**NTNU – Trondheim**
Norwegian University of
Science and Technology

# Application of Parallel Programming in a Automatic Detector for a Pulsed MTD Radar system

Automatic Detection and Fast Ordered Selection Algorithms

Hans Erik Fjeld

Master of Science in Electronics
Submission date: June 2012
Supervisor: Morten Olavsbråten, IET
Co-supervisor: Yngve Steinheim, SINTEF
Erik Løvli, Radian AS

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

# Oppgavetekst

SINTEF og Radian AS på Kongsberg arbeider for tiden med nye metoder for bedring av deteksjonsevne for navigasjonsradar.

En av metodene går ut på å forbedre deteksjon av bevegelige mål ved å ta i bruk et sett av parallelle digitale dopplerfiltre.
Dette kalles også MTD (Moving Target Detector) eller "puls-doppler" prosessering.
En dopplerfilterbank skiller uønskede ekko (clutter) fra interessante objekter i separate hastighetskanaler.
Dermed blir det enklere å detektere objekter som er i bevegelse i forhold til sine omgivelser.
Arbeidet så langt viser at MTD signalbehandling på disse forholdsvis enkle og billige radarsystemene er mulig.
Signalopptak og analyse av reelle radarsignaler er nå godt etablert og en arbeider for tiden med en teknologi-demonstrator som kan vise et MTD-behandlet radarbilde i nær sann tid. Demonstratoren er implementert på en standard PC-plattform.

Demonstratoren videreutvikles nå for automatisert deteksjon ved å skille objekter fra clutter vha signal-styrke og bevegelse.
Teknologien vurderes for flere anvendelser der signal-behandlingen må tilpasses hver spesifikk oppgave.
Eksempler kan være deteksjon av piratangrep, inntrengingsalarm for oppdrettsanlegg og båtmarinaer, eller deteksjon av fugl for forskning på fuglers bevegelsesmønster.
Et mastergradsarbeid på automatisert deteksjon av båter på sjøen ble utført våren 2011 etterfulgt av et prosjektoppgave-arbeid høsten samme år.
Dette vil danne et godt faglig grunnlag for masteroppgaven.

Oppgaven går ut på å sette seg inn i radar-signalbehandling med vekt på metodene som allerede er under utvikling i prosjektet.
Dernest skal en arbeide videre med metode og algoritmeutvikling.
Siden prosjektets teknologidemonstrator er PC basert er det naturlig at ny programvare utvikles på PC plattform i MATLAB og høynivåspråk.
Aktuelle problemstillinger og oppgaver er som følger:
• Hurtige CFAR algoritmer anvendt på reelle radarsignaler
• Implementering på forskjellige hardware plattformer
• Parallelle algoritmer og datastrukturer som egner seg for flerkjerne hardwareteknologi
• Anvendelser av egnede programmeringsspråk(C/C++,Matlab)
• Anvende utvidelser og biblioteker for parallell programmering i høynivåspråk.(CUDA C, OpenMp, Thrust, Arrayfire)

Veiledere: Morten Olavsbråten(IET) Yngve Steinheim(SINTEF IKT) og Erik Løvli(Radian AS)

# Abbreviations

ASR - Airport surveillance Radar
CA - Cell Averaging
CFAR - Constant False Alarm Rate
CPU - Central Processing Unit
CUDA - Compute Unified Device Architecture
CUT - Cell(s) Under Test
FFT - Fast Fourier Transform
FIFO-First In First Out
FPGA - Field Programmable Gate Array
GO - Greater Of
GPU- Graphics Processing Unit
GP-GPU - General Purpose GPU
MTD - Moving Target Detector
PRF - Pulse Repetition Frequency
PRI - Pulse Repetition Interval
RADAR - RAdio Detection And Ranging
SIMT - Single Instruction Multiple Threads STL - Standard Template Library

# Abstracts

Automatic CFAR Detection is to be implemented in a real time pulsed MTD radar system, used in a maritime application.
The CFAR should be able to have good detection properties in bad weather conditions, where rough sea states, heavy downpour and high winds are expected.
Sufficient detection properties may be achieved using an Ordered Statistics based CFAR to generate detection threshold for the MTD radar video signal.
The MTD video is the coherent raw video of the signal filtered in a bandpass filter bank, separating the Doppler frequency space of the video into a number of individual Doppler channels.
The Doppler frequency shift relates to a velocity, implying that every Doppler channel represents a velocity space, so that targets and clutter may further resolved by their relative Doppler velocity
CFAR algorithms are applied to all the test cells in the MTD video signal. These algorithms have to estimate a threshold that is used at discriminating real targets from clutter in all the velocity channels of the MTD video.
A good threshold estimate is to have a low probability of false detections, and a high probability of declaring actual targets.
This is to be valid in all clutter conditions, even when one or multiple targets are surrounded by non-stationary clutter and closely spaced.
The Ordered Statistics algorithms involves using the k'th largest value of the test window as a mean clutter power estimate for its corresponding test cell.
The ordered statistics model makes a threshold selection based on the rank of the samples. A task with complexity increasing as a function of window length and k parameter.
This task is to be performed on a large number of test cells in a system running real time. In a real time radar system, all processing have to be done before the next scan becomes available.
Radian AS works on developing a PC based MTD Radar system for a pulsed Doppler radar.
The radar interfaces the PC through a PCI Express radar receiver card developed by Radian AS.
This thesis investigates the application of parallel programming in C/C++ in order to achieve real time automatic detection in a PC based MTD radar. Two means of parallel programming are considered, involving exploitation of multi core CPU architecture as well as using a dedicated GPU as a co processor.
OpenMP is an Open Source library with compiler instructions for running tasks in parallel over multiple cores in a CPU. It is easily incorporated into C/C++ code, and may be used with most multi core CPUs.
nVidia has made GP-GPU computing available to the public through CUDA, selling CUDA enabled graphics cards and providing the tools as well as documentation needed for a programmer to be able to use the GPU as co processor.
CUDA C integrates the SIMT abstractions of CUDA, and a programmer may write C code that is compiled and executed on the GPU.
Different implementations of the OS-CFAR algorithm for threshold estimation are implemented using CUDA and OpenMP.
The different implementations are evaluated and compared to each other in terms of the results gathered from executing them on MTD video.
The experiences drawn from this work is concluded with respect to the application of parallel programming, and further recommendations for the future of the project of making a PC based pulsed MTD Radar signal processor.
This thesis introduces a CUDA algorithm for high throughput ordered selection using short window lengths on a large number of cells under test.
An algorithm developed in C for the project assignment leading up to this thesis is enabled openMP, along with a C++ STL algorithm, for performing ordered statistics ranked selection on the CPU. In addition, the CUDA OS-CFAR algorithm is ported to C with openMP.
The three implementations in C/C++ are compared to the CUDA C implementation.

# Sammendrag

Automatisk deteksjon ved bruk av CFAR skal implementeres i et pulset MTD radar system, som kjører i sanntid og er rettet mot maritimt brukt. CFAR detektoren må ha gode deteksjonsegenskaper selv under dårlige værforhold, der det kan forventes høy sjø, kraftig nedbør og sterk vind.
Tilstrekkelige deteksjonsegenskaper oppnåes ved å bruke en CFAR som ved ordnet utvalg velger terskler som brukes for å deklarere deteksjoner i MTD video signalet. MTD videosignalet er den koherente råvideoen, båndpassfilterer i en filterbank.
Dette deler hele dopplerspekteret i råvideoen opp i doppler frekvensbånd.
Doppler frekvensskift i videosignalet kan relateres til en hastighet, som betyr at hver dopplerkanal dekker et eget hastighetsområde. Og alt signal i kanalen ligger innenfor dette hastighetsområdett.
CFAR algoritmen behandler all sampler som kvalifiseres som testsampler i MTD signalet, og genererer terskler for å diskriminere reelle mål fra clutter i hvert hastighetsbånd.
Et godt estimat må ha en lav sannsynlighet for falsk deteksjon, samt høyst sannsynlig detektere alle ønskelige mål. Estimatet må være godt selv under forhold med mye clutter. Og må kunne separere en eller flere mål som befinner seg nær hverandre, selv om de er omringet av clutter.
OS-CFAR, foretar et ordnet utvalg ved å velge den k'te største verdien som befinner seg i et vinduet ekstrahert for et testsample. Denne verdien brukes som et estimat for clutter støyeffekt for et korresponderende testsample. Og en innbyrdes rangering av utvalgsmengden må foretaes. En oppgave som øker i kompleksitet med utvalgsmengden og k verdi.
Denne oppgaven må utføres for en stor mengde testsampler i et system som kjører i sanntid. For å realisere et sanntidssystem må all prosessering være ferdig før neste scan med video blir tilgjengelig.
Radian AS jobber med å utvikle en PC basert MTD radar prossesor for en pulset doppler radar.
Radaren er koblet til PCen gjennom en radarmottaker laget på et PCI Ekspresskort, utviklet av Radian AS.
Denne oppgaven søker å utforske mulighetene for å bruke parallell programmering i C/C++ til å realisere kjøretidene som kreves for automatisk deteksjon i et sanntidssystem.
To innfallsvinkler mot parallell programmering er valgt; utnyttelse av flerkjerners prosessorarkitektur, samt bruken av et grafikkort som en ekstern co-prosessor.
OpenMP er et open source initiativ som tilgjengliggjør et bibliotek med et sett kompilatorinstruksjoner som gjør en programmerer i stand til fordele oppgaver ut over flere prosessorkjerner. OpenMP integreres lett i C/C++ kode, og kan brukes med de aller fleste flerkjernearkitekturer.
Grafikkortprodusenten nVidia har tilgjengeliggjort GP-GPU programmering til forbrukerene gjennom CUDA.De selger CUDA støttede grafikkort, og tilgjengeligjør verktøyene, samt dokumentasjonen som trengs for en programmerer til å lære seg å bruke grafikkortet som en coprosessor.
CUDA C integrerer SIMT abstraksjonene til CUDA med C, og programmereren kan skrive C kode som kompileres for å kunne kjøres på grafikkortet.
Det er laget forskjellige implementasjoner av CFAR algoritmer for terskelberegninger ved å bruke CUDA og openMP.
De forskjellige implementasjonene er evaluert og sammenliknet med hverandre, basert på de resulterende kjøretidene for prosessering av MTD video.
Erfaringer skapt gjennom dette arbeidet ligger til grunne for en konklusjon vedrørende bruken av parallel programmering i en PC basert pulset MTD radar prosessor.
Denne oppgaven introduserer en CUDA algoritme som søker å maksimere yteevne for et ordnet utvalg i små tallmengder, for en stor mengde tall.
En algoritme utviklet i C i prosjektoppgaven som ledet opp mot denne oppgaven er utforsket sammen med en algoritme som finnes i C++ STL biblioteket. OpenMP er tatt i bruk på begge disse OS-CFAR implementasjonene.
I tillegg er algoritmen utviklet for CUDA portet om til 'vanlig' C og testet med openMP. De tre C/C++ implementsjonene er sammenliknet opp mot CUDA implementasjonen.

# Contents

# 1 Introduction

**Motivation**:
A radar systems consists of transceiver section where a high frequency pulse is generated, this pulse is radiated from a antenna connected to the the pulse generator. After radiating a pulse, the antenna listens for a return echo, which is amplified and sampled in the receiver section of the radar system.
Then signal processing is performed on the signal available at the end of the receiver section. This echo signal is referred to as the Video signal.
Modern digital radar receiver is realized using FPGAs as processing cores, performing number crunching and controlling external digital signal processing (DSP) circuits or application specific integrated circuits ASICs.
The entire radar system is realized in a circuit, that is printed on PCB boards, and mounted in a black box.
The black box may come with I/O units for control, and setting parameters, as well as a dedicated monitor for the PPI display.
This is an expensive solution of realizing a radar system, making the manufacturer of the system the only ones with creative control over it. And all alterations and modification of the system have to be implemented by the producer.
Although the firmware of the system may be upgraded after it has entered production, all hardware choices are set.
As technology advances, the personal computer now available to the consumer marked have become more advanced and continues to grow in functionality.
If the receiver of the radar system may be connected to a PC, all signal processing may be performed using hardware available for computers.
This hardware is mass produced. And capable of processing a lot of information with a large throughput.
A PC also comes with a lot of slots where new hardware may be mounted to upgrade it's performance and functionality.
Processors have evolved into multiple core architectures, running tasks in parallel to increase the computational throughput demanded by the consumer.
The demand for high resolution computer graphics have never been higher, and advance graphical processing units are standard components that comes with every computer that attaches an external monitor.
In addition, the processor of a PC may be programmed to perform tasks created by the user.
Numerous high and low level computer languages have been developed to program the computer, and the instructions of how to learn these languages are available for free on the internet, through reference manuals supplied with compilers, turorials and documentation.
The compiler interprets instructions of a program code into computer code, and builds the program into sets of instructions that are executed by the computer.
This thesis will use the C/C++ language, and the Visual Studio 2010 development program and compiler.
Additional extensions may easily added to this program.
Such as the CUDA C tool box, featuring ways to create and build programs that runs on a CUDA enabled graphics card, as well as analysing how they are run on the GPU.
The idea is to use the programmable GPU card as a co processor, to remove bottlenecks in the processing chain. So that more advanced functionality may be incorporated in a system running in real time.
A radar system running on a PC is easily upgraded in terms of software, and additional functionality may be easily incorporated depending on application.
By adding a CUDA enabled graphics card as a processor to the system. The system specification and cost will depend on the prize of the CUDA enabled GPU.
CUDA enabled graphics cards are available over a large prize range, where the price reflects the computing power of the device.
CUDA enabled cards are available in a large product range. From cheap entry level cards with 16-48 CUDA cores, costing about 50 $ to advanced cards used for supercomputing with 10000 cores, costing thousands of $s.

