therefore a vector of size $w$ is padded with $w$ zeros giving a size of $2w$. But the vectors for the segments are not of size $w$. From $V_m$ it holds that a segment of $w$ values is used for computing the cross correlation function. However, from $V_{m-1}$ the number of values taken is $w + 2 \cdot L_{\text{max}}$. The size of both vectors is defined as $segV_{\text{size}} = 2 \times (w + 2 \cdot L_{\text{max}})$.

Furthermore, the FFT needs sizes of powers of two in order to be efficient. Thereby, the size of the segment vectors is re-defined as $R_{\text{size}} = 2^{\lceil \log_2(2w-1) \rceil}$. Equation (42) shows how the segment vectors for each segment $i$ is defined.

$$segV_A^i(n) = (V_m(i \times s + n))W_A(n)$$
$$segV_B^i(n) = (V_m(i \times s + n + l))W_B(n)$$

where

$$W_A = \begin{cases} 1 & n \leq w \\ 0 & n > w \end{cases}$$

$$W_B = \begin{cases} 1 & n \leq (w + 2 \cdot L_{\text{max}}) \\ 0 & n > (w + 2 \cdot L_{\text{max}}) \end{cases}$$ (42)

In this equation, it is understood that:

Figure 26: Time consumed by CPU and GPGPU implementations at different input vector sizes
\[ V_m(n+l) = \begin{cases} 0 & (i \times s + n + l) < 0 \\ 0 & (i \times s + n + l) \geq N \end{cases} \tag{43} \]

For computing the cross correlation function of each segment \( i \), Equation (44) is implemented. For each \( i \) the segmented vectors are transformed into the frequency domain. Then the transform of the input is multiplied to the conjugate of the transform of the reference. After the inverse FFT is applied to the product yielding the cross correlation function.

\[ \rho_i(lc) = \text{ifft} \left( \text{fft}(segV_i^A) \ast \text{fft}(segV_i^B)^* \right) \tag{44} \]

Since the complete cross correlation is computed, obtaining the lag with maximum cross correlation value being the maximum lag \( L_{max} \) implies selection from the cross correlation function \( \rho_i(lc) \). That is shown in Equation (45).

\[ V_{dis}(i) = \text{arg}_{lc_{max}}(\rho_i(lc)W_{\rho}(lc)) \quad \text{where} \quad W_{\rho} = \begin{cases} 0 & 0 \leq lc < R_{size} \\ 0 & L_{max} < lc < R_{size} - L_{max} \\ 1 & \text{else} \end{cases} \tag{45} \]

### 7.2 GPGPU Implementation

In this section the overview of the strain visualization algorithm is presented when the frequency domain is used. This implementation is done on a GPGPU and has different properties than the time domain approach. The differences with the time domain approach and the steps to achieved a frequency domain implementation are presented.

The GPGPU implementation is shown in the pseudo code of Figure 27. For every segment there are five steps to compute the local displacement. First the segmentation of the input vector, second the fourier transforms, third the complex multiplications on frequency domain, fourth the result is transform back to time domain and fifth step is finding the lag of the maximum cross correlation value. The five steps are explained as follows:

**A) Segmentation of Input Vectors**

In the time domain approach, the input vectors \( V_m \) and \( V_{m-1} \) are directly taken to compute the multiplications and additions over its values. The local displacement is computed in each segment by taking the values from the input vectors. In the frequency domain approach, a segment vector \( segV^i \) is declared. This is because the fourier transform is applied to the specific segment and then transform back to the time domain for computing the cross correlation function. Therefore, the segment \( segV^A_i \) and \( segV^B_i \) are first filled as shown in Equation (42. This is an important step that is not required by the time domain approach and it produces considerable extra overhead to the process. This is shown in figure 27 in lines 12 through 17. First, the limits of the segment to be copy are computed, then with \textit{cudaMemcpy} the segment is copy from the input vectors to the segment vectors.

**B) Fourier Transforms of the segments**

The FFT of each pair of segment vectors is computed for each segment \( i \). The FFTs used are Real-To-Complex transforms. Therefore, the size of the segment vectors in the frequency domain is defined as \( F_{size} = R_{size}/2 + 1 \) where \( F_{size} \) represents complex numbers. The implementation of the FFT transform is explained in Subsection 7.3, for now, it suffices to understand that a real vector of size \( R_{size} \) is the input and a complex vector of size \( F_{size} \) is the output and it represents the frequency domain of the input.
// CLASS

TYPE can be defined as float or double

w = 64; // Segment Size
s = 4; // Step Size
N = 4096 // Vector Size
I = (N-w)/s +1; // Number of segments
MaxLag = 20; // Maximum Lag

cudaMemset(seg_A, 0.0, 256*sizeof(CLASS_TYPE)); // Set the segments to zero in all positions. This

cudaMemset(seg_B, 0.0, 256*sizeof(CLASS_TYPE)); // generates the padding automatically.

for(int nI=0; nI < I; nI++)
{
    // Memory Copy
    pOffset = min(MaxLag, nI*s);
    aOffset = max(0, nI*s - MaxLag);
    nSearchLength = min(N - aOffset, w + 2*MaxLag);
    cudaMemcpy(seg_A + pOffset, V_m + nI*s, w*sizeof(CLASS_TYPE), cudaMemcpyDeviceToDevice);
    cudaMemcpy(seg_B, V_m-1 + aOffset, nSearchLength*sizeof(CLASS_TYPE), cudaMemcpyDeviceToDevice);

    // FFT Execution
    fftR2C(seg_A, seg_ACom);
    fftR2C(seg_B, seg_BCom);

    // Complex Multiplication
    complexMul<<<blocks, threads>>>(seg_OCom, seg_ACom, seg_BCom, F_size);

    // Inverse FFT Execution
    ifftC2R(seg_OCom, rho);

    // normalizeFFT<<<blocks, threads>>>(rho, R_size); Not required in this implementation.

    // Find the lag with the max cross correlation function.
    maxLagF<<<blocks, threads>>>(V_dis, rho, MaxLag, nI);
}
C) Complex Multiplication
Equation (46) is used to multiply complex numbers. Each of the $F_{size}$ values of both transforms are multiplied following this equation. The product is a vector of the same size. Mores details about the function that computes these multiplications is provided in Appendix A.1.

$$(r_A + y_A)(r_B + y_Bi) = (r_A \times r_B + y_A \times y_B) + (r_A \times y_B + r_B \times y_A)i$$

D) iFFT of Product in Frequency Domain
The inverse fourier transform is applied for deriving the cross correlation function, as stated in Equation (44). Because of the properties of the inverse digital fourier transform, the values in time domain derive from an iFFT are normalized. However, normalization is not required for this implementation since we will pick only the max cross correlation value.

E) Find Lag of Max Cross Correlation Function
For the final step it is required to implement an extra function that finds the lag corresponding to the maximum cross correlation function. In the frequency domain the cross correlation function is derived from a inverse fourier transform. Therefore, all the cross correlation values are known after the inverse fourier transform computation. Furthermore, the lag to be selected is only within a range of the cross correlation function. This range is described in Equation (45). Hence, a function is implemented to find the lag within the range defined by the maximum lag selected. This function is not required in the time domain approach since every time a cross correlation value is computed it is compared with the previous one in order to keep the maximum. This function is explained in detail in Appendix A.2.

