Advanced Computer Architecture

CFAR Implementation with CUDA

Bert Pyck
Hugo Charmant

ELICT: ELECFS
16 mei 2025
Master in Engineering Technology
Academic year 2024-2025

1 REPORT

1.1 Introduction

For the course Advanced Computer Architecture, we implemented a project to demonstrate the difference between CPU and GPU performance, particularly highlighting the advantages of using a GPU when processing large amounts of similar data. We chose to implement a signal processing algorithm known as Constant False Alarm Rate (CFAR), which is used to detect target reflections against a background of noise, clutter, and interference [1].

The method involves transmitting a signal and measuring the time it takes for the signal to return. This time can be converted into path length by multiplying it by the propagation speed of the wave. The propagation speed depends on the type of wave. Electromagnetic waves (RF) or pressure waves (acoustics). To test our implementation, we collected data using the acoustic-RF Testbed Techtile [2]. We chose to use acoustic signals due to their lower propagation speed and robustness. The setup for measuring acoustic signals is relatively simple and provides accurate and reproducible measurements.

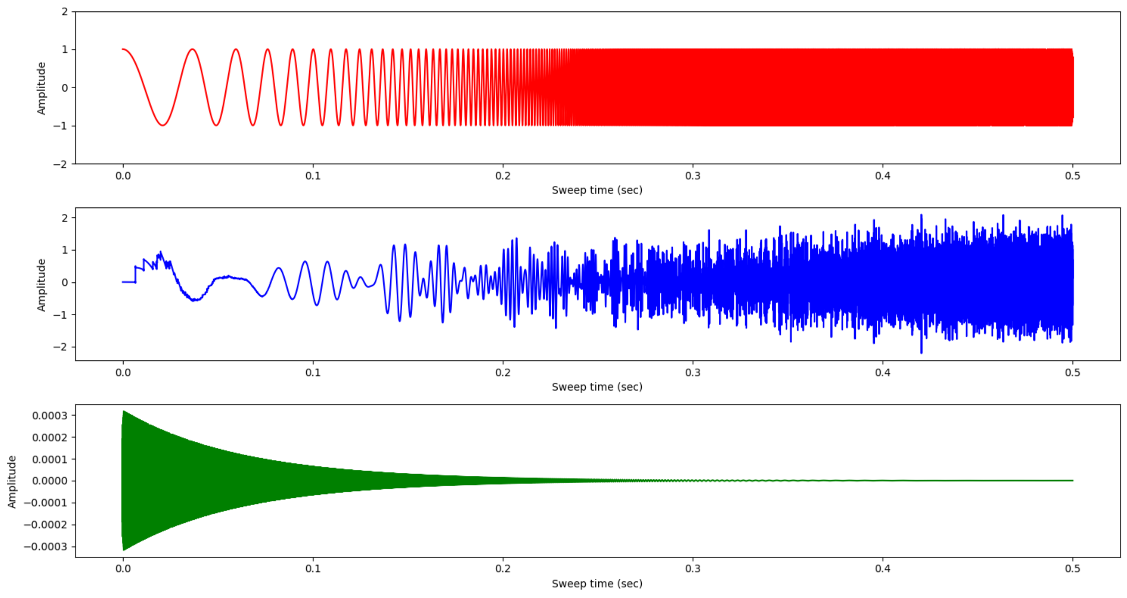
1.2 Measurement technique

The method of measuring the distance to a target is similar to how bats, whales, and dolphins navigate through their environment. They use echoes to locate themselves. We do the same: we place a microphone next to a source and then emit an excitation signal and measure the response. The excitation signal will travel to a surface and reflect back.