If using a GPU a as a processing unit, the hardware cost of integrating more advanced signal processing in the radar system will just be the cost of the CUDA card.
CUDA is scalable, so that functionallity may easily be ported to suit the chosen GPU, reducing development time for new applications. Making it possible to provide an advanced application specific radar system at a low cost for both the consumer and the developer.
Radian AS has produced a receiver PCI Express board, using a Xilinx FPGA ,a six layer PCB and standard components that performs detection, and makes the radar scan data available in the PC RAM.
Making the scan data ready to processed by the computer, and displayed on the computer monitor with the program PPI 4.2 for the operator.
The challenge lies in achieving real time radar signal processing for a large coverage area in a system having MTD functionality.
In a real time system, the processing of each scan has to be done, and the scan printed to the PPI display, before the next scan becomes available.
To be able to create a framework around the signal processor. The specifications of the radar system have to be set.
The detection specifications is derived from testing different implementations in Matlab, and the solutions selected has to achieve good detection rates in a large number of scenarios.
The hardware specification is set by the PCI Express card and is the amount of data gathered each scan depends on the maximum range/ sector coverage needed in the application.
Solutions made in Matlab are too slow for a real time implementation, so bottlenecks in Matlab are moved to be executed outside of Matlab.
Eventually, the entire signal processor is to be moved outside of Matlab. And be implemented in C/C++ to reduce runtime.
This thesis looks into an implementation of automatic detection using CFARs in C/C++ using different extensions for parallel programming.

**The Assignment**:
Nowadays, multicore CPUs and powerful GPUs also having multiple cores for rendering high definition video, game graphics and 3D video, are found in almost all new computers. This assignment investigates applications where multicore CPUs and GP-GPU programming are used to reduce the run times of algorithms being executed in a radar processor.
A project assignment was completed the fall of 2011, leading up to this master thesis, where an algorithm was developed and implemented, reducing the original processing time of the CFAR detector.
This OS-CFAR algorithm is revisited, and made to exploit multiple computer cores with the OpenMP library. A C++ STL version of the mentioned algorithm is also tested, using C++ built in data containers and functions. In addition, both an OS CFAR algorithm, as well as a CA CFAR algorithm is implemented in CUDA C. To be executed on the computers graphics card.
This thesis have consisted of reading up on different aspects of parallel programming, and an implementations using CUDA C with alternatives will be presented and thoroughly discussed throughout the thesis.
This thesis will try to serve as a guide to beginning CUDA programming, underlining some of the architectures functionality and characteristics. While at the same time evaluating the application of parallel programming in a MTD radar system.
The flexibility and compromises concerning different means of performing processing in parallel will be discussed, and will be considered when interpreting the resulting run times of the different implementations.
Leading up to a solution recommended by the author based on the work being presented before the conclusion of this report.

# 2 Theory

**Preface**

In this section, general theory concerning radar systems, detection of signals in noise, along with the signal processing chain of the radar system will be explained.
The parts of the signal processing chain that is to be optimized, along with the optimization technique used will also be explained.
Some basics on the front of algorithms will be presented along with the nature of the problem.
The actual implementation of the algorithms will be presented in the design section.

## 2.1 The Radar Principle

**From pulse generations to the display**
A radar displays how the emitted pulses are propagating in the environment which surrounds the radar.
The radar radiates pulses from an antenna, and listens for the return echo of the transmitted pulses. The square of the magnitude of each pulse echo is printed as a line on the PPI display.
The radar may be omnidirectional, and have a 360 degree view of the surrounding area. Or only be monitoring a fixed sector of the surroundings.
The sector covered by the radar is often referenced in polar coordinates with components in both the azimuthal plane, and in range for practical reasons. As the antenna is in the origo of coordinate system describing the sector.
One pulse echo is called a PRI. And it covers the time the receiver listens for the return signal.
Each PRI is sampled and quantized in the detector circuit in the receiver, and contains the return echo partitioned into range samples.
One sector may consist of a large number of pulses. Where this number is given by the rotation rate of the antenna, and the length of each PRI.
The PRIs form a matrix which is a map of the area which the radar covers, this is called the video signal.
The antenna is highly directional, and shapes the exited electromagnetic energy into a narrow beam. Shooting out focused pulses of energy with a high rate as the antenna slowly rotates, giving a dense coverage of the environment monitored by the radar.
The PRF(Pulse Repetition Frequency) of the radar system is the inverse of the PRI. It is a measure of how many pulses are emitted per second.
The antenna may be mechanically rotated to be able to scan, and some more advanced radars employs phase steering to steer the the radiation pattern formed by an array of antennas.
The array of antennas are fixed, but the phase of the signal power fed to the antenna elements are controlled. Thereby controlling the radiation pattern.
The radiation pattern of an array of antennas is a function on how one individual element radiates, as well as an array factor describing how the elements are mounted in relation to each other.
There exists a vast application for radar systems. Deploying various signal processing to reject clutter and detect targets of interest.
The design of a radar system is customized to the application and the market for the solution.
Tailor made solutions, as well as research and development is expensive. This makes technology expensive before even considering production costs.
A radar is a sensor used to gather remote information. That information may be meteorological, it may be surveillance of sea areas by satellite using radar imaging. It might be aerospace control and monitoring. Collection of bird data. Marine harbour surveillance, or ship bridge control. etc.
Each application demanding its own signal processing.
The radar as a remote sensor, may be used to measure velocity of objects if phase shifts between different

PRI samples may be measured.
This can be done if coherence is established between the radiated pulse and the received pulse, for every pulse.
Doppler resolution may be retained if each pulse is generated from a stable oscillator. The return signal may be mixed with a signal generated from the same oscillator that generated the transmitted pulse.
Radars using a stable oscillator are called Constant Wave(CW) radars.
CW radars may not inherently have a good range azimuth resolution, but have high frequency resolution [1]. The CW signal may be modulated to create better azimuth/ range resolution.
CW radars often have very short wavelength, and are often used in close range applications.
Radars shooting pulses during sweeps are called pulsed radars. These have good range/azimuth resolution.
And in simpler radar systems, a magnetron tube is used as a pulse generator. In more advanced pulsed radar systems, a stable oscillator is used for pulse generation, but this increases the expense of the system.
As a magnetron is used as a pulse generator in the radar supplying data for this thesis, the focus should be directed to what a magnetron tube really is.
The magnetron was developed in the late 1930s [1]. And is a cavity resonator, where the cavity design determines the oscillating frequency.
When a magnetron is applied a high voltage pulse. It begins to resonate at a high frequency. This generates a signal pulse with a random starting phase.
To make a magnetron based radar system coherent. Estimates for the signal parameters of the transmitted pulse has to be made. Generating Doppler resolution in this manner is called coherence on receive [1].

**Clutter**

Clutter in radar terminology refers to any unwanted return echo that may interfere with the detection of real target [1].
It is simply noise, but yet noise that share a lot of properties with the targets we may wish to detect with the radar. As detecting moving vehicles or birds with an airport surveillance radar. Refractions from an uneven sea surface, or large rain drops from an approaching storm.
Clutter may be spread over large areas, generating a large number of correlated samples. It may surround a real target. And may be moving with a Doppler velocity.
Clutter may severely degrade CFAR detection performance if the system is not designed properly [4]. Making the radar useless in a situation where it would make a great aid.
As in example a boat that is navigating passage with high traffic density of cargo ships and ferries in bad weather with high sea state and heavy downpour.

**MTD**

MTD is an evolved way of MTI processing explained in the theory section in the project thesis [6].
Summarized, MTI involves filtering the signal to remove clutter from stationary objects in the radar echo.
The video signal after filtering now represents how the return signal power is reflected as a function of range for a sector that is scanned by the radar.
Since the signal is coherent. It has Doppler resolution, and the velocity of objects detected in the system may be found.
MTD partitions the Doppler spectrum of the video signal into a number of separate channels. Each channel representing a band in the velocity space that is to be processed individually.
The MTD originates from an Airport Surveillance Radar developed by MIT Lincoln Laboratory, according to Skolnik [1], and was an innovative step in detection of air crafts in clutter in 1989.
The Lincoln Laboratory Journal, 1989 [11] mentions the following properties of the MTD radar system.
The ASR employed adaptive digital signal and data processing. It used an eight pulse FFT digital filter bank with three pulse delay line canceller.
And used Doppler processing to eliminate ground and rain clutter, and a clutter map, so that targets moving perpendicular relative to the radar line of sight could be detected from the clutter background.

It also applied CFAR detection with adaptive thresholds, suppressing false targets as birds and moving ground vehicles. Alternate PRFs was used to avoid Doppler ambiguities, and to unmask targets in clutter.[11]
MTD was developed to achieve sufficient stationary clutter rejection, good performance in rain, low false alarm rate and further clutter rejection by scan to scan processing[11]. These are also characteristics desired in a naval radar system. A system that also have to provide good detection characteristics in a wide range of clutter situations that arises from bad weather. Such as waves on the sea surface and heavy rain.

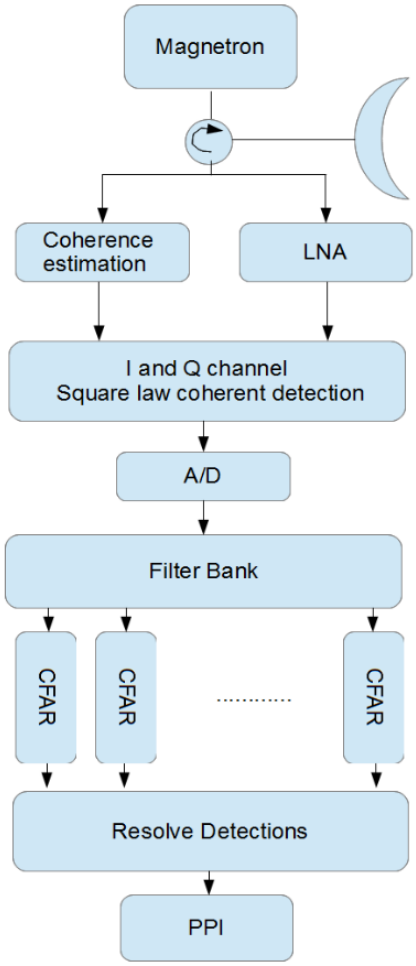


Figure 1: MTD Processing Chain

**Automatic Detection**
In the early years of radar, an operator had to monitor the PPI display, and make out detection based on experience or visual confirmations.
With the evolution of radar systems. Radar signal data have been thoroughly studied, and statistical models have been found for describing clutter models and detection probabilities for radars used in different applications.
This knowledge relieved the radar operators of the cumbersome task of detection. And digital radar systems use automatic detectors, which employs detection algorithms.
Detection algorithms may be tested, and performance in terms of probability of detection and the rate of false alarms established based on research available on radar signal theory [4], [5] [7].
The radar operator still has to make an evaluation of what is detected. But detection parameters may be changed, and tweaked according clutter conditions. Hopefully, making the detection task easier.
Skolnik [1] discusses the functionality found in automatic detectors in chapter 5.5, and list that automatic detection of radar signals involves the following:

1: "Quantization, sampling and Analog to digital conversion of the radar signal".

This sets the boundary of radar coverage resolution in terms of range cells. Each range cell has contain at least one sample, depending on detection characteristics.
The A/D conversion sets the resolution of the samples in terms of number of bits, and sets the accuracy of the radar system.
The amount of data generated each scan is determined by this stage of the automatic detector.