In the next section, the implementation on a GPGPU of the FFT transform as well as the inverse FFT transform are explained.

7.3 FFT: GPU Implementation
The explanation of the Fourier transform and its origin fall out of the scope of this thesis. The reader can refer to [16] for more information about the Fourier transform and its applications.

Equation 47 shows the formula for the implementation of a DFT, see [17]. This equation implies $N^2$ complex multiplications where $N$ is the size of the vector to be transformed into the frequency domain. The DFT equation can be easily implemented on a GPGPU since the computation of each value is independent from the rest. A GPGPU implementation of the DFT is implemented with benchmark purposes but it is not covered in this document.

$$X^k = \sum_{n=0}^{N-1} x_n e^{-2i\pi \frac{k}{N} n}$$

During the 60’s the American mathematicians Cooley and Tukey [18] developed an algorithm for computing the Fourier transform with only $N \log_2(N)$ complex computations. Their approach enables a recursive implementation using Equation (48) as a base. The vector is first split in two, having the odd indexes in one segment and the pair indexes in another segment. Then, the segments are divided again in two until the size of the subsegments is two. This is depicted in a diagram on Figure 28. The FFT of the subsegments of size two is computed, then the FFT of the subsegment is used to compute the FFT of a higher level segment until the complete FFT is computed.

$$X^k = \sum_{m=0}^{N-1} x_{2m} W^{(2m)k} + \sum_{m=0}^{N-1} x_{2m+1} W^{(2m+1)k} \quad where \quad W = e^{-\frac{2i\pi}{N}}$$
The operations used for computing the complete FFT can be graphically seen by a butterfly network diagram. In Figure 29 there is an example of butterfly network for computing the FFT of a vector of size 16.

As depicted in Figure 29, it takes \(4 \times (\log_2(N))\) steps to compute the fourier transform for a 16 \((N)\) samples vector and each step requires 16 \((N)\) complex multiplications. Therefore, the diagram shows that the total number of complex multiplications is \(N\log_2(N)\). The reader can notice that each of the 4 steps depend on the results of previous steps. The dependencies are drawn with arrows. These dependencies represents a challenge for a GPGPU implementation. The implementation is to be modified in order to increase the parallelization level.

In this thesis the approach taken is to convert the recursive method into an iterative method. In Figure 30 there is the pseudo code which represents the FFT algorithm in an iterative way. The minimum size of the input vector accepted by the implementation is \(N = 4\). In the first step, pair of values are taken from the input vector. In the second step, the values used come from the result of the first step and it is the same for the following steps until the complete FFT is being computed. Each steps is done in parallel by \(N/2\) threads but the steps cannot be computed independently from the previous steps. Therefore, after each step, the threads are synchronized.

The synchronization is possible only on threads within the same block. Therefore, when multiple blocks are released, each block computes a partial FFT and stores it on global memory.
Figure 30: Pseudocode: FFT in shared memory

```c
X(n) fftItera(x(n), N)
{
    th // thread Id
    bl // block Id
    N // size of the vector
    x(n) // vector in the time domain
    X(n) // Output vector in the frequency domain

    nIndex = bl + th * #Blocks;
    tempLength = N/2;
    blockLength = tempLength/gridDim.x;
    step = (anHeight/2)/gridDim.x;
    Nt = 2;
    i = 0;
    j = 1;

    X_i[th] = x[nIndex] + x[nIndex + tempLength];
    X_i[th + blockLength] = x[nIndex] - x[nIndex + tempLength];

    synchthreads(); // Synchronize the threads

    Nt*=2;
    step/=2;
    while(step>0)
    {
        k = th/step;
        W_k = (k*exp(2*PIi))/Nt;
        X_j[th] = X_i[th + k*step] + X_i[th + (k+1)*step]*W_k;
        X_j[th + step] = X_i[th + k*step] - X_i[th + (k+1)*step]*W_k;
        i++; j++;
        Nt*=2;
        step/=2;
    }
    synchthreads();
    for(int n=0;n<N/2+1;n++)
    {
        X[n] = X_i[n];
    }
    return X[n];
}
```
Figure 31: Pseudocode: FFT in global memory

```c
X(n) fftComb(x(n), N, step)
{
    th // thread Id
    N // size of the vector
    x(n) // vector in the time domain
    X(n) // Output vector in the frequency domain

    nOffset = threadIdx.x + blockIdx.x*blockDim.x;
    dbFactor = (-2.0 * 3.141592654);
    nInOffset = anHeight/2;
    nJump = gridDim.x*blockDim.x;
    while(nOffset < nInOffset)
    {
        k = nOffset/anStep;
        cs = cos(dbFactor*(CLASS_TYPE)k/(CLASS_TYPE)anHeight/(CLASS_TYPE)step));
        sn = sin(dbFactor*(CLASS_TYPE)k/(CLASS_TYPE)anHeight/(CLASS_TYPE)step));
        nI1 = nOffset%step + 2*k*anStep;
        nI2 = nI1 + step;
        tmp1 = cs*apdbInput[nI2].r + sn*apdbInput[nI2].i;
        tmp2 = cs*apdbInput[nI2].i - sn*apdbInput[nI2].r;
        if(step!=1||nOffset==0)
        {
            X(n)[nOffset + nInOffset].r = x(n)[nI1].r - tmp1;
            X(n)[nOffset + nInOffset].i = x(n)[nI1].i - tmp2;
        }
        X(n)[nOffset].r = x(n)[nI1].r + tmp1;
        X(n)[nOffset].i = x(n)[nI1].i + tmp2;
        nOffset+=nJump;
    }
}
```
Then the kernel function $FFTComb()$ that utilizes $N$ threads combines the partial FFTs, the pseudocode of this function is shown in Figure 31.

The partial FFTs are possible as described by Equation (48). The vector can be split in pairs and odds and apply the FFT to both segments independently. It is also possible to subdivide the segments for allowing 4 FFTs in parallel. This property can be graphically seen in Figure 29 where the input values are rearranged. In the first half we find the pair numbers and in the second half we find the odd numbers. Furthermore, we can see each half is further split in odds and pairs.

We can see in Figure 29 that for computing the first half of the FFT values, the computations of the first three steps are independent from the computations done for the second half of the FFT. The first half only uses pair numbers and the second half only odd numbers. Therefore, the two halves can be compute in parallel and it is only in the last step that they are combined. It is also possible to divide the input vector in 4 segments and execute the first two steps independently, then in the last two steps the results are combined. An example of splitting the samples in 4 is shown in Figure 32. In this example, the functions $fftItera()$ and $fftComb()$ introduced in figures 30 and 31 are used. The procedure is the same, the four segments are computed in parallel and then they are combined into two segments that represent the odds and pairs segments. Finally the two intermediate segments are combined to derive the FFT. A similar method is used by Govindaraju[21].