2: "Signal processing, integration, CFAR and Clutter map":

Signal processing involves removing as much noise, clutter, interference as practical before detection is performed. This step involves maximizing signal to clutter ratio.
As the antenna rotates, the sector may be partitioned into a number of beam-widths. One beam width contain a number of PRIs, and these may be added/integrated to improve the probability of detection.
Signal processing and integration aims to increase detection of real targets in the signal. Unfortunately clutter will still be present in the signal.
CFAR processing aim to detect real targets after integration by evaluating each sample in every PRI, making a threshold that is compared to the sample value.
The threshold is estimated, based on some a priori knowledge about the clutter situations. And the accuracy of the CFAR is determined from the statistical properties of the signal as well as the probability of detection.
The effective accuracy of a radar is given in terms of its probability of detection, $P_D$ and its probability of false alarms $P_{FA}$.
A clutter map is often realized by storing the previous scan, and subtracting it from the current scan. This removes stationary clutter. However, not all clutter is stationary. If Doppler processing is used, moving clutter may also occupy the signal frequency space.
A clutter map may also separate real targets without Doppler, moving perpendicular in relation to the antenna, from the stationary background.

## 2.2 CFAR

A CFAR seeks to perform detection at a constant false alarm rate. And generates a threshold value that is compared to a CUT. If the CUT is over the threshold value. A detection is declared.
The CFAR process may be broken down to:

$$e(Y) = \begin{cases} target & \text{if } Y \geq \alpha \cdot Z \\ notarget & \text{if } Y < \alpha \cdot Z \end{cases} \tag{1}$$

Where Z is the threshold estimate. And $\alpha$ is the scaling factor used to achieve desired $P_{fa}$.
The threshold $Z$ are adaptively generated, based on signal statistics. If the estimate is good, the false detection rate will be low. How good the estimate is, depends on how well the clutter environment can be approximated statistically.
The performance of the CFAR may be tested by applying noise which have the same statistical probability density function as the one used to create a clutter statistics model.
The time between false alarms can be evaluated giving CFAR statistical parameters.
The CFAR parameters can then be adjusted to give Constant false alarm rate for the probability distribution on which it has been constructed.
This thesis does not look into detection probabilities and CFAR performance in different clutter situations. It aims at realizing the use of OS-CFAR detection in a real time MTD radar system.
CA CFAR is also shown to provide rather good detection properties, and is far less compute intensive compared to OS-CFAR.

**Cell averaging CFAR**
In a cell averaging CFAR, a cell under test is compared to a scaled threshold value. This threshold value is generated by averaging a window of samples. The total window consist of both a leading and a lagging sub window, each separated from the CUT by a guard cell to further resolve the statistical independence of the test cell [4]. Guard cells isolate the test cell from the mean estimation.
Allowing us to define the threshold when combining the sub windows into one window:

$$Z_{CA} = \frac{1}{N_{win}} \sum_{i=0}^{N_{win}-1} X_i \tag{2}$$

Where $X_i$ denotes a random variable contained within the window.
Using the statistical signal mean as a clutter power estimate does have some disadvantages, studied by [4],[7], arguing that the CA-CFAR power estimate becomes poor when some of either the leading or lagging window cells contains clutter. This is the case where clutter is approaching the window, or are moving out of it.
In a maritime radar environment, sea clutter dominates, and sea clutter are likely to appear over large areas, spreading over many range samples in multiple PRIs.
This makes a CA-CFAR prone to false detections near clutter edges, where clutter may be falsely detected as targets or real targets may not be detected due to clutter power being magnitudes larger than power reflected from the target.
There also are problems detecting multiple closely spaced targets, discussed in [7], and multiple improvements to enhance detection performance have been proposed by introducing a greater of selection of the two sub windows as an average [4].
The general CA-CFAR may easily be modified to a CA-GO CFAR, where the leading and the lagging window sum is compared, and the largest one is used to estimate the threshold.
This will improve performance near clutter edges, without increasing the complexity of an implementation significantly.
**Ordered Statistics CFAR**
Instead of using the mean of the signal, the median or any other member of an ordered window may be used

as a threshold estimate of clutter power.
The OS-CFAR, proposed by Hermann Rohling [4] in 1982 introduced a technique from image processing for a radar application. And thoroughly studied the detection properties using the algorithms with a large number of parameters in non uniform clutter.
The master thesis of Morten Ihlen [5] 2011, further investigates detection of targets in sea clutter, as well as OS detection parameters, for the radar system developed for Radian AS.
The window layout follow as in the CA CFAR, although there are no use of guard cells in OS CFAR, see figure(2).
No guard cell is needed in the OS-CFAR, and this makes it more likely to separate out two closely spaced

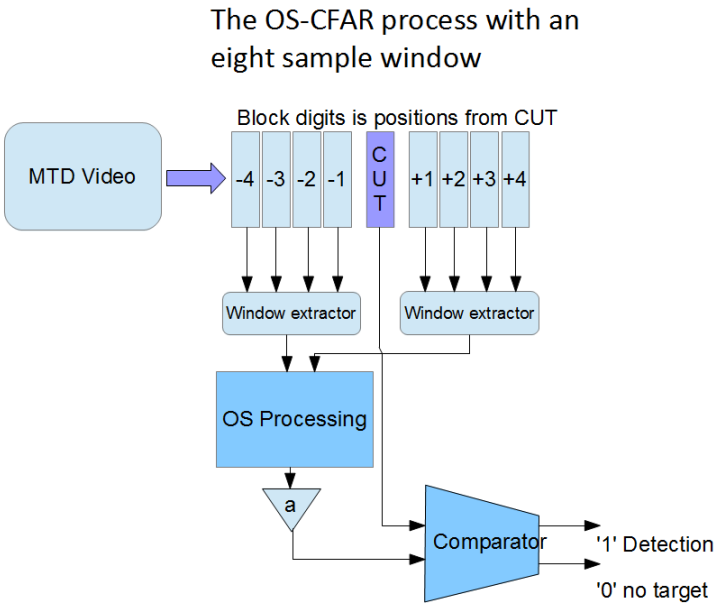


Figure 2: OS-CFAR block diagram

targets, that might have be masked or lost using the CA implementation, further discussed in [4].
In an OSCFAR, the k'th largest window sample is selected as a threshold. This threshold selections is proved to be better around clutter edged, and can discriminate between closely spaced targets. But this comes at a cost of a much more complex implementation.
If one may assume the window X stores the samples in ascending order, the threshold may be written as:

$$Z_{OS} = X_k = X(k); \tag{3}$$

There exist numerous methods as an implementation. Based on sorting the entire window, or just a part of it, until the sample of interest is found. Or an algorithm comparing samples, and ending up on the k'th sample may be implemented.
Arguments for the use of OSCFAR in a sea clutter dominated environment is given from [5] [4] and [7], where parameters for achieving good detection rates are also provided.

## 2.3 Parallel Programming

**Data Parallelism**

To be able to make the signal processing execute a program spread out over a large number of multiprocessor, each performing the same operation on a large number of samples a the same time, requires that the maximum amount of parallelism is extracted from the problem.

If the problem may be broken down to a throughput problem, a GPGPU implementation can achieve a significant speed gain compared to an algorithm sequentially executing the same operation a large number of times. [2]

Examples of such operations is matrix sums, and matrix multiplications, where a single function is applied to all elements. And a sequential implementation would be to write a single, or nested loops traversing the matrix, executing the function at each element.

**Task parallelism**

Task parallelism, is performing several serial task simultaneously, and is a way of extracting some form of parallelism from a task that may not be broken up into a lightweight kernel.

If a large level of data parallelism cannot be extracted from the problem, processing it on a GPU may not be able to speed up runtime due to a large overhead in the serial task.

In this thesis, openMP will be used to reduce the runtime in a task parallel implementation of the OS-CFAR.

**GPGPU**

The graphics processor of a computer is to provide the PC user high definition video and slender game graphics, some cards even support 3D video.

Graphics rendering are operations performed on single precision floating point numbers. And the CUDA C programming guide [2], along with [3] and [8] as well as [9] shows the throughput of a GPU processor versus a CPU on some problems solving a task using different data types.

Clearly showing the throughput of floating point operations are performed quicker on a CUDA card. The last two of the references also looks into using CUDA programming for maximizing throughput in different radar systems.

A CPU has to schedule tasks, cache data, and optimize flow control for the a lot of tasks running simultaneously on computer. Where as a GPU is specialized to perform compute intensive tasks, and are designed so that more transistors are dedicated to data processing rather than data caching and flow control [2].

In applications where a GPU is used as a co processor. Two graphics cards are typically used, and one card is connected to the monitor. While another one is dedicated as a co processor.

GPU programming may be done in a matter of ways. Matlab supports plug-ins for processing on the GPU, i.e through the 'Parallel Programming Toolbox' and 'Jacket'. Fortran, C and Python amongst others are supported through the nVidia CUDA group.

OpenCL, ArrayFire and other initiatives also provide a way of using the GPU as a co processor.

This thesis will look into CUDA C for developing algorithms where data is processed in parallel on the GPU, and discuss some special extensions incorporated in the nVidia GPU computing toolbox, which contains the CUDA libraries, code examples, CUDA C extension libraries, as well as documentation.

The code is analysed with tools available in nVidias nSight program, also available for free.

Installing the CUDA software integrates Microsoft Visual Studios 2010 with the nVidia tools, and provides a CUDA debugging environment, as well as analysis tools. Such as a profiling and application traces, so that kernel calls and API calls may be monitored and analysed.

The GPU toolkit is available for free from the nVidia homepage, to get full functionality, you have to register as a CUDA developer.

**CUDA**

In this section the basics of CUDA will be explained.
CUDA is an abbreviation for Compute Unified Device Architecture, developed by nVidia.
CUDA at its core are three key abstractions [2]. The first being a hierarchy of thread groups called blocks, The second is shared memory which is shared amongst each thread. The final abstraction is synchronization, which marks the end of a task performed by a thread. And makes the work of that thread available to all other threads within the same block.
These abstractions is for the programmer to exploit when writing a kernel doing some work on a problem where parallelism may be exposed.
The limitations to these abstractions is the computability of the CUDA card, and is the rules made for the programmer on how he may use threads to do work on one, or multiple data samples.
This maintains scalability for a CUDA programmer developing code that is to be run on different CUDA cards.
nVidia sells graphic cards over a large range depending on application and price.
A standard graphics card in a laptop may have from 16 to 384 CUDA cores, a desktop graphics card typically between 96 to 3072 CUDA cores, according to the nVidia geForce website as of may 2012.
In addition, nVidia produces cards marketed for supercomputing in their Tesla product range.

**Kernel**

CUDA C launches a 'Kernel' which is a description of the task the threads within the current thread block is to do. A kernel is launched with parameters specifying the block dimension of the kernel launch and the number of blocks which constitutes the full grid that is to be processed by the kernel.
A number of multiprocessors resides on the GPU, and when a kernel is launched. One multiprocessor is assigned to process the current thread block.
This block may come from a grid spanning a vector, a 2D matrix/surface or a 3D matrix/volume. And the block may be 2 or 3D itself.
The complete collection of thread blocks is called a 'grid'.
So if a MxN matrix is to be processed. One kernel launch process M elements. And this kernel is launched N times to ensure us the entire MxN grid of the matrix is processed.
Each block have a set of threads. And the instructions for the threads are given in the kernel.
Inside a kernel. The block index is know. So is the thread index. In addition, there are functions for getting the block and thread dimensions.
Shared memory is allocated and populated within the kernel. And this memory space resides on the multiprocessor.
When a kernel processes a block of threads, it partitions the number of threads into sections of 32 threads. Each set of 32 threads is known as a 'warp'.
Multiple warps are processed simultaneously. In addition the threads within a warp should use the shared memory. Which has lower latency, and are according to the programming guide [2] and the learning CUDA by example book [3] up to 100 times faster than access in the global memory.
Each time threads are assigned work. They have to bee synchronized at the end of the task. If threads do no synchronize after a task. There is ambiguity in whether or not the results of a thread is written or not when its result is to bee used by other threads. This is known as a race condition, and are avoided by proper use of synchronization.

**Hardware**

A GPU consists of a large number of streaming multiprocessors "SMs" which executes warps of threads assigned at compile time. Thus CUDA is built around a scalable array of multi-threaded SMs.[2]
This architecture has to include a framework for executing kernel launches, distributing and scheduling work that is performed by each multiprocessor and keeping all multiprocessors occupied with new task so that none is idling in when the kernel is being invoked.