The inner rectangles in Figure 32 represent each a GPGPU block. Therefore, the function in each square is executed in parallel by different threads and using shared memory. Then the middle size rectangles represent how the threads of two GPGPU blocks work together to combine the results. The threads access global memory for computing $fftComb$. Finally, the outer rectangle represents how all the threads combine the results, also by reading from global memory. Therefore the FFT is computed in three steps. Govindaraju[21] does something similar and he calls it "Hierarchical FFT", he divides the FFT into small blocks that can be computed using shared memory. Then the blocks are combined to compute the FFT. Govindaraju[21] also considers FFT sizes that are not powers of two and explains in detail the decisions taken for efficient global and shared memory access. In this thesis only FFT with sizes that are power of two are considered and no further optimizations other than an optimal selection of number of threads and blocks and use the use of shared memory.

For executing $fftItera()$, the selection of the number of blocks to release depend on two factors: the size of the FFT and the maximum number of threads that can be released. The
number of blocks released is the number of segments to divide the input vector. When \( N \) is too small, it is more efficient to simply use one GPGPU block. In this way, global memory is only accessed on the first step and the rest of the execution shared memory is used. If \( N \geq 1024 \), it is better to make use of more than one block. The minimum number of threads required to compute an FFT is \( \frac{N}{2} \) and for this FFT implementation it is more efficient to have at least 256 threads per block. The maximum number of threads to be used is determined by the shared memory of the multiprocessors and the maximum number of threads that can be allocated in a block

\[
#\text{Threads}_{\text{FFT}} = \min\left(\#MAX\text{Threads}_{\text{block}}, 2^{\log_2\left(\frac{\#\text{Mem}\text{Block}}{2\times (type)}\right)}\right) \tag{72}\]

With this, the number of blocks that represent the number of sub-FFTs is

\[
#\text{Blocks} = \left\lceil \frac{N}{2\times #\text{Threads}_{\text{FFT}}} \right\rceil \tag{73}
\]

and the number of threads is defined as

\[
#\text{Threads} = \frac{N}{2\times #\text{Blocks}} \tag{74}
\]

For executing \( fft\text{Comb}() \), the selection of the number of blocks is simpler. Since this function does not use shared memory, there are far less restrictions. Thereby, the number of blocks used is set as

\[
#\text{Blocks} = #MP \times 2 \tag{75}
\]

and the number of threads is simply

\[
#\text{ Threads} = \min\left(#\text{MAX}\text{Threads}/2, \frac{N}{2\times #\text{Blocks}}\right) \tag{76}
\]

In the configuration of Table 19 the device is a Quadro 600 which has 2 multiprocessors. For efficiency purposes the minimum number of blocks is 2 per multiprocessor because the multiprocessor can hold 1546 threads and the maximum number of threads per block is 1024. But in this case the input vector size is \( N = 256 \) which means only 128 threads are used to compute the FFT. Therefore only one block per multiprocessor is enough. The function \( fft\text{Comb}() \) is not used since the FFT was not segmented.

In the following section there is a comparison of this implementation with NVIDIA’s implementation and with the CPU library \( \texttt{fftw} \).

### 7.4 Benchmark FFT: GPU Implementation, \( \texttt{fftw} \), \( \texttt{cuFFT} \)

This section is dedicated to analyzing the performance of the FFT implementation explained in this thesis with a GPGPU implementation and a CPU implementation form the market.

The GPGPU FFT implementation selected for the benchmark is NVIDIA’s \( \texttt{cuFFT} \) Library. The GPGPU used for the benchmark is a Quadro 600 and the CPU is a Intel Xeon CPU (3.33 GHz). In table 17 is a summary of the time consumed by the three implementations at different FFT sizes. This time only covers the execution of the FFT and it does not take into account any memory transfer. This is important because the time spent to transfer memory from CPU to GPU and backwards is not trivial.

The reader can notice right away there is a big difference in time consumed between the CPU implementation and the GPU implementations. These results indicate that a GPGPU implementation is much better than a CPU implementation when the Fourier transforms are required. The three implementations are easy to use and the developers are not required to have any knowledge about Fourier transforms not about the architecture of GPGPUs or CPUs. The only advantage of the CPU FFT is that the user does not have to include special memory transfers.

Memory transfers from CPU memory to GPGPU global memory is something that has to be dealt with and must be taken into account on the benchmark. In table 18 there is the times spent in the memory transfer \( (\text{GPU} \rightarrow \text{CPU} + \text{CPU} \rightarrow \text{GPU}) \) for each size of the FFT. Despite the extra time consumed for the memory transfers the GPGPU implementations have a better performance. Figure 33 shows a graph with the time consumed for computing the FFT of different sizes, the time shown includes the memory transfers. For a better performance, the developer using a GPGPU FFT can take advantage of the property of GPGPU architecture that allow for transferring data \( (\text{GPU} \rightarrow \text{CPU} \text{ or viceversa}) \) and executing computations at the same time. The kernel calls are do not have to be synchronized with the CPU. This allows
<table>
<thead>
<tr>
<th>input size (doubles)</th>
<th>GPGPU Implementation (ms)</th>
<th>NVIDIA cuFFT (ms) 3.2</th>
<th>fftw (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>0.02</td>
<td>0.013</td>
<td>0.34</td>
</tr>
<tr>
<td>512</td>
<td>0.025</td>
<td>0.015</td>
<td>0.60</td>
</tr>
<tr>
<td>1024</td>
<td>0.036</td>
<td>0.017</td>
<td>0.55</td>
</tr>
<tr>
<td>2048</td>
<td>0.046</td>
<td>0.02</td>
<td>0.60</td>
</tr>
<tr>
<td>4096</td>
<td>0.102</td>
<td>0.036</td>
<td>1.12</td>
</tr>
<tr>
<td>8192</td>
<td>0.242</td>
<td>0.051</td>
<td>1.7</td>
</tr>
<tr>
<td>16384</td>
<td>0.52</td>
<td>0.091</td>
<td>3.5</td>
</tr>
<tr>
<td>32768</td>
<td>1.12</td>
<td>0.21</td>
<td>5.05</td>
</tr>
<tr>
<td>65536</td>
<td>2.33</td>
<td>0.37</td>
<td>10.7</td>
</tr>
<tr>
<td>131072</td>
<td>4.78</td>
<td>0.75</td>
<td>23</td>
</tr>
<tr>
<td>1,048,576</td>
<td>43</td>
<td>6.8</td>
<td>335</td>
</tr>
</tbody>
</table>

Table 17: Computation time of a GPGPU FFT implementation, cuFFT and FFTw at different input sizes

<table>
<thead>
<tr>
<th>input size (doubles)</th>
<th>Size (Bytes)</th>
<th>Mem Transfer time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>2Kb</td>
<td>2 × 0.039</td>
</tr>
<tr>
<td>512</td>
<td>4Kb</td>
<td>2 × 0.040</td>
</tr>
<tr>
<td>1024</td>
<td>8Kb</td>
<td>2 × 0.0418</td>
</tr>
<tr>
<td>2048</td>
<td>16Kb</td>
<td>2 × 0.0435</td>
</tr>
<tr>
<td>4096</td>
<td>32Kb</td>
<td>2 × 0.047</td>
</tr>
<tr>
<td>8192</td>
<td>64Kb</td>
<td>2 × 0.061</td>
</tr>
<tr>
<td>16384</td>
<td>128Kb</td>
<td>2 × 0.083</td>
</tr>
<tr>
<td>32768</td>
<td>256Kb</td>
<td>2 × 0.127</td>
</tr>
<tr>
<td>65536</td>
<td>512Kb</td>
<td>2 × 0.209</td>
</tr>
<tr>
<td>131072</td>
<td>1Mb</td>
<td>2 × 0.395</td>
</tr>
<tr>
<td>1,048,576</td>
<td>8Mb</td>
<td>2 × 2.23</td>
</tr>
</tbody>
</table>