A multiprocessor is designed to execute a large number of threads concurrently. The management of the threads is handled by SIMT(Single instruction multiple thread)[2] architecture. The SIMT is described in the CUDA C Programming Guide for version 4.1, and highlights: A multiprocessor has to create, manage, schedule as well as execute threads in a 'warp'.
A warp is a group of 32 parallel threads.
Individual threads within a warp start together at the same program address, but each thread have its own instruction address counter and register state. Allowing each thread to branch out and execute independently within a warp.
In a kernel call, groups of threads are collected into blocks. When a multiprocessor is handed one or more thread blocks to execute. It partitions the blocks into warps, where each warp contains 32 threads of consecutive thread index. And the execution of the warps is controlled by the 'warp scheduler'.
A warp is executed one common instruction at the time. If all threads follow the same execution path, the warp converges and the throughput of the warp maximized. If data dependent conditional branching occurs within a warp, the warp diverges, and each branch taken has to be serially executed. When all branches are handled. The warp may converge back to the same execution path.[2]

**Hardware control**

CUDA exploits data parallelism, meaning a simple operation is to be made on a single element, and the results and calculations have to be completely independent.
Since this operation is independent, and a GPU have a large number of streaming processors, speed gains is achieved by breaking the problem into blocks of threads which are processed in parallel.
In CUDA, one write the kernel of an operation. The kernel is often analogue to the inner computations in a nested loop.
The kernel is written as a global void function. And the number of threads per block along with the number of blocks per grid is handed as input parameters to the global void kernel function.
The computations are not performed in an ordered form, but rather areas of the matrix is processed at any given time.

**Programming with CUDA**

The CUDA programming model assumes a system consists of a host and one or several devices.
All devices are physically separate from the host, and have its own memory space and processing unit.
The host is the CPU, which have access to the computer RAM. A device is in this case a GPU, and a computer may have several CUDA devices available to work with.
Kernels may only operate on device memory, so all data that is to be processed by a kernel have to be copied from host memory to device memory ($CPU->GPU$) RAM.
Data may be allocated as either linear memory or CUDA arrays.
Linear memory corresponds to the standard C/C++ array memory, where as CUDA arrays are opaque memory layouts optimized for texture fetching involved with graphics rendering of 2D or 3D objects.
Within a kernel as mentioned earlier, different memory may be used depending on application.
The global, local as well as texture memory is memory located on GPU RAM, where as shared memory resides on the multiprocessor chip.
Since CUDA is a C/C++ extension. Well known C API functions as 'malloc()', 'memcpy()' and 'free()' amongst others are available for handling device memory.
These API follow the naming of their C counterparts, one just have to add 'cuda' and begin the API function call with an upper case letter. I.E; cudaMalloc().
For a list of all API function, see the programming guide [2].
The linear memory address space is either 32-bit or 40-bit, compute capability 1.X or 2.X respectively. And are allocated by the function cudaMalloc() and freed with cudaFree().
But may also be allocated through cudaMallocPitch() or cudaMallco3D(), depending on dimensions of the problem.

Using the dimensional allocations ensures best performance accessing row addresses, or performing copying of 2D arrays or other regions of device memory, compared to a flattened multidimensional array, according to the CUDA 'Best Practices Guide' available alongside [2].
As mentioned, shared memory is to be about 100 times faster than global memory, and any opportunity to replace global memory access with shared memory access should be exploited.
If these threads may need access to data outside the block. This data may be loaded to the shared memory.
The process of fetching data for shared memory may also be done in parallel.
Threads are spawned which copies a chunk of global memory to shared memory. Any sample within the grid may be fetched to the shared memory of the current thread block.
By synchronizing the threads after reading out global memory. These variables become available in the shared memory. And the threads are freed, and ready to be set to perform the next parallel task.
The shared memory is limited to 48K bytes in compute capability 2.X, and 16K bytes in 1.X [2], tables extracted and added to the enclosure.
This means that 12 000 floating point numbers may be stored in the shared memory of a CUDA 2.0 multiprocessor.

**OpenMP**

OpenMP is an API that handles parallel programming using multiple processor cores.
Open MP is open source, and resources as tutorial, white papers and other documentation is widely available at their home page www.openmp.org .
OpenMP is a set of compiler directives, which tell the compiler where parallel regions in the code starts.
How these parallel regions are executed, as well as how work is shared amongst threads to complete execution is also specified through openMP pragma directives.
Open MP lets the master thread in the program generate a team of threads to aid processing.
After a parallel region have been declared, one have to specify how data should be treated by the threads.
This is done by making clear to the compiler what data is to be private and what is to be shared amongst threads.
Whether the data is initialized before being used by a thread or not, as well as the thread execution order.
How large a chunk of the problem have to be processed sequentially? How should work be assigned threads, and how is the task schedule?
Open MP will be more thoroughly explained in the design section (4), and these questions posed will be answered.
Open MP can be used with the Visual Studio Compiler without any configurations.

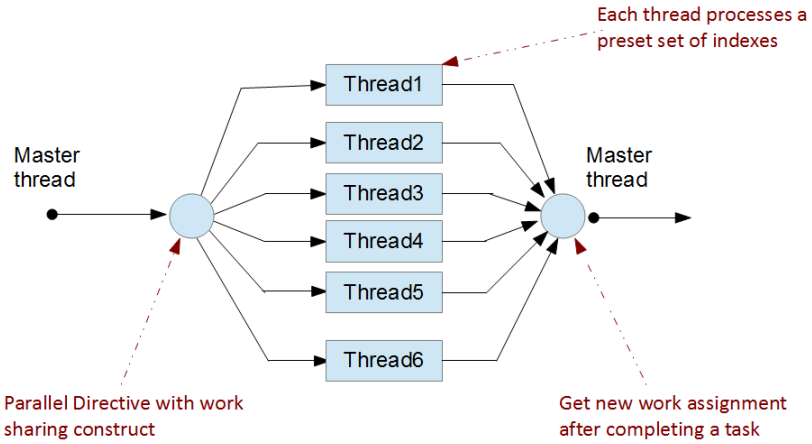


Figure 3: OpenMP parallel construct

## 2.4 Selection Algorithms

Time Bounds for Selection[10] introduces a group of algorithms called PICK. Although this set of algorithms was a dead end in terms of a parallel programming algorithm. It presents a good explanation of the nature of the problem.
Basically, one wish to find the k'th largest sample in a set of R distinct numbers. This problem may be traced to the realm of sports and tournaments [10]. How many games have to be played to find the best competitor? And is the player who lost the finale really the second best competitor?
Certainly not in a knock out tournament. But then how does one separate the ones who clearly is not among the top half of the competitors from the best half? How many matches have to be played to establish some kind of relation between the competitors. And are there clear indications of who is the best or worst early on?
Firstly one have to find a way to establish a ranking. Then the order of the matches, as well as how many matches that may be played at the same time, to execute the tournament as quickly as possible.
The efficient way of arranging a tournament largely depends on the the number of contestants. As well a how many other ranks other than the largest have to be established.

This allusion may be directly ported to the OS-CFAR. First, lets assume a window is available for the current CUT. A selection in this window is to give a better estimate of the mean clutter power, as compared to the window mean found by CA-CFAR.
If clutter is present within the window. It will accumulate in one edge of a sorted window. As clutter have high signal power.
A Naive way of approaching the ordered selection problem is sorting all the samples before selecting the k'th largest sample.

This gives complete knowledge of the order of all the elements, which is redundant. As the OS-CFAR is only concerned about the k'th largest element.
By reducing the number of elements that is ordered, or finding only the k'th largest element. The operations for each CUT may be reduced.
Selection algorithms for finding the k'th largest element in a really large array is a problem well studied [10] and [12]. As are application of GPU for speeding up these algorithms [12] , both the naive implementation using sorting, and other implementation.
The C++ standard template library includes an algorithm called "*nth_element*" which inputs a set of numbers. And ensures that the n'th largest number is at its correct position. Without caring about the position of the others samples.
However. In the OS CFAR, a selection is to be made, but from a rather small set of numbers. Ranging from eight to 24 samples.
And this operation is to be performed on a large number($10^5 -> 10^6$) of test cells generated each scan.
In this thesis, a number of different approaches is made.
The Naive implementations was investigated in the project report [6]. In this thesis, results are compared to the algorithm developed there. And that was found to be a lot faster than the naive approach.
Three implementations are tested in this report.
Each implementation exploits different properties of the selection problem in order to improve throughput, and thereby decreasing the runtime of the problem.

# 3 Signal processing Requirements

## 3.1 The MTD radar demonstrator

**The PCI Express card; What is available**
The PCI express digital receiver board performs square law detection on coherent radar data.
The output at the square law detector is sampled in two components. An in phase component, and a quadrature component channel, which is 90 degree out of phase in relation to each other.
The sample rate is 40 MHz and the resolution of each sample component is 14 bit[14].
As the antenna rotates. A large number of pulses are radiated and each of these contains a given amount of range samples.
After one scan. The entire data matrix containing samples of the received echo signal is available in the RAM. And the signal is ready to be processed.
To be able to realize a MTD system with functionality described in the theory section in a real-time system. All steps in the signal processing chain have to be executed before the next data matrix from a scan becomes available.

**MTD demo computer specification**

The MTD Demo computer is equipped with an AMD Phenom II X6 6 core processor, that is clocked at 3.3GHZ.
The computer have 8GB ram, a 128GB SSD drive, and two CUDA enabled graphics card. One GT 430 graphics card, which is used with the monitor. And a Zotac 1GB GTX 560 GPU, used as a co processor.
The GTX 560 Card and GT 430 cards have the following specs:

| **Hardware** | **GT430** | **GTX560** |
|---|---|---|
| CUDA Cores | 96 | 336 |
| RAM | 1GB | 1GB |
| RAM Clock | 1600 MHz | 4008 MHz |
| Graphics Clock | 700 MHz | 820 MHz |
| Processor Clock | 1400 MHz | 1640 MHz |
| Compute Capability | 2.1 | 2.1 |
| Number of Streaming MPs | 2 | 7 |
| Shared Memory | 48K | 48 K |
| Max threads pr block | 1024 | 1024 |
| Max instructions pr kernel | 512 M | 512M |
| Power Consumption | 49 W | 150 W |
| min PSU rating | 300 W | 450 W |

The computer is running a 32-bit operating system, Windows Embedded 7, and the following programs are installed to develop solutions for the thesis:

-Mathworks MatLab R2012a
-Microsoft Visual Studio 2010 Ultimate
-nVidia GPU Computing SDK 4.1
-nVidia Parallel Nsight 2.1

**Concerning the system**
The raw radar video signal available from the PCI card is a matrix containing the total number of range samples time the number of PRIs elements. In the signal data processed in this thesis $350x290x20$ elements. Each element is a sample of the measured return power level of a return echo coherently referenced the transmitted pulse.
This signal is sampled by the PCI card to contain the magnitude of an In phase and Quadrature component both represented on a 14 bit signed format.
By sampling both In phase and the Quadrature component, we are in practice sampling the signal, and a orthogonally shifted version of the signal.
Coherence is established, and spatial phase shifts between adjacent samples may be related to a doppler velocity, by looking at the signal in the frequency domain.
By looking back at the automatic detection signal flow from figure( 2), the processing requirements may be found, and the framework for developing algorithms using parallel programming techniques established.
The size of the data generated each scan is approximately 4-5 megabytes, depending on the sector size.
The first eight samples of each PRI are headers, holding information of sector parameters, etc..
The header from each each file is stripped. And data is decimated by 3 samples in rangeand then filtered by a filter bank. In the filter bank, the PRIs of the MTD signal are decimated by 10, all PRIs in the raw video are needed for the filtering process
With a 40MHz sampling rate, the range resolution is approximately 3.75 meters/sample. This makes the range coverage of the MTD radar:

$$350[samples] \cdot 3(Decimation factor) \cdot 3.75(\frac{[meters]}{[sample]}) = 3950 - 3940[meters] \tag{4}$$

And is to be achieved when a 120 degree sector is covered.
The raw normal data is filtered by a Doppler filter bank with 20 velocity channels. This process makes the video signal grow from a single channel, to 20 channels, thereby increasing the amount of data by a factor 20.
By removing as much of the signal that is not needed, the decorrelation between samples are increased. Samples are highly correlated, and making them less correlated, reduces the redundancy of the CFAR thresholds, since all targets that we wish to detect most likely is present over multiple samples.
The size of a typical MTD video data struct is ca 7.1mB, when decimated by 10 in azimuth and 3 in range. This data struct contains approximately 2 million cells under test. And each cell under test have a unique threshold that have to be calculated for it.
The threshold evaluation is the main bottleneck in the processing chain. Especially the OS-CFAR algorithm, which have a high complexity, as it cannot be broken down to a simple arithmetic evaluation that is applied to all CUT.

## 3.2 Testing and developing the implementations

**CUDA GPU Computing**
CUDA integrates with Visual studio or any other OS compiler, through the CUDA SDK, toolbox and nSight packages.
This makes it able to start new CUDA projects as any other code projects i Visual Studio.
All CUDA algorithms are developed in a C style testbench, where data is passed from a main file, through as gateway. Making a porting implementations to mex easier.
In the main file. Synthetic radar data is generated, and the data size of the test signal is made to be the same of the MTD video.
All CUDA code have to be written in .cu files, and are compiled with the nvcc compiler.
Unfortunately, there seems to be no easy way of integrating the nvcc compiler with the mex environment in

Matlab. As running the :MEX -SETUP does not find the nvcc compiler.
However, advanced profiling and code analysing tools comes with the nVidia software. And the CUDA implementations are profiled using these, and compared to the C code using openMP in mex processing real MTD data.
The ordered statistics algorithm developed for CUDA is also implemented in regular C using openMP, and tested with real signal data to verify that the thresholds are matching those of the other implementations.

**OpenMp**
OpenMP may be integrated with mex, making it easy to test algorithms using OpenMP multiple core parallel programming.
The run time of the algorithm, can then be tested using real MTD signal, and Matlab Profiler.
This will get the actual execution time of the algorithms, but Matlab Profiler cannot be used for other than getting the time spent in total by the mex function.
OpenMp functions are also tested in a C test bench with synthetic MTD data. By using the profiler that comes with the 'Ultimate' version of Visual Studios, each line of C code may be timed, and the code further evaluated.

**MEX**
C code containing openMP may be compiled with mex, but this have to be activated for the compiler, or flagging the openMp compiler flag explicitly.
The project assignment [6] gives an introduction to the mex environment, and presents a way to use it by illustrating some examples.

**C Testbenches**

When working with the the CUDA extension of C++ towards GP-GPU programming, the code executing on external devices have to be built with nVidias nSight nvcc compiler. This compiler compiles the .cu files into executables that contains instruction on how the code is to be run by the host, utilizing the device efficiently.

However, the nSight nvcc compiler cannot be set up to be used as an independent compiler in Matlabs mex setup. So the problem arises, how should mex filed be compiled so that they may be run from Matlab.

There exists at least two solutions. One, manipulate Matlab system files, and invoke the nvcc compiler through a hack. CUDA .cu files may now be compiled in the Matlab command line by the nvcc compiler. Or alternatively build the CUDA code in Visual Studios 2010, creating .dll files which may be linked to be executed through normal C/C++ code, compiled in Matlab command line.

For the future of the project, and as a thought for others working with it later on. All CUDA code should be built as dll, and linked to the C/C++ code.

This allows for easier implementation of code in the mex environment, and makes it easier to move the project between computers with similar system architecture and OS.

Any way, the .cu file have to contain the mex gateway function, passing vectors from matlab and into the CUDA C environment. Also the "mex.h" header file have to be included in the .cu file.

Matrices passed from Matlab is of type mxArray. A raw pointer of the correct data type have to be cast to access/manipulate the matrices. And raw pointer can be used to copy the data from host and to device.

If the elements within the mxArray are scalars, i.e only floating point number or integers. The data is stored in memory as the familiar C/C++ array.

However if each row in the mxArray is a string. One have to use column wise indexing. That means, adjacent elements are spaced by the total number of columns in the mxArray. The figure(4) illustrates how the MTD Video is stored as a multidimensional array, where each channel is represented as sheets.

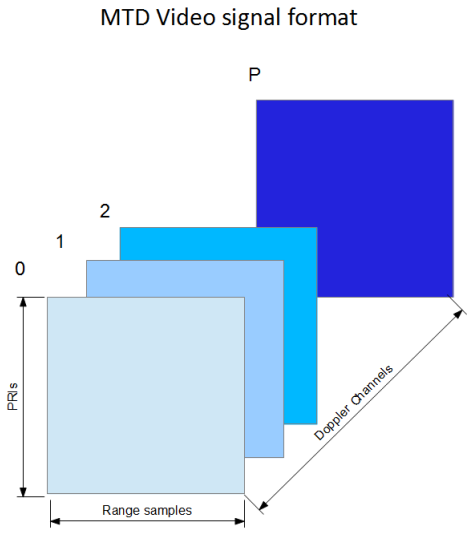


Figure 4: The MTD Video Format

## 3.3 Extracting parallelism

**Cell averaging**
Cell averaging cells may be done using reduction operators, as summing elements is quite rudimentary arithmetic operation a processor has to perform.
The CA-CFAR extracts two windows. The result of the summation of the threads within one sub window is accumulated by reduction.
Since summation is such a low level operation. A high level of parallelism is already present in the problem. And a CUDA implementation is straightforward; and a good way of starting and getting known with CUDA programming.

**Ordered statistic selection**
Each CUT has its own window. Neighbouring CUT share a lot of samples in their windows. If one imagines the window sliding over each test cell within a PRI. The windowing process becomes strongly connected. And each PRI have to be executed in sequence from first CUT to the last CUT.
The algorithm developed in the project assignment generates an initial sorted window for the first CUT. Then the positions of the samples in the sorted window, relative to their original position is stored in a separate array.
Then the window starts sliding over the samples, two samples are dropped. And the positions of these samples are found by searching through the already sorted array to find the position of the two new samples. This is done for each CUT, in order.
Execution of a CUDA kernel however is not performed in predetermined order. Warps are launched in random order. So strongly connected calculations have to be broken up in order to efficiently implement the algorithm.
To extract parallelism, each CUT evaluation has to be made as general as possible. There should be no dependencies of anything that happens outside the current CUT evaluation. As this will create conditions where values may be read or written by different CUT evaluations running simultaneous in parallel. And wrong threshold values are generated.
The problem has to be broken down to a general algorithm, running the selection of a threshold in fixed number of operations. How many CUTs, as well as which CUTs that are to be executed in parallel is set at the launch of the kernel.
The entire grid which is to be processed contains all the test cells in the MTD video signal. Where this signal is stored in a three dimensional MxNxP matrix, where P is the number of doppler channels. The same operation is to be executed on typically one million test cells.
By partitioning this grid into a number of thread blocks that are to be processed simultaneously each kernel launch in a smart manner, thread cooperation may be fully utilized, and the shared memory of the device may be used efficiently.

# 4 Design

## 4.1 CUDA

**Cell Averaging CFAR**
The Cell Averaging CFAR estimates the mean signal power by extracting a window of the signal, and averaging it.
This may be done in terms of reductions, accumulating sums or by convolution or a sliding window.
This is the classical cell averaging algorithm, but it may be modified to be more dynamic, and to limit some of the disadvantages the classical implementation does not take into account.
By modifying the CA-CFAR to a CAGO-CFAR [4], detection capabilities may be improved.
The window extraction can be decomposed into extracting one leading and one lagging window. Each of these windows are of length N/2. For each CUT. The average of both windows is calculated, and a comparison between the averages performed, so that the largest one is used as an estimate.
This will improve performance near clutter edges, and closely spaced targets, as the leading window will contain clutter samples with large magnitude while the lagging windows does not[4].
Clutter samples as mentioned earlier will have a significantly higher signal power than clutter free samples. If a large window is used to estimate the mean clutter power, a mean estimate may be to low when both clutter and non-clutters samples are in the estimate. Creating false alarms.

**Application Kernel**
The kernel is to operate on the total 'surface' or 'volume' of the matrix. This may be achieved if the kernel indexes spans the dimension of the matrix, meaning a 2D matrix requires both x and y dimensions, a 3D matrix x,y and z dimensions etc. The kernel code processing a matrix in example a of an $A[N][N] \cdot B[N][N]$ matrix multiplication have to perform an operation on N*N elements of a. This operation also have to load one row from A, and multiply it with one column of B.
Threads launched in a kernel may, as mentioned earlier have access to any element within the grid.
That means any value may be moved to the shared memory to allow faster memory accesses.
This have to be extensively used in a CUDA algorithm performing ordered selection written in CUDA. As comparing elements require intensive memory lookups.

**Ordered Statistic CUDA Algorithm**
The Ordered statistic algorithm implemented in CUDA have to be able to complete a selection in a given number of operations.
This will ensure that threads run concurrently. Avoiding branching and the total number of operations varying from thread to thread.
The way SIMT is used in this assignment is that one thread is responsible for generating a threshold sample if it is a valid CUT.
The procedure the thread has to follow to find the sample of desired rank is generalized. So that each thread runs the same operation. And the desired rank is returned from a register holding the end value of the thread run. Of which is the largest sample found in the last run is the k'th largest sample.
The tournament allusion from the theory section is difficult to implement in a SIMT efficiently when scalability between different window lengths is desired.
A branching solution soon becomes complex if one thread is to perform this work. As the state of the algorithm is not known in advance.
A lot of temporary states, coming from conditional branching,as well as considering that changing the k parameters, severely changes the execution paths. Renders a tournament type algorithm far to complex to be of practical use, although this come at the expense increasing the total number of operations to be performed each run.
The implementation is discussed in 4.2

The CUDA CFAR is built around a window size 16 and the k'th sample chosen to be the 0.75 largest sample. A common value suggested by Rohling [4] and Ihlen [5] for being a good estimated clutter power for a detection in a radar system. Giving sufficient $P_D$and $P_{FA}$.
The 0.75 value is the 12.th sample in a 16 samples window. And the 6th sample in a eight sample window. And it is common to use up to 24 sample windows.
These are common window lengths which is desired to use in order to achieve a sufficient detection rate in the MTD system.
The number of operations required for the algorithm to find the k'th largest neighbouring cell algorithm is exponential. See table in section 4.2.
This algorithm finds the k'th value without any estimation of that value based on statistics. Alas it searches through all the same window samples four times, ignoring the samples that are currently found to be amongst the largest samples.

## 4.2 OS-CFAR CUDA Implementation

Copy data to device, allocate output, and prepare kernel launch.
The input to the kernel represents the total number of PRIs that is to be processed (Azimuth x noFilters) where one the Array is partition so that one PRI constitutes one thread block.
This launches all the range samples within a given PRI in a kernel invocation. Spawning one thread for each range sample within the PRI.
From a kernel point of view; one PRI block have one thread per range sample.
When invoking the kernel with its launch parameters. the kernel have to find a threshold corresponding to a range sample if it is a CUT.
Inside the kernel, each thread writes its value to the shared memory. Which means that the range samples in the current PRI is loaded to the shared memory.
This operation is done in parallel. As all threads write their value to the shared memory.
Then the threads are synchronized.
The entire PRI is now available to be accessed by any thread without the latency associated with the global memory.
The proposed algorithm uses two temporary arrays that resides in the shared memory. One array holds the position of the samples larger than the k value. The other one the position of the large samples, so that they can be excluded from comparisons.
From a thread point of view; One thread is assigned to work with one window.
The window is a section of the shared memory.
The thread iterates through the window, and stores the value and index of the largest sample. At the end of a run, the largest sample along with its position is written to the two temporary arrays.
These are also in the shared memory. And one thread have its private section of these arrays to work with, so that no thread works on shared temporary values, eliminating race conditions.
When the largest sample is found. It is not being excluded from the search space.
Then the thread searches through the window again, and the second largest sample is found.
This process is repeated until the k'th largest sample is found.

$$NOP = OP_N + OP_{N-1} + OP_{N-2} + \cdots + OP_K \tag{5}$$

If N is the window size and OP is the operations required each run.
This is not a particularly elegant solution, as the number of operations increases with window size and k parameter according to the table below.

| Window | K = 0.75 | NOP |
|---|---|---|
| 8 | 6 | 18 |
| 12 | 9 | 38 |
| 16 | 12 | 65 |
| 20 | 15 | 99 |
| 24 | 18 | 140 |

The algorithm from a thread point of view is illustrated in figure(5) on page 25.

## 4.3 OpenMP Implementations

**The algorithm from the Project assignment**

The algorithm developed in the project assignment attacks the problem as optimizing a sequential algorithm for a larger task.

An initial window is created and used to estimate the first threshold value. Then as the window slides to the next CUT, the two oldest samples in each sub-window is dropped. And two new samples are inserted at their correct position in the already sorted window.

This is done for every CUT, for every PRI for all the the Doppler channels.

The core of the algorithm is optimized to speed up the process of picking a threshold sample for each CUT in one PRI.

This was the initial algorithm, and the first try at achieve a speed up in terms if runtime from the original MatLab implementation.

This code was developed on laptop, and was found to decrease the computational time of the problem to some extent. [6].

An effort investigated in this thesis was to further speed up this code, by spreading the task over multiple CPUs. And the goal is to make the task of selecting thresholds independent for each PRI.

So that multiple PRIs may be computed simultaneously by different cores, even though each PRI represents a problem that have to be executed serially.

The original algorithm did the threshold selections in different unique steps. It also stored a unique window for each PRI.

The first step of the old algorithm was to extract a unique window for each PRI in the doppler channel, and sort this. This action was used for the first CUT, and this made the first threshold for all the PRIs.