Table 18: Time consumed for memory transfers between CPU and GPGPU

The CPU to make all the kernel calls required and then wait for the results, instead of waiting after every call. In this fashion, the CPU can make a kernel call, request a memory transfer immediately after and then another kernel call without waiting for any of those to be completed. The GPGPU will first execute the kernel, when it finishes it will start the memory transfer and the second kernel at the same time (provided that the second kernel does not require the results of the first one). This is call streaming, the reader can find more information about it in [22].

The version used of the NVIDIA library is cuFFT 3.2 released in 2011. The improvement of the library can be discovered by observing the benchmarks made in the past for measuring and comparing the cuFFT. In 2008, Govindaraju [21] compares his implementation of the fft on a GPGPU with the cuFFT. He does not specify the version of the cuFFT that he used. In his benchmark he used a GTX280 with 240 cores and the performance of the cuFFT was up to 100 GFLOPS for 1D FFTs. And the performance of his own implementation reach a performance up to 250 GFLOPS. However, at the end of 2011 Duan [23] develops an FPGA implementation of the FFT and compares it with cuFFT 3.2 on a GTX480 with 480 cores. In this comparison the performance of cuFFT 3.2 is up to 400 GFLOPS for 1D FFTs. He concludes that the GPGPU FFT has a better performance in GFLOPS while the FPGA is better for efficient power consumption. Furthermore, FPGA implementations represent a bigger
challenge for design.

There is not much information about the architecture of the library cuFFT 3.2. However, it is clear that a lot of effort has been applied to its optimization since it released in 2007 until the release of the last version.

7.5 Frequency Domain vs Time Domain

In this section, an analysis of the frequency domain implementation for the strain visualization algorithm is provided. This includes results using the configuration shown in Table 19. Moreover, the analysis includes the number of computations required and it is compared with the time domain approach. Finally the comparison on time consumed by the two implementations is presented.

The most expensive process in number of computations comes from the Fourier transforms. Each of the transforms require $R_{size} \log_2(R_{size})$ complex multiplications and additions. That means the total number of multiplications for one FFT is $\#MUL_{fft} = 4R_{size}\log_2(R_{size})$ and the number of additions is $\#SUM_{fft} = 2R_{size}\log_2(R_{size})$.

For each segment $i$ three Fourier transforms are required. Therefore, the total number of multiplications required for the Fourier transforms is $\#MUL_{fftI} = 12I \times R_{size}\log_2(R_{size})$ and the number of additions is $\#SUM_{fftI} = 6I \times R_{size}\log_2(R_{size})$. In addition, the frequency representations of the segment vectors are multiplied. The size of these vectors is $F_{size}$ complex numbers, therefore for every segment $i$, $4F_{size}$ multiplications and $2F_{size}$ additions are added. The total number of computations required for the frequency domain implementation of the strain visualization algorithm is shown in Equation (49). SFD stands for strain in frequency domain.

$$\#MUL_{SFD} = I(12 \times R_{size}\log_2(R_{size}) + 4F_{size})$$ (49)
Table 19: Example of System’s Settings

<table>
<thead>
<tr>
<th>GPUProperty</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU Name</td>
<td>Quadro600</td>
</tr>
<tr>
<td>#MP</td>
<td>2</td>
</tr>
<tr>
<td>#MAXThreads</td>
<td>1536</td>
</tr>
<tr>
<td>SMem&lt;sub&gt;MP&lt;/sub&gt;</td>
<td>48Kb</td>
</tr>
<tr>
<td>#Registers&lt;sub&gt;MP&lt;/sub&gt;</td>
<td>32768</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>4096</td>
</tr>
<tr>
<td>w</td>
<td>64</td>
</tr>
<tr>
<td>s</td>
<td>4</td>
</tr>
<tr>
<td>L&lt;sub&gt;max&lt;/sub&gt;</td>
<td>20</td>
</tr>
<tr>
<td>L</td>
<td>41</td>
</tr>
<tr>
<td>I</td>
<td>1009</td>
</tr>
<tr>
<td>R&lt;sub&gt;size&lt;/sub&gt;</td>
<td>256</td>
</tr>
<tr>
<td>F&lt;sub&gt;size&lt;/sub&gt;</td>
<td>129</td>
</tr>
</tbody>
</table>

\[ #SUM_{SFD} = I(6 \times R_{size} \log_2(R_{size}) + 2F_{size}) \]

The comparison in number of computations with the naive implementation is shown in Equation (50). The decision to compare with the naive implementation is because it comes simpler to make a comparison and to analyze cases with different configurations.

\[
\frac{#MUL_{SFD}}{#MUL_{STD}} = \frac{I(12 \times R_{size} \log_2(R_{size}) + 4F_{size})}{IwL} \quad (50)
\]

According to the configuration in Table 19, the number of multiplications required is 25,317,828. The straight forward implementation in the time domain only requires 2,647,556 multiplications. The difference is much bigger and provides very convincing proof that the frequency domain approach is not efficient for the implementation of this strain visualization algorithm.

There are two reasons for the difference in multiplications: i) the fourier transform is applied in segments of a vector and it has to be applied many times (once for each segment i) plus every time the inverse transform is applied. ii) In the time domain, the cross correlation function is not complete, therefore less multiplication are required meanwhile in the frequency domain the complete cross correlation function is computed.

We can make a better comparison if we decide to compute the complete cross correlation function also for the time domain approach. Then we define \( L_{\text{opt}} = \frac{w}{2} - 1 \). And we redefine \( R_{size} \) as \( R_{size} = 8L_{\text{max}} + 4 \). With this modifications we can derive Equation (51), STD stands for strain in time domain. The interpretation of this equation is that a large segment size \( w \) would make more efficient implementation than the naive time domain implementation. According to this equation, the ratio between \( \frac{w}{2} \) does not play a role nor does the size of the input vector \( N \).

\[
\frac{#MUL_{SFD}}{#MUL_{STD}} = \frac{I(12 \times (8L_{\text{max}} + 4) \log_2(8L_{\text{max}} + 4) + 8L_{\text{max}} + 12))}{I(4L_{\text{max}}^2 + 6L_{\text{max}} + 2)} \quad (51)
\]

Now we will find a configuration such the frequency domain would be more optimal than the time domain approach. In order to rule out the optimizations done in the time domain, we can
assume $w = s$ which means there are no overlaps within the segments and the optimizations in time domain have no effect. In Table 20 we show the multiplications required by time domain approach and frequency domain approach at different segment size. The vector size was set as $N = 16384$. The results suggest that for large segments the frequency domain can be more efficient in number of computations.