Then for each CUT in the PRI, two samples was dropped, and and two new samples were added at their respective place within the sorted window generated in the first step of the algorithm.

Such an algorithm has to track the positions as well as the values of all the window elements.

This algorithm was developed when working with MTI data. This is a radar Doppler data with only one channel, holding the entire frequency spectrum of the return echo.

Later it was decided MTD processing was to be used to get more functionality and more advanced signal processing.

Currently, 20 doppler channels are used. Making the number of number of CUT 20 times larger than for the MTI case without decimation.

To drop the number of CUT, the signal is decimated by 10 in azimuth. Implying that only every 10th original PRI of the full MTD video is processed by the CFAR.

Parallel computing is an effective tool for decreasing the runtime of code, or to increase the size of the problem that needs to be solved.

OpenMP described in the theory section was studied to be used for speeding up the algorithm over a PRI where a low level of data parallelism could be extracted. Due to the fact the state of the window changes between adjacent CUTs. And the kernel has to be executed ordered.

This makes this kernel useless as a CUDA Data parallel kernel, but the algorithm may be spread over multiple CPU's in a multi core CPU.

. OpenMP is a collection of compiler directives, library routines and environmental variables that is incor-

porated to the C/C++ workspace.
This code snippet shows how the openMp clauses are used:

```
//The Parallel construct
#pragma omp parallel
        shared(data,[chunk])
        private(index,tmpArray)
        firstprivate(declaredVarable)
```

and:

```
//Work Sharing Construct
#pragma omp for
        schedule(schedParam,[chunk])
        ordered
```

The schedule parameter describes how the loop iterations are to be divided between the threads, and how many iterations are to be carried out by each thread spawned at the parallel construct.
- 'ordered' tells the compiler, each thread have to execute the processing of a chunk in order. So that serial problems may be processed with a single thread.
The Private data is private to each thread. Where as shared data is data stored in a memory location, globally available to all threads.
Private variables are loop indexes, or stack variables in subroutines called from the parallel region of the code.
These constructs provides control over the data environment in the parallel construct. And defines which variables are visible to all threads in the parallel section, as well as which variables will be privately allocated by each thread.
When using OpenMp and shared memory, it is important to be aware of race conditions,as these may occur here too as mentioned for CUDA SIMT.
In the OS-CFAR algorithm. Stack variables are made private, and the MTD data, as well as detection matrix is shared.
Each thread is to process one PRI. As this is the length of the sequential problem.
Meaning the chunk size is made to be the same as the number of CUT.
Dynamic scheduling is used so that once one thread is finished working on one PRI. It is assigned another one, independent of the state of the other threads. It does not have to wait until the other threads have finished, reducing overhead.
The the serial part of the code have to be specified, and the indexes that are evaluated ordered is sectioned.

**STL implementation**

An implementation using the *n_thelement* STL function is created.
In this allocation, a window is allocated as a stl::vector container. Each thread makes a local copy of the window when doing work.
And one thread is used for each CUT.
The first step is to extract the window samples for the CUT. Then the 'stl' function is called. And the threshold is written.
The 'stl' function is within a nested loop, and the order of execution of this loop may be random.

**CUDA algorithm in openMP**

The algorithm developed for CUDA is ported to run openMp. The difference is that the entire PRI is not processed in parallel. And it does not use shared memory.
One thread does the work of selection for one CUT. So each thread creates it's own copy of the two temporary arrays.
The order of execution is irrelevant in the CUDA algorithm. And threads are scheduled dynamically to ensure no threads are idling.

## 4.4 Implementations that was discarded

**The C++ STL and Thrust GPGPU STL**

As a short cut around CUDA, an OS-CFAR thrust implementation was considered.
The project code was ported to C++ STL, in order to get experience and an overview of the STL library containers and functions.
This evolution of the rather standard C code of the project towards C++ stl increased runtime of the algorithm.
openMP seemed unable to speed up the implementation.
The thrust contains sort functions and vector containers for GPGPU processing and is an extension of CUDA.
A thrust [13] implementation would allow implementing a Naive algorithm running on the GPU to be used as a reference.
However, due to lack of knowledge over how the thrust and CUDA actually uses the GPU. This short-cut only showed that the main principles behind CUDA had to be understood in order to use Thrust efficiently.

**Arrayfire**

ArrayFire is an open source API for parallel computing that integrates with the system C/C++ compiler.
Arrayfire also have built in sort functions, and was considered for a Naive implementation of OS-CFAR.
Accelereyes, the creators of Arrayfire also sells a plug-in for Matlab called Jacket, for GPU computing.
Arrayfire uses a different syntax as C, although functions are written in a C environment. As Arrayfire is just an API, that runs CUDA code. It is also a little 'behind' the CUDA toolkits provided by nVidia.
The version of Arrayfire available as of march 2012 only supports CUDA Toolkit 4.0, where as version 4.1 is the newest provided by nVidia.
This made Arrayfire unable to use the newest stable GPU drivers from nVidia. And moving from working in CUDA to Arrayfired, required the Graphics drivers of the system to be rolled back to a previous version.

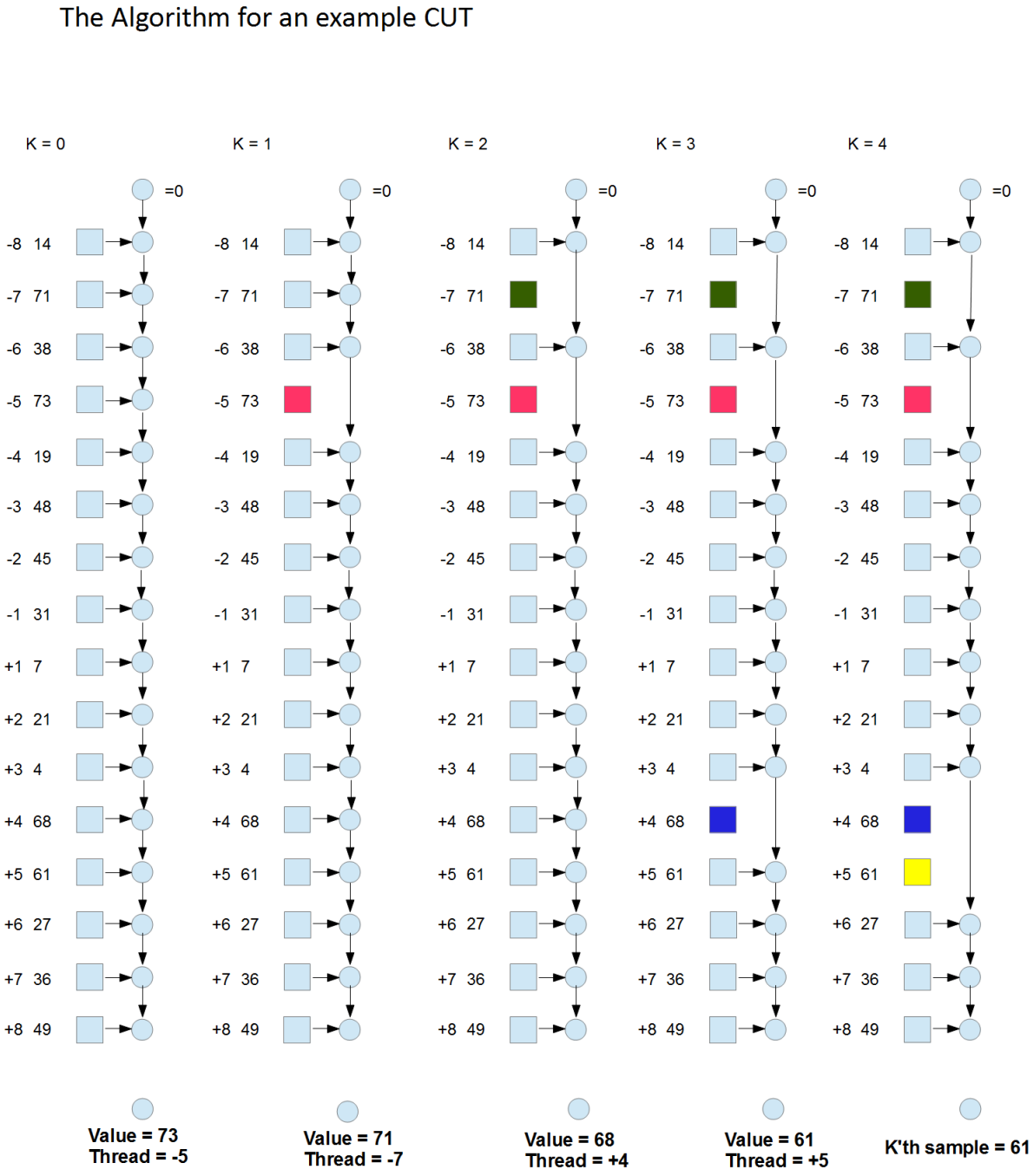




Figure 5: Cuda algorithm with example data and thread indexing relative CUT

# 5 Results

## Runtime of the algorithm

The OS-CFAR Algorithm implemented in CUDA have the following kernel run times on a array of 400x300x20 samples.

| OS-CFAR | | | CA-CFAR | | CAGO-CFAR | |
|---|---|---|---|---|---|---|
| **Window** | **K value** | **Run time[ms]** | **Window** | **Run time[ms]** | **Window** | **Run time [ms]** |
| 8 | 6 | 8.0 | 8 | 3.02 | 8 | 3.39 |
| 12 | 9 | 15.4 | 12 | 3.94 | 12 | 4.30 |
| 16 | 12 | 25.1 | 16 | 4.85 | 16 | 5.21 |
| 20 | 15 | 41.0 | 20 | 5.77 | 20 | 6.13 |
| 24 | 18 | 50.9 | 24 | 6.68 | 24 | 7.05 |

The table only shows the execution time of the kernel. Before the kernel is run, data is copied to the device, and the device have to schedule the tasks ahead.
After the kernel is executed. Results are copied back. And the memory allocated by the CUDA function call have to be cleaned up
All these operations are handled by the CUDA API, which may be timed using the nSight monitor.

| **Run** | **Capture Duration** [s] | **cudaMalloc** [ms] | **cudaMemcpy** [ms] |
|---|---|---|---|
| 0 | 0.11 | 70.98 | 30.47 |
| 1 | 0.13 | 92.87 | 29.78 |
| 2 | 0.11 | 72.06 | 29.61 |
| 3 | 0.12 | 82.65 | 31.62 |
| 4 | 0.12 | 68.25 | 34.23 |
| 5 | 0.11 | 73.69 | 32.36 |
| 6 | 0.11 | 63.43 | 32.16 |
| 7 | 0.10 | 73.29 | 31.29 |
| 8 | 0.12 | 69.51 | 32.45 |
| 9 | 0.11 | 72.77 | 30.40 |
| 10 | 0.11 | 68.28 | 32.23 |
| 11 | 0.10 | 72.86 | 29.92 |

The table shows the total time of the launch, which compromise of the runtime of the algorithm, as well as the most time consuming API routines.
The mean of the API function time is

$$T_{API} = 73.2[ms] + 31.4[ms] = 104,6[ms] \tag{6}$$

The runtime of the algorithm is estimated by using:

$$T_{ALG} = 104,6[ms] + T_kernel[ms] \tag{7}$$

Which might be a fair estimate, but using .mex may add some overhead to the runtime of the algorithm, as .mex data are passed as mxArrays. Which may be converted to regular array before cudaMemcpy may be run.

Different implementation of the OS-CFAR algorithm run over multiple computer cores are shown in figure(6). Where the 0.75 value of the window lengths are used as K value.

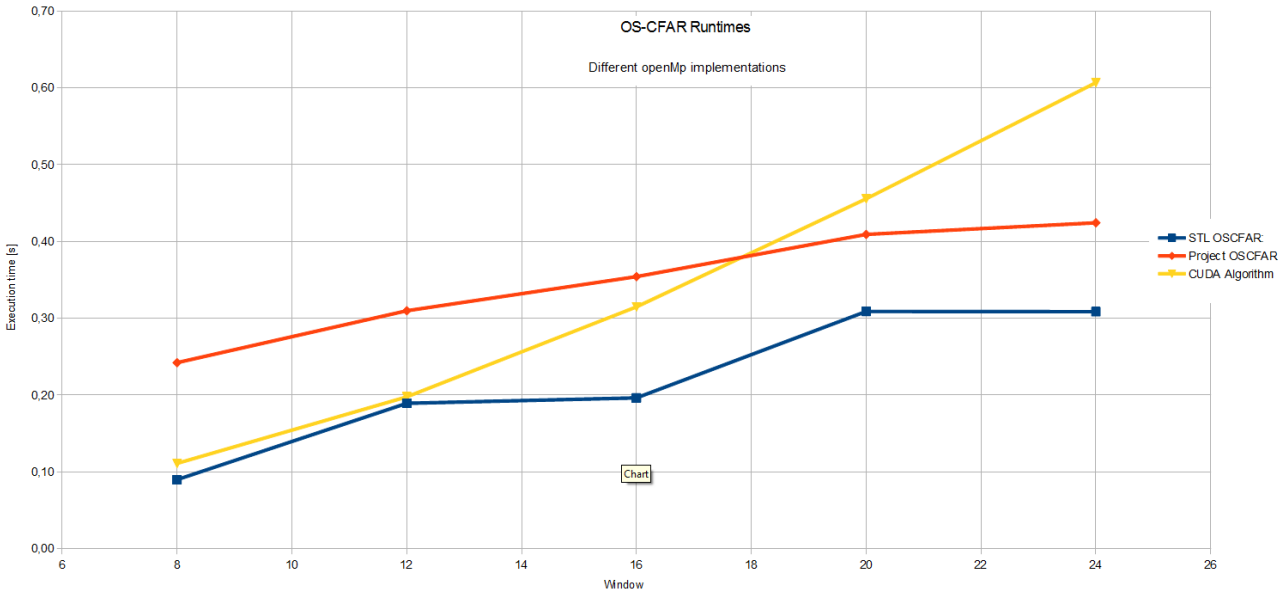


Figure 6: Matlab profile time with openMP enabled

The runtime of the different implementations of ordered selection not using openMP is shown below in figure(7), so that the impact of openMP may be illustrated for the different algorithms.
To put the CUDA runtimes in perspective, they are plotted along with the fastest of the OS-CFAR runs. It is worth noting that the CUDA run is not timed with Matlab profiler.
The code that is timed is equivalent to the mexFunction being timed by the matlab profiler. The CUDA data is of dimentions 400x300x20, while the mex algorithms are sized 352x290x20.
Although, figure(8) illustrates the flat runtime of the CUDA kernel performing OS-CFAR.

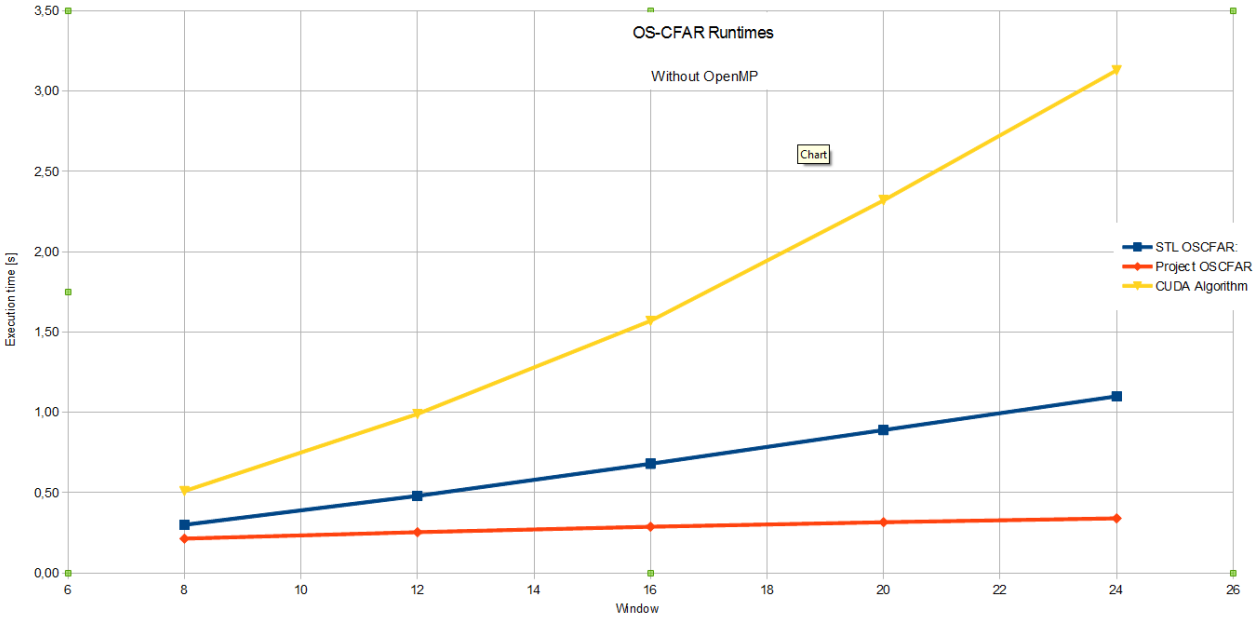


Figure 7: Matlab profile time with openMP disabled

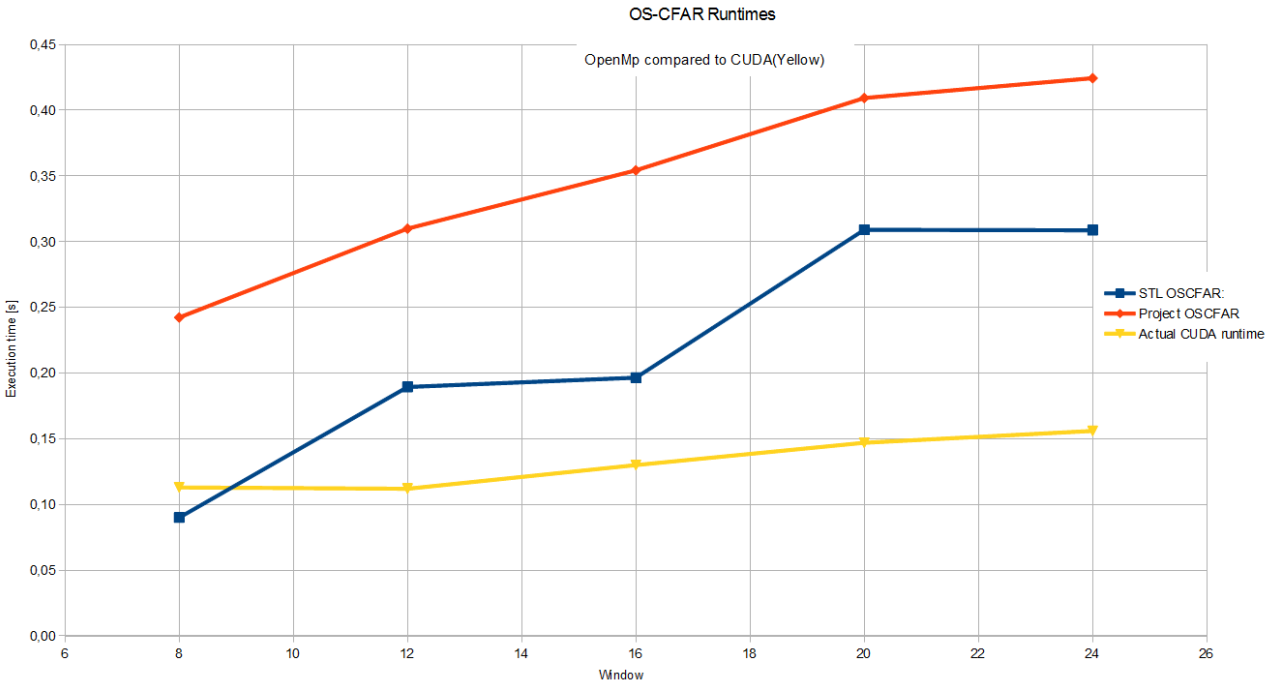


Figure 8: Matlab profile time OpenMP and CUDA

# 6 Discussion

## 6.1 Algorithms

**CUDA Algorithm**

This implementation not really an effective way of making an ordered selection in terms of the number of operations required by the algorithm to find this value, but it works with the current graphics card having a the CUDA specifications derived in 4.1

The current implementation suffer from poorer runtime if the number of range samples within one PRI becomes to large or if the window becomes too large.

Since samples are decimated in range. There really does not exist any reason too have window lengths larger than 24. A 24 sample window spans a range of 135m prior and after each CUT in this radar system.

This system uses a one dimensional window. Other applications may want to use two dimensional windows for the ordered selection of a threshold sample.

The efficiency of the OS-CFAR CUDA algorithm decreases if larger multidimensional windows are to be used.

Since the shared memory occupancy is directly linked to the number of samples within the window, redesigns have to keep in mind the capacity of the shared memory. Especially if they are to bee backward compatible and to be used on a large number of GPUs.

Depending on application, different cards have to be chosen.

Since the algorithm is based on comparisons. All operations are lookups in memory, and depending on a decision. One sample may be written as a temporary largest sample each time the algorithm is executed.

By utilizing the shared memory which resides on the multiprocessor. Each memory operation is performed in small number of clock cycles according to the programming manual [2]. Where as a lookup to external memory takes up to more than 400 clock cycles [2].

That means that the algorithm may be run efficiently if the CUDA card has enough multiprocessors, and sufficiently large memory to meet the specifications of the radar system in terms of range.

Since the current implementation stores the PRI and a copy of the N-K+1 largest samples and their position for each cell under test in the shared memory. Scaling of the algorithm have to consider the available shared memory of the CUDA card.

This assignment implemented the CUDA algorithm on the CUDA enabled nVida GTX560 card. This card have 7 streaming multiprocessors, executing the algorithm on 7 PRIs in parallel.

A block may as previously mentioned consist of threads spanning over arrays of more than one dimensions, but as the problem appears in this assignment. They are one dimensional.

The OS-CFAR algorithm uses one thread per CUT. This was a decision made on the basis of all operations required to make one selection.

If one thread was to process multiple CUT, fewer threads would be spawned each run, and the occupancy in terms of shared memory may be increased.

Currently the algorithm only uses 6000 32-bit variable allocations of the total 12000 available in the shared memory of a compute capability 2.x card, This is when the window length is 24 and the k'th sample is the 18.th largest.

When the window length is 16 and the 12th largest sample is used, the shared memory occupation is 4400 32-bit variables.

Increasing the occupancy of the kernel may theoretically increase throughput, as the shared memory is more extensively used each kernel launch.[3]

However, since the algorithms uses such a large number of operations each run, a speed gain may not be achieved for the OS-CFAR CUDA algorithm. But may aid increase performance of the CA and CA-GO CFARs, due to the finer granularity of their algorithms.

The OS-CFAR was designed in order to have a constant number of operations being applied to most of the grid, those areas where a complete window cannot be extracted are discarded.

This is done to reduce the ranked selection to a full throughput problem, with roots in the idea of making the processing time for each thread constant.
As opposed to an naive implementation where the entire window is sorted before the selection is performed in a not predetermined number of operations.
This implementation utilize the fact that only a section of the window has to be sorted, and the worst case in terms of number of operations may be expected when the k'th sample is the median of the window samples. It can also be seen from the CUDA OS-CFAR algorithm; as the window gets large, the number of operations increases. This number of operations is also strongly connected to the k'th value, as may be seen from the table in 4.2.
Other algorithms such as the one from the C 'stl' library also utilize the fact that only a section or only just one sample have to be ordered and placed in its right place. While executing in a lesser number of operations.
These algorithms are all branching out, not running in a fixed number of operations, making them difficult to realize efficiently with the SIMT architecture.
The proposed algorithm suggests an alternative; executing a slow algorithm on a lot of elements in parallel, and have a high potential for further modification as well as for other applications using CUDA to make a ranked selection for a large number of short windows.
The theory section advocates the use of pitch memory allocations. This was not used in the CUDA test bench, so that all API function times in the table in the results section are those of linear memory.
If the CUDA kernel is to be used with mex, it has to be investigated whether or not mxArrays are compatible with pitch memory allocations. This thesis just assumes that pitch memory allocation is not possible, so the API function times and the results illustrating CUDA performance illustrates the worst case.