However, for the purposes of this strain visualization it is not tangible to set the segment size to $w = 1024$. That would require a vector size of $N = 64536$ which implies either a much better resolution or increasing the maximum depth to be monitored. Either way is not required by the visualization tool. The following comparison is on time consumed, but due to the analysis on number of multiplications the reader should not expect that the time consumed by the frequency domain would be better than that of the time domain.

<table>
<thead>
<tr>
<th>Segment Size ($w$)</th>
<th>Maximum Lag ($L_{\text{max}}$)</th>
<th>Number of Segments</th>
<th>$#\text{MUL}_{\text{STD}}$</th>
<th>$#\text{MUL}_{\text{STD}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>31</td>
<td>255</td>
<td>6,168,960</td>
<td>1,028,160</td>
</tr>
<tr>
<td>128</td>
<td>63</td>
<td>127</td>
<td>6,967,728</td>
<td>2,064,512</td>
</tr>
<tr>
<td>256</td>
<td>127</td>
<td>63</td>
<td>7,711,200</td>
<td>4,112,640</td>
</tr>
<tr>
<td>512</td>
<td>255</td>
<td>27</td>
<td>7,270,560</td>
<td>7,064,064</td>
</tr>
<tr>
<td>1024</td>
<td>511</td>
<td>15</td>
<td>8,838,720</td>
<td>15,728,640</td>
</tr>
<tr>
<td>2048</td>
<td>1023</td>
<td>7</td>
<td>8,941,296</td>
<td>29,345,792</td>
</tr>
<tr>
<td>4096</td>
<td>2047</td>
<td>3</td>
<td>8,255,520</td>
<td>50,319,360</td>
</tr>
<tr>
<td>8192</td>
<td>4095</td>
<td>1</td>
<td>5,897,520</td>
<td>67,100,672</td>
</tr>
</tbody>
</table>

Table 20: Comparisson of number of multiplications in Time domain and Frequency domain

The time consumed by the frequency domain approach is divided in the 5 steps required to compute the strain using the fourier transforms. In Table 21 are the results with the configuration of Table 19 and using the fft implemented by NVIDIA.

<table>
<thead>
<tr>
<th>Step</th>
<th>Time consumed per segment (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment Memory</td>
<td>0.078</td>
</tr>
<tr>
<td>FFT's</td>
<td>0.03</td>
</tr>
<tr>
<td>Complex Multiplication</td>
<td>0.005</td>
</tr>
<tr>
<td>Inverse FFT</td>
<td>0.016</td>
</tr>
<tr>
<td>Findlay</td>
<td>0.01</td>
</tr>
<tr>
<td>TOTAL</td>
<td>0.139</td>
</tr>
</tbody>
</table>

Table 21: Time consumed for computing the cross correlation per segment in frequency domain approach

The time consumed by every segment is 0.139 ms which means it takes 140 ms for computing only the input vector. The reader can see that the step that takes the longest is the segmentation of the input. This step requires memory transfer of global memory. The fourier transforms and the multiplications together take less time than the memory transfer. This can be solved by using streaming, the steps can be done in the order shown in Figure 34. In order to achieve this, the memory transfers are done using `cudaMemcpyAsync()` and GPGPU streams, the reader can refer to [22] for more details about streaming in GPGPU. Meanwhile the computations of the current segment are being executed, the data transfer for the next segment is being done at the same time. Thus, 0.39 ms are saved per line. Nevertheless, that gives a overall time of 0.1 ms per segment and 100.9 ms per line.
Therefore it is stated that the frequency domain is not efficient for computing the strain visualization algorithm and it is much better to keep an implementation in time domain.

CONCLUSIONS: × For the desired configuration of the strain visualization tool, the time domain implementation proved to be better than a frequency domain implementation. The main reason is that an FFT and iFFT are computed for each segment of an input vector. Since the segments are small e.g. \( w = 64 \) or \( w = 128 \) it is faster to compute the cross correlation function in time. Besides, the frequency domain approach computes a complete cross correlation function whereas normally only a few lags are required.

× For a configuration with very large segments and where the complete cross correlation has to be computed, the frequency domain is more efficient than a time domain. However, a configuration with segments \( (w > 1024) \) is not used.

× The fourier transform implemented on a GPGPU is much better than a CPU FFT implementation.
8 Post-Processing

The strain visualization has noisy results as shown in Figure 13. Therefore a post processing step is required to provide a meaningful visualization. This section is intended to analyze the decisions taken for the post processing. It was concluded that the reason for the noisy region is because the amplitude of the signal in that area is low and mostly due to noise. Thereby the algorithm for strain visualization takes random lags for representing the local displacement. Hence, it was decided to pre-process the input signals for avoiding the regions where the input is mostly noise.

In order to leave out the noisy regions, the signal is clipped in a range defined as \(-th > V_m < th\) where \(th\) represents a threshold where the signal is too low to be account for. The signal on this range is simply set to zero and the rest of the signal remains equal. The value of the threshold is not easy to determine and may not give the same results at every time during an ablation experiment. In Figure 35, are 4 images: the original image and three with different threshold values. The experiment is 2.4 seconds of information. The region where the curves are displayed is the region of interest, the rest of the image seems to be only noise. The original image is clearly too noisy outside the region of interest while it is completely left out with a threshold = 0.08. However, some information of the region of interest is also lost. There is a tradeoff on the noise removed and the meaningful information lost. It may be necessary to implement a dynamic threshold change on the visualization tool for providing freedom on selecting meaningful strain visualization.

![Figure 35: Strain Visualization: Output after local displacement detection with pre-filter input at different thresholds](image)

Part of the noise problem is apparent quick displacement of the tissue. The gray values...
oscillate too fast from light values to dark values which would mean sudden changes on the tissue. However, it is not normal that the heart tissue moves so fast and we conclude that the algorithm miss-match some segments and point them to segments farther than the actual displacement. In order to prove this we run an experiment with a limitation of the max speed sensed. In figure 36, 4 images showing 2.4 seconds of the same experiment are displayed. The gray level representation covers 41 values that represent the ranges of speed \([-0.075,0.075]\) mm/ms. The top right image shows the result when the max speed sensed is the maximum possible 0.075. The other three images are limited on the max speed sensed. The result is less noisy images with a narrower gray range. The bottom left image is limited to a speed 10 times slower than the maximum, therefore, only 5 gray levels are available. However, it gives an image with less noise.

![Figure 36: Strain Visualization: Local displacement estimation limited to different speeds.](image)

With a limitation on the speed sensed, it is possible to lose information because the algorithm may not be able to detect the displacement of a segment simply because it moved too fast. However, as shown in Figure 37, the gray values are the same on a max speed of 0.075 and a max speed of 0.018. This implies at this region, the displacement is localized in the same lag which is on the range of the 11 values possible on a max speed of 0.018 mm/ms. As shown in Figure 37, the only difference in color is on the noise hence the limitation of the speed helps the visualization.

The result can use contrast enhancement for extending the narrow gray level range. However, if we down sample the input rate, we can decrease the maximum speed sensed and stick with 41 values. Hence, we can achieve a better resolution of the changes of speed on the tissue. In order to make these changes more clear, the gray level scale is changed into a color visualization. The values in the range \([0.0, 0.5]\) are changed into a range of \([0.0, 1.0]\) of color blue and represents the upwards movement and the values in the range \((0.5, 1.0]\) are changed into a range of \([0.0, 1.0]\) of color red and represent the downwards movement. In Figure 38 the reader
can see the representation in red-blue of the strain visualization sensing a maximum speed of 0.075mm/ms. In addition, the gray scale strain visualization and the contrast visualization of the same information is displayed.

Figure 38: Strain Visualization in Color
In order to have a better resolution on the changes of speed we down sample the input by half. This changes the input rate for the strain visualization from $1000\text{Hz}$ to $250\text{Hz}$. Then the maximum speed is set to $0.018\frac{\text{mm}}{\text{ms}}$ and since the input rate is slower, more lags are required to sense that speed. Therefore 41 values do the scale for the maximum speed $0.018\frac{\text{mm}}{\text{ms}}$ instead of only 11 values. With this settings, Figure 39 shows the visualization if we limit the max speed sensed to $0.018\frac{\text{mm}}{\text{ms}}$, $0.009\frac{\text{mm}}{\text{ms}}$ and $0.0045\frac{\text{mm}}{\text{ms}}$. Unlike the previous experiment where the maximum speed was $0.075\frac{\text{mm}}{\text{ms}}$, at this maximum speed, it is notorious that limiting the sensed speed, information is lost. With the limit in $0.018$ and $0.009 \frac{\text{mm}}{\text{ms}}$ the range of colors is the same, but when the limit is $0.0045 \frac{\text{mm}}{\text{ms}}$, the range of colors is narrower. Therefore, some local displacement is not detected anymore.

There are external factors that influence the displacement sensed by the visualization tool such the movement of the catheters and the difference on the influence of the heart beat on the tissue of different regions of the heart. This work only explores the possibilities of extra computations for increasing the quality of the visualization. Further research is to be done on determining the correct configurations for the threshold, the maximum sensed speed and the down sampling. What it is important for this work are the implications on performance. By decreasing the maximum speed sensed, the maximum lag is also decrease which implies less computations for calculating each cross correlation function. Another possibility to keep the maximum lag is to down sample the input rate which means less RF lines computed per second. Either way, the implication is less work for the processing unit implemented, CPU or GPGPU.
Figure 39: Strain Visualization in Color at different maximum speeds
9 System Benchmark: CPU vs GPU

This section is intended to make a comparison on performance of the implementations in a Intel Xeon CPU (3.33GHz) and in a Quadro 600 GPU. The benchmark is done comparing the time consumed for processing 4000 RF lines in both contrast visualization and strain visualization. This benchmark includes the memory transfers from CPU memory to GPGPU memory required in the GPGPU implementation.

The purpose of this thesis is to implement a strain visualization on a GPGPU. The strain visualization is the second visualization developed for the visualization tool intended for helping physicians during the catheter ablation procedure. The idea of adding a second visualization namely the strain visualization is to provide with more information during the ablation procedure. It is understood that one visualization does not replace the other and it is the combination of the two that comprises the visualization tool. The straight forward approach is to simply set a toggle that allows the user to change from one visualization to the other. This is the step implemented, tested and used for the benchmark throughout this thesis. However there is a possibility of showing both visualizations combined in one but it is not presented in this thesis work.

In Figure 40 the reader can see a graph showing the time consumed by each of the steps of the implementation. First, the data is copy into the GPGPU, this step is not required by the CPU implementation. After the data is normalized into a range from (-1 to 1), this data is used by the two visualization algorithms. The contrast visualization requires first the envelope detection and the constrast enhancement. Later the data is gamma corrected and converted in a 8 bit gray scale. Finally the data is stored back to CPU memory. The strain visualization algorithm applies the cross correlation function and also applies gamma correction. Then the data is also transfer to CPU memory. The strain algorithm requires the vector processed for processing the next vector, therefore a memory copy from GPGPU memory to GPGPU memory is implemented.

![Figure 40: Time consumed at every step of the implementation on both Intel Xeon CPU (3.33GHz) and Quadro 600](image-url)
In the graph of Figure 40 there are two configurations shown, both with \( w = 64 \) and \( s = 4 \) while one has a vector size \( N = 4096 \) and the second one \( N = 12288 \). The comparison on different input sizes is important because in the future it may be required to see deeper in heart tissue which implies a larger input vector for storing all the information. In addition, a setup where the number of computation required is larger, the GPGPU has an advantage over the CPU. In Figure 40 the reader can see when the input size increases, the difference in performance between the two implementation is larger. The envelope detection developed in [3] is not optimized for CPU, maybe an optimization would decrease the gap in performance with the GPGPU. However, it can hardly improve to reach the GPGPU implementation. Besides, further optimization on the GPGPU envelope detection could also be applied. Furthermore, despite the cross correlation step is optimized in CPU for avoiding redundant operation, the implementation on a GPGPU is faster.

A very simple function is the step for color conversion where gamma correction is applied. Gamma correction requires to increase by the power of \( \frac{1}{\gamma} \) and a multiplication to scale the range \((0...1)\) to a 8 bit value. The function is very straightforward, the operation is done for every pixel. Therefore, the number of computations done by both CPU and GPGPU implementations is the same. Thus, the color conversion step is a good example to analyze the power of the GPGPUs. In Figure 40, it is shown that GPGPU is 10 times faster for computing color conversion with an input vector of \( N = 4096 \) and it is also around 10 times faster with \( N = 12288 \). The conclusion is for algorithms is absolutely no dependencies among the data to be processed, a Quadro 600 with 96 cores is 10 times faster then a Intel Xeon CPU (3.33GHz).

For the rest of the steps which represent a minor challenge for computation power, GPGPU remains slightly better than CPU. One disadvantage of the GPGPU implementation is the memory transfer between the CPU and the GPGPU before starting the process. However this is not a bottle neck for new GPGPU architectures. GPGPU devices have the capacity to make memory transfers from CPU to GPGPU and vice versa in parallel with the computations. Therefore, the only real drawback is additional coding for the implementation for allowing memory copies and kernel functions at the same time.

Despite the fact that the memory transfers from CPU to GPGPU can be neglected, in Figure 41 the total consumed time including memory transfers is shown for both implementation. This time includes all the steps previously discuss for contrast visualization and strain visualization. In addition, the GPGPU includes the post processing introduced in Section 8. The CPU does not include the post processing because it was not implemented on the CPU in this thesis. Even with the extra post processing, the implementation on a Quadro 600 is better than the CPU implementation. In Figure 41 the configurations used are also with input vector sizes of \( N = 4096 \) and \( N = 12288 \) each with maximum lags \( L_{\text{max}} = 20 \) and \( L_{\text{max}} = 5 \). The motivation for this 4 configurations is that in the future of the visualization tool it may be necessary to increase the input vector size for increasing the depth seen in the heart tissue; and that the maximum lag may be reduced for achieving a more meaningful strain visualization.