**openMP**

The use of openMp severly reduced the run times the 'stl' and 'cuda' implementation of OS-CFAR. Although it did not improve the runtime of the project code. The mean of the run times when running openMp on the project code is nearly the same as when disabling it.
Although with openMP disabled, the runtime is exactly the same each run, enabling openMP creates a spread in the runtime, and monitoring the workload of the computer cores shows activity rise on multiple processors at the instance of execution. It also shows the one core spiking and working longer with a higher intensity when openMP is disabled.
Screendumps of workload on processors on a 6 core as well as a quad core system is included in the enclosure.
The project algorithm have to execute one PRI in sequence. This is a heavy task, since the window is used as a "fifo" stack, where the two 'oldest' samples in sub windows are replaced with twoi approaching samples when a new CUT is to be evaluated.
The entire algorithm have to be executed in order. It may be that the algorithm simply is to complex to be run more efficiently by the system. Since the CPU also have to handle other system tasks while executing the algorithm. And the work load assigned to each thread is too heavy to efficiently be able to make them run concurrently on multiple processor cores, when one thread handles one PRI.
Since the CPU workload may imply the task is run serially, but assigned differently amongst the cores, instead of having one core fully dedicated to finish task. And the scheduling simply cannot allow the algorithms running simultaneously on multiple cores.
If the OS-CFAR from the project assignment is to be used in an openMP implementation, the algorithm should be broken up so that each thread have to handle a shorter task.
When looking at the run times of the 'stl' algorithm and the 'cuda' algorithm. Increase in runtime is achieved. And the cuda algorithm have the best benefits. The stl algorithm performs a pivoted search through the elements. That is, one sample is selected as a pivot for the comparisons, thereby containing a lot of comparisons, and not perform the task in a fixed number of operations.
This may explain the large spread in the run times of the 'stl' algorithm. The granularity of the algorithm may not be fine enough for the processor to be able to structure the task amongst all GPUs at the same

time. Thereby loosing multiprocessors to other tasks, since the scheduling of the algorithm has to be done for each run.
The 'cuda' algorithm is already design to have a high level of parallelism extracted. It is a lightweight task ,and it shows a great speed up in openMp applications as well for small window lengths, and a k value on the larger side of the median.
Modifications to this algorithm may be made to reduce the number of operations performed in total for each CUT.
If for example the mean is estimated while finding the largest sample. Only samples larger than the mean may be evaluated to find the rank of what we know is the largest samples.
This adds some complexity, but was never tested in this thesis. But is proposed when working with longer arrays.
By collecting statistics from the runs, the algorithm may be optimized. But it might not be necessary, since run times are already short, although it is more elegant.

## 6.2 Application

**Using CUDA**
CUDA processing requires data to be moved back and forth between the memory space of the host and device.
Mex handles data in mex specific class called mxArrays. The use of CUDA mallocPitch may not be possible, due to how three dimensional mxArrays appear to the C memory space.
Multidimensional arrays in C may be viewed as one long array, where each column is spaced by the length of the row.
These are all continuous memory addresses. When using malloc pitch, the stride or row length is used to space the memory addresses.
This makes access in a large memory space more efficiently [2].
And may reduce the time it takes to copy large arrays from host to device, and back.
In terms of CUDA runtime. The execution time of the algorithm is just a fraction of that of the API functions.
This gives rather constant run times for the different window lengths tested. All quicker than the run times achieved using openMP.
There seems to be some spread in the run times of the API functions as well. As can bee seen from the table in the results, and when examining the Application trace of the algorithm, copying data from device-¿host takes longer than copying data from host− >device.
If the GPU is to be fully utilized, more processing should be done on the GPU. This thesis will suggest the Doppler filter bank be implemented in CUDA as well.
CUDA comes with a FFT library that may be used in FIR design, or else new implementations of finite filter structures have to be investigated.
By inputting 'raw' video to the GPU, the video data can be filtered and split up in multiple channels. Decimation of PRIs in azimuth may be performed in the filter, so that amount of array space being allocated is kept to a minimum.
It should be noted that if Doppler processing is to be performed on the GPU. Only the raw data have to be copied to the GPU. The raw data is half the size of the MTD data if the filter bank contains 20 filters and is decimated by a factor 10.
Decimation in azimuth have to happen after filtering where also integration is performed. All PRIs are needed for the full Doppler Space.
In MTD, the integration in performed in the Doppler filter. So the raw video signal may not be decimated before being copied to the device for filtering.
All the PRIs are needed in the Doppler filters, as the number of PRIs used in the filter relates to the velocity

resolution by of the output signal.
Integration of pulses increases the probability of detection by decorrelating PRIs within each antenna beam width of the sector scanned by the antenna.
The video signal after the Doppler filter bank is on a MTD format, ready for CFAR processing.
As copying data between processing units is quicker than performing allocations. The initially allocated arrays should be used as much as possible.
An the state of the GPU array holding the video signal can be be copied out before a new step in the processing chain is executed.
There are some previous attempts at making a radar signal processor in CUDA,[8] and [9], looks into different steps in the processing chain, not all relevant to this MTD system.[8] only looks into a CA-CFAR with a two dimentional window to be implemented on the GPU. And compares the runtime of the GPU implementation using different memories, and compares the CUDA run times against those of a CPU.
[9] also implements FIR filters and CA-CFAR, and investigates the use of OpenCL as well as CUDA.

**Using openMP**

OpenMP reduces runtime, but introduces spread in performance, with the different implementation.
The priority of tasks may be set in the system. This may help make the openMP run times more consistent, as this runtime ambiguity is strong blow to the stern when it cannot be accounted for when it comes to using openMP in a real time system.
The tables in the enclosures shows large spread in runtime. The 'stl' algorithm is the worst, and in some cases a 16 sample window OS-CFAR may run at between 0.146 s to 0.258, which is too unreliable to be used in a real time system.
The CUDA algorithm ported to OpenMP however had a runtime between 0.323 s to 0.311 s, and is overall more stable for different window lengths.
An occasional slow run is also observed for the project code using openMP.
OpenMP is an easy to use tool to speed up code executions within loops. It does not require much additional code, and provides a speed gain in multi core processors.
If the inconsistent run times is no problems, i.e the slowest runs are still quick enough. It should be used where beneficial, and using openMP in a Doppler filter bank should be investigated in a C/C++ radar processor.

**Practical Aspects**

Adding a graphics card as a co-processor means a high increase in system power usage.
The nVidia GTX 560 card uses up to 150 W when being fully used, so that installing a powerful GPU co-processor means that the power supply of a computer likely have to be modified. The GTX 560 cards as an example requires a minimum power supply of 450 W.
If the radar is to be used with a battery as a power source, a CUDA co-processor is not recommended over the use of openMP.

# 7 Conclusion

## 7.1 Automatic Detection using Parallel Programming

It would be suggested the CFAR threshold detection be performed using a CUDA enabled co-processor. This gives consistent run times, and utilize efficient parallel computing even if the computer is under some workload.
The consistency of the run times is an important factor in a real time system, and choosing a algorithm with a large spread in the run times, the system should be designed for the worst case.
As the API routine of CUDA takes up most of the time spent when invoking a device, as much of the signal processing in the automatic detection processing chain should be moved to the CUDA SIMT environment.
Doppler filtering is a process where large amounts of parallelism may be extracted. And the SIMT instruction set of CUDA may be used efficiently.
Filtering in CUDA is well studied, and several articles may be found in the IEEE archive, as well as here[9].
Linear algebra and FFT library extensions comes with the GP-GPU toolkit from nVidia, along with a number of other extensions, such as thrust[13].
Since making the entire signal processor in CUDA C would require some development commitment, and reduce some of the flexibility that comes with normal C/C++. OpenMP is also a good move in order to achieve real time processing.
OpenMP is easier to use, it does not require learning and understanding a new programming architecture, although openMP should be further investigated in order to use the work sharing constructs in a more effective manner if it is possible, especially for the project assignment OS-CFAR.
The main motivation behind learning the SIMT architecture would be to know how a GPU performs computations, so that the programmer have absolute control over how threads perform their work, aiding in the comprehension of other data parallel architectures.
Since openMP is only one or two lines of code used in the code where one wants the parallel region of the code to begin. The programmer have less control of how the parallel region of the code is executed, as depends on how the compiler interprets the source code.
It has been proven by the results found in this thesis that openMP is valuable at speeding up task where parallelism is exposed. As may be seen when directly comparing the graphs showing run times when openMP is enable and disabled respectively for the different implementations.
The largest speed gains comes from the CUDA algorithm ported to openMP, which proves efficient exploitation of parallelism extracted from the problem. Both for GPU and multicore CPU.
This provides further motivation for learning CUDA, as this will aid the understanding of how to expose parallelism, showed crucial in order to use openMP in a most efficient manner.
Another benefit of CUDA is that a graphics card may be chosen to suit different applications. Making a GP-GPU radar processor easily portable between ranges of different graphics card having a spectre of processing power.
This is mass produced hardware, available to the consumer marked and real time MTD processing may be expected for both laptops and desktop computers, provided they have a sufficient CUDA card or a sufficient number of core CPU clocked at a high enough frequency.
By building the GPU code as .dll library extensions, it may be called from any C style source code, as well as mex.
These extensions have to be pre-compiled. Scalability in terms of window lengths and what sample to select have to be investigated.
The current code is generated for compute capability 2.x, if the processor is to be backward compatible, and work on older graphics cards, how the grid is divided into thread blocks have to be altered, so that the number of CUT processed each kernel invocation may use the shared memory in order to achieve sufficient throughput.

## 7.2 The Implementations

**CUDA**

A CA-CFAR as well as an OS-CFAR algorithm was developed in CUDA C.
The CA-CFAR only requires the arithmetical operations of summation and division. And is a problem which inherently have a large amount of parallelism exposed.
It can be seen from the table of kernel run times in the results, a conditional statement increases the runtime of the CA-CFAR algorithm in the CAGO-CFAR.
Comparing kernel run times, the OS-CFAR kernel performs a lot worse when the sample size increases compared to the CA-CFARs.
The first implementation of OS-CFAR did not use shared memory, and the kernel execution time was well over two seconds. Shared memory use decreased the run time to what is presented in the results.
The number of operations as mentioned increases exponentially. And a suggestion is made to decrease runtime by calculating the mean of the signal power within the window.
This may affect runtime for the worse by making the number of operations required a variable. And may increase branching within thread warps. Making warps diverge, and loosing parallelism in the execution path.
The main idea behind the algorithms was to create throughput problem, not necessarily the shortest path to finding the sample of interest, but a fixed path that could be performed on a lot of elements simultaneously, without any overhead or scheduling.
Algorithms performing searches in a reduced number of operations have to make assumptions, these assumptions creates ambiguity in the execution path of the thread. This have to be studied if the run times achieved with the current algorithm is not satisfactory.
The CUDA run times are shorter than those of the other implementations, even when not using pitch allocation, as well as much more stable than those of the openMP.
Advocating the use of CUDA.

# References


[1] Merrill Skolnik,
*Introduction to Radar Systems 3. edition.*
ISBN-13: 978-0-07288138-7
McGraw-Hill
2002.

[2] nVidia Developer Zone,
*NVIDIA CUDA C Programming Guide Version 4.1*
http://developer.nvidia.com/nvidia-gpu-computing-documentation,
November 2011.

[3] Jason Sanders and Edward Kandrot,
*CUDA by Example: An Introduction to General Purpose GPU programming.*
Addison-Wesley
ISBN-13: 978-0-13-138768-3
2011.

[4] Hermann Rohling
*Radar CFAR Thresholding in Clutter and Multiple Target Situations*
AEG-Telefunken,
1982.

[5] Morten Ihlen,
*Automatic Detection for MTI Processed Radar Signals*
NTNU Department of Electronics and Telecommunications,
Norway
2011

[6] Hans Fjeld,
*Quick OS-CFAR Algorithm Implementation in MatLab*
NTNU Department of Electronics and Telecommunications,
Norway
2011

[7] Stephen Blake
*OS-CFAR Theory for Multiple Targets and Nonuniform Clutter*
Analytic Associates Ltd.,
IEEE Transaction On Aerospace And Electronic Systems VOL. 24, NO. 6,
November 1988

[8] C.J. Venter[1,2], H. Grobler[2] and K.A. AlMalki[3],
*Implementation of the CA-CFAR Algorithm for Pulsed-Doppler Radar on a GPU Architecture*
[1]: Defence, Peace, Safety and Security, Council for Scientific and Industrial Research, South Africa
[2]: Department of Electrical, Electronic and Computer Engineering, University of Pretoria, South Africa
[3]: Electronics, Communications and Photonics, King Abdulaziz City for Science and Technology,
2011 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT),
Kingdom of Saudi Arabia
2011.

[9] Jimmy Pettersson and Ian Wainwright
*Radar Signal Processing with Graphics Processors (GPUs).*

University of Uppsala,
Sweden,
2010.

[10] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, And Robert E. Tarjan
*Time Bounds for Selection*
Department of Computer Science, Stanford University,
Stanford, California 94305,
1972.

[11] M.L. Stone and J.R. Anderson
*Advances in Primary-Radar Technology.*
The Lincoln Laboratory Journal, Volume 2, Number 3,
1989.

[12] Tolu Alabi, Jeffrey D. Blanchard, Bradley Gordon, and Russel Steinbach (ABGS),
*Fast K-selection Algorithms for Graphics Processing Units*
Grinnell College.,
2011.

[13] nVidia Developer Zone,
*CUDA Toolkit 4.1 Thrust Quick Start Guide*
http://developer.nvidia.com/nvidia-gpu-computing-documentation,
November 2011.

[14] E. Løvli (Radian AS) and Y. Steinheim(Sintef IKT),
*MTD for magnetronradar En populærvitenskapelig introduksjon*
2011.