9.1 Conclusions

On the configurations with input size \( N = 4096 \), the GPGPU can process 4000 lines in less than one second. That allows the visualization tool to work at an input rate of \( 1kHz \). The CPU implementation can only achieve \( 500Hz \) when the maximum lag is \( L_{\text{max}} = 5 \), this configuration may be enough for the visualization tool with the price of the lower resolution. Besides, the time consumed by CPU implementation is not taking in account any post-processing.

The GPGPU implementation is overall 4 times faster than the CPU implementation. With input size of \( N = 4096 \) is around 3.5 times faster and with \( N = 12288 \) it is around 4.5 times faster.
faster. The speedup is shown in Figure 42. With this results, it is stated that the GPGPU allows for better resolution visualization and yet leaves space for post processing.

An important remark is that a CPU implementation is still feasible with lower input rates and higher latency. But this implies a lower resolution on the visualization. Another remark is that the processing of the inputs is the same for a number of transducers which implies
a multi-core implementation can also be useful. A very important advantage of a multi-core implementation is on market status. One concern of Philips researchers about the GPGPU approach is the availability of the graphic card in the market in the long term. It is more likely to count on a Intel multi core processor than on a NVIDIA graphic card. This concern may change in the close future because of the great power of the GPGPUs and because of the number of new research efforts around it.

It is fair to say that the information of each transducer can easily be assigned to an independent core of a multi-core CPU. Thereby, the performance increases 4 times on a configuration of 4 transducers and 4 cores. This allows to close the gap on performance with the GPGPU implementation and allows input rates of 1kHz with input vector sizes of $N \leq 4096$. However, this only holds if the CPU implementation does not have to compute the post-processing introduced in Section 8. With input vectors sizes $N > 4096$, the GPGPU implementation is more than 4 times faster than the CPU implementation even when the CPU does not compute the post processing.

The bottom line is that a CPU implementation is feasible by reducing the quality of the visualization, even when adding more cores to the execution. More than one core per transducer would be more expensive than a GPGPU (The price for a Intel Xeon CPU (3.33GHz) with 6 cores is 583.00$ and the price of a Quadro 600 with 96 cores is 180.00$). A better investment in case of more computation power required would be buying a better GPGPU (The price of a Quadro 2000 with with the double of the cores of a Quadro 600 is $< 430.00$). Besides the price, a GPGPU implementation written with CUDA C allows for replacing the GPGPU device and exploit the resources without changing the code. This is not the case of a multi-core implementation where a different implementation may be required as the number of cores increases.

A GPGPU implementation is easy to port on more powerful GPGPU devices and performs better than a CPU on algorithms with high computation demands. Allows for flexibility in the processing added in the system. E.g. the addition of post processing or pre processing for achieving better quality. CPU implementations may imply too many sacrifices for achieving a good time performance.


10 Thesis Conclusions

★ A strain visualization was implemented in GPGPU in time domain and frequency domain. It was proved that the time domain approach in this particular project is no more efficient than a frequency domain approach. Mainly because the complete cross correlation function is not required and because the FFT and iFFT have to be computed for many segments in each input vector.

★ In time domain, the strain visualization was implemented on a CPU and optimized for reducing the number of operations. The reduction of operations with respect to the GPGPU implementation produces a difference in operations to be computed up to millions. i.e. the GPGPU computes millions of computations whereas the CPU computes a few thousands. Despite this, the GPGPU has a better performance for useful configurations.

★ An implementation of the Fourier transform was done for a GPGPU and optimized. It was compared to NVIDIA library cuFFT and a CPU library FFTw. It was proved that GPGPUs have a great performance computing FFTs and that cuFFT is a well design library that achieves much better performance than other processing units, including FPGAs when computing very large FFTs.

★ The subject algorithm, the cross correlation function, has a high level of parallelization. However, it also can be optimized for sequential processing and reduce hundreds of times the number of computations. Therefore, it is not the best example of an algorithm where the GPGPU has a major advantage in front of a CPU implementation. However, it was proved that even in such cases, the power of GPGPU can overcome the disadvantage in number of computations and still give a better performance.

★ Finally, it was proved that a GPGPU implementation of the strain and contrast visualizations is much faster than a CPU implementation. Hence, it can achieve higher input rates at real-time and with low latency. This holds even when extra post processing is adding to the visualization tools.

★ A GPGPU offers a possibility of high performance at low price and easily scalable implementations. It allows for high data input rates which in visualization tools is translated in better quality Images. The prices of the GPGPU devices are more affordable than multi-core CPUs and the replacement of the devices does not introduce extra effort on the implementation.
Appendix

A Functions used in FFT Strain Visualization Algorithm

A.1 Complex Multiplications

The pseudo code for computing the multiplications of complex numbers is shown in Figure 43. Every element \( i \) of \( X[i] \) is multiplied with the conjugate of the element \( Y[i] \). Every thread release computes a multiplication when \( \#Blocks \times \#Threads = N \). The number of blocks release is \( \#Blocks = 2 \times \#MP \) and the number of threads is \( \#Threads = \frac{\#MAXTHREADS}{2} \). If \( \#Blocks \times \#Threads < N \), the threads compute more than one complex multiplication. This is achieved by the while which increases the index by the number of total threads available \( (\#Blocks \times \#Threads) \). When \( \#Blocks \times \#Threads > N \) then not all the threads are used.
A.2 Find the Lag of the Maximum Cross Correlation Value

Finding the maximum lag is done only by one thread using the pseudo code in Figure 44. The thread analyzes $2 \times L_{\text{max}} + 1$ values of the cross correlation function computed by the inverse FFT. The values compare are in the ranges $\rho[R_{\text{size}} - L_{\text{max}} \leq index < R_{\text{size}}]$ for the lags in the range $-20 \leq l < 0$; and $\rho[0 \leq index \leq L_{\text{max}}]$ for the lags in the range $0 \leq l \leq 20$. For large $L_{\text{max}}$ it is better to have multiple threads within a block analyzing different numbers and storing the greater value on shared memory. And then compare the values of shared memory to find the lag.
B  GPGPU Optimizations Benchmark

This appendix contains the table with the summary of results of 4 optimizations at different configurations.
<table>
<thead>
<tr>
<th>Step size</th>
<th>2nd Opt</th>
<th>3rd Opt</th>
<th>4th Opt</th>
<th>5th Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.37s</td>
<td>5.61s</td>
<td>6.9s</td>
<td>9.43s</td>
</tr>
<tr>
<td>2</td>
<td>1.86s</td>
<td>1.7s</td>
<td>2.92s</td>
<td>3.67s</td>
</tr>
<tr>
<td>4</td>
<td>2.11s</td>
<td>0.71s</td>
<td>1.46s</td>
<td>1.64s</td>
</tr>
<tr>
<td>8</td>
<td>3.28s</td>
<td>0.81s</td>
<td>0.74s</td>
<td>0.78s</td>
</tr>
<tr>
<td>16</td>
<td>5.76s</td>
<td>1.7s</td>
<td>0.39s</td>
<td>0.47s</td>
</tr>
<tr>
<td>32</td>
<td>10.5s</td>
<td>5.5s</td>
<td>0.22s</td>
<td>0.33s</td>
</tr>
<tr>
<td>64</td>
<td>5.8s</td>
<td>5.81s</td>
<td>0.12s</td>
<td>0.19s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step size</th>
<th>2nd Opt</th>
<th>3rd Opt</th>
<th>4th Opt</th>
<th>5th Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.05s</td>
<td>9.71s</td>
<td>6.05s</td>
<td>13.78s</td>
</tr>
<tr>
<td>2</td>
<td>3.31s</td>
<td>3.12s</td>
<td>3.31s</td>
<td>4.88s</td>
</tr>
<tr>
<td>4</td>
<td>3.9s</td>
<td>0.98s</td>
<td>2.3s</td>
<td>1.94s</td>
</tr>
<tr>
<td>8</td>
<td>6.28s</td>
<td>0.93s</td>
<td>1.21s</td>
<td>0.95s</td>
</tr>
<tr>
<td>16</td>
<td>11.21s</td>
<td>1.74s</td>
<td>0.66s</td>
<td>0.48s</td>
</tr>
<tr>
<td>32</td>
<td>20.69s</td>
<td>5.68s</td>
<td>0.35s</td>
<td>0.36s</td>
</tr>
<tr>
<td>64</td>
<td>10.73s</td>
<td>6.09s</td>
<td>0.18s</td>
<td>0.24s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step size</th>
<th>2nd Opt</th>
<th>3rd Opt</th>
<th>4th Opt</th>
<th>5th Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10.73s</td>
<td>16.56s</td>
<td>14.83s</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>5.78s</td>
<td>4.5s</td>
<td>7.42s</td>
<td>6.93s</td>
</tr>
<tr>
<td>4</td>
<td>7.47s</td>
<td>1.67s</td>
<td>3.72s</td>
<td>2.54s</td>
</tr>
<tr>
<td>8</td>
<td>12.27s</td>
<td>1.29s</td>
<td>1.86s</td>
<td>1.05s</td>
</tr>
<tr>
<td>16</td>
<td>21.16s</td>
<td>1.99s</td>
<td>0.94s</td>
<td>0.53s</td>
</tr>
<tr>
<td>32</td>
<td>21.5s</td>
<td>5.73s</td>
<td>0.59s</td>
<td>0.31s</td>
</tr>
<tr>
<td>64</td>
<td>40.1s</td>
<td>6.77s</td>
<td>0.3s</td>
<td>0.27s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step size</th>
<th>2nd Opt</th>
<th>3rd Opt</th>
<th>4th Opt</th>
<th>5th Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19.9s</td>
<td>30.17s</td>
<td>25.4s</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>10.95s</td>
<td>7.95s</td>
<td>12.71s</td>
<td>N/A</td>
</tr>
<tr>
<td>4</td>
<td>14.6s</td>
<td>2.66s</td>
<td>6.36s</td>
<td>3.64s</td>
</tr>
<tr>
<td>8</td>
<td>24.7s</td>
<td>1.78s</td>
<td>3.27s</td>
<td>1.49s</td>
</tr>
<tr>
<td>16</td>
<td>44.8s</td>
<td>2.24s</td>
<td>1.71s</td>
<td>0.66s</td>
</tr>
<tr>
<td>32</td>
<td>76.07s</td>
<td>5.85s</td>
<td>1.08s</td>
<td>0.47s</td>
</tr>
<tr>
<td>64</td>
<td>41.43s</td>
<td>7.74s</td>
<td>0.55s</td>
<td>0.28s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step size</th>
<th>2nd Opt</th>
<th>3rd Opt</th>
<th>4th Opt</th>
<th>5th Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>36.7s</td>
<td>50s</td>
<td>41.47s</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>20.51s</td>
<td>17s</td>
<td>20.74s</td>
<td>N/A</td>
</tr>
<tr>
<td>4</td>
<td>28.93s</td>
<td>4.63s</td>
<td>10.38s</td>
<td>N/A</td>
</tr>
<tr>
<td>8</td>
<td>48.22s</td>
<td>2.77s</td>
<td>5.35s</td>
<td>N/A</td>
</tr>
<tr>
<td>16</td>
<td>87.92s</td>
<td>2.75s</td>
<td>2.68s</td>
<td>0.81s</td>
</tr>
<tr>
<td>32</td>
<td>136.35s</td>
<td>6.12s</td>
<td>1.76s</td>
<td>0.41s</td>
</tr>
<tr>
<td>64</td>
<td>77.71s</td>
<td>9.77s</td>
<td>1.02s</td>
<td>0.26s</td>
</tr>
</tbody>
</table>

Table 22: Comparison of computations and time consumed on the five GPGPU implementations (Sections ) on a Quadro 600.

Remark*: In the fifth optimization, it requires too much memory for large overlaps, thereby the implementation does not apply to those cases.
<table>
<thead>
<tr>
<th>Step size</th>
<th>N=8192 w=64</th>
<th>N=8192 w=128</th>
<th>N=16384 w=64</th>
<th>N=16384 w=128</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9.13s 11.62s 12.21s 19.02s</td>
<td>16.61s 20.14s 18.2s 27.5s</td>
<td>11.2s 16.56s 18.64s 30.9s</td>
<td>20.09s 28.6s 27.74s 41.9s</td>
</tr>
<tr>
<td>2</td>
<td>5.81s 4.16s 6.12s 7.4s</td>
<td>10.43s 6.26s 9.11s 9.75s</td>
<td>6.3s 5.4s 9.1s 11.01s</td>
<td>11.23s 8.37s 13.88s 14.85s</td>
</tr>
<tr>
<td>4</td>
<td>4.21s 1.79s 3.07s 3.39s</td>
<td>7.77s 2.33s 4.56s 3.88s</td>
<td>4.53s 2.22s 4.56s 4.92s</td>
<td>8.3s 2.97s 6.95s 5.9s</td>
</tr>
<tr>
<td>8</td>
<td>6.5s 1.86s 1.58s 1.62s</td>
<td>12.53s 2.08s 2.29s 1.9s</td>
<td>6.68s 2.03s 2.3s 2.37s</td>
<td>12.7s 2.23s 3.49s 2.76s</td>
</tr>
<tr>
<td>16</td>
<td>11.46s 3.6s 0.8s 0.92s</td>
<td>22.43s 3.75s 1.21s 0.96s</td>
<td>11.55s 3.69s 1.17s 2.37s</td>
<td>22.51s 3.81s 1.76s 1.39s</td>
</tr>
<tr>
<td>32</td>
<td>20.85s 11.27s 0.46s 0.65s</td>
<td>41.23s 11.34s 0.71s 0.71s</td>
<td>21.47s 11.4s 0.65s 0.87s</td>
<td>32.37s 11.47s 1s 0.95s</td>
</tr>
<tr>
<td>64</td>
<td>11.45s 12.14s 0.24s 0.37s</td>
<td>21.33s 12.18s 0.36s 0.48s</td>
<td>16.53s 17.22s 0.33s 0.5s</td>
<td>31.35s 17.27s 0.52s 0.64s</td>
</tr>
</tbody>
</table>

Table 23: Comparison of computations and time consumed on the five GPGPU implementations (Sections ) on a Quadro 600 [Continuation]
References