Instead of exciting a dirac impulse in a specific direction and measuring the response, we use an exponential sine sweep. This is because generating a perfect dirac impulse is physically impossible. We can mimic it by popping a balloon, firing a gun, or clapping hands. These methods are feasible but still not fast enough. They are useful when a rough estimate of the reflection is sufficient. However, the signal-to-noise ratio (SNR) will typically be too low, making the reflections indistinguishable from noise. The biggest reason to use other methods is reproducibility. With the described techniques, it is impossible to generate two perfectly identical impulses. Practically, we can determine the reflection or acoustic response $h(t)$, knowing the excitation signal $x(t)$ and the recorded signal $y(t)$. Instead of sending a dirac impulse, we send an exponential sine sweep, a frequency-varying sine wave that exponentially covers a broad frequency spectrum. As shown in Figure 1.1, we send an exponential sine sweep (red signal) through the room, and the recorded signal $y(t)$ will be a distorted version of the exponential sine sweep (blue signal). The distortion is determined by the acoustic characteristics of the room.

From this, we can determine the reflection response by convolving the recorded signal $y(t)$ with the inverted and exponentially compensated exponential sine sweep (green signal). The reason we need to convolve is to extract the 'sweep' from the recorded signal. We must exponentially compensate the amplitude of the inverted exponential sine sweep because, due to its exponential nature, an exponential sine sweep contains most of its energy in the lower frequencies. Therefore, the inverted signal needs to retain the amplitude at higher frequencies and exponentially attenuate the amplitude at lower frequencies to compensate for this energy distribution.

Although convolution is a theoretically sound method, for practical implementation we use the FFT (Fast Fourier Transform) and IFFT (Inverse FFT). Convolution in the time domain corresponds to multiplication in the frequency domain. This means that the relatively heavy mathematical calculations can be simplified into a division. The transformations between the time domain and the frequency domain are efficiently carried out using the FFT and IFFT methods.



Figuur 1.1: Measuring reflection with an exponential sine sweep

1.3 Constant False Alarm Rate (CFAR) algorithm

We must try to extract the reflection from the measurement. A lot of noise is introduced, so some form of signal processing must be implemented to properly distinguish the reflection from the noise. As the term peak-detection suggests, the most logical approach is to implement a threshold level, and when this level is reached, classify it as a reflection. However, the drawback of this method is that the threshold value is dependent on the distance which is, of course, the very thing we want to calculate. Therefore, we must choose and apply a static threshold value. If the value is set too high, reflections might not be detected at all. If it is set too low, noise elements will be selected, which can overly burden the processing system. So we need to look for a dynamic peak detection system. One example of such a system is the use of Constant False Alarm Rate (CFAR) detection. This system uses an adaptive threshold value instead of a static one. There are several CFAR techniques that are situation-dependent and that predict the adaptive threshold value in different ways [3]. The fundamental and most commonly used technique is cell averaging CFAR detection. A one-dimensional vector is constructed from the samples of the response $h(t)$. This vector is built around a CUT (CUT), with guard cells and reference cells on both sides. The vector slides across the signal so that each sample becomes a CUT once The goal is to process the entire signal and generate an adaptive threshold value. The function of the reference cells is to calculate the average of a certain number of leading and lagging samples around the CUT. The guard cells, located between the CUT and the reference cells, are there to exclude target energy from the noise estimate.

The structure of the signal can be seen as a collection of samples:

$$\text{signal} = [X[0], X[1], ..., X[k], ...X[i]] \tag{1.1}$$

This is the structure of the one-dimensional vector around a central sample $X[k]$:

$$[X[k-G-R], ..., X[k-G], ..., X[k-1], X[k], X[X+1], ..., X[k+G], ..., X[k+G+R]] \tag{1.2}$$

With:

- $X[k]$: the CUT
- R: amount of reference cells on one side
- G: amount of guards cells on one side

We can calculate the adaptive threshold value $\alpha[k]$ using the formula below:

$$\alpha[k] = B_\times \cdot \frac{1}{2R} \sum_{\substack{j=k-R-G \text{ and} \\ j<k-G \text{ of } j>k+G}}^{k+R+G} X[j] + B_+ \tag{1.3}$$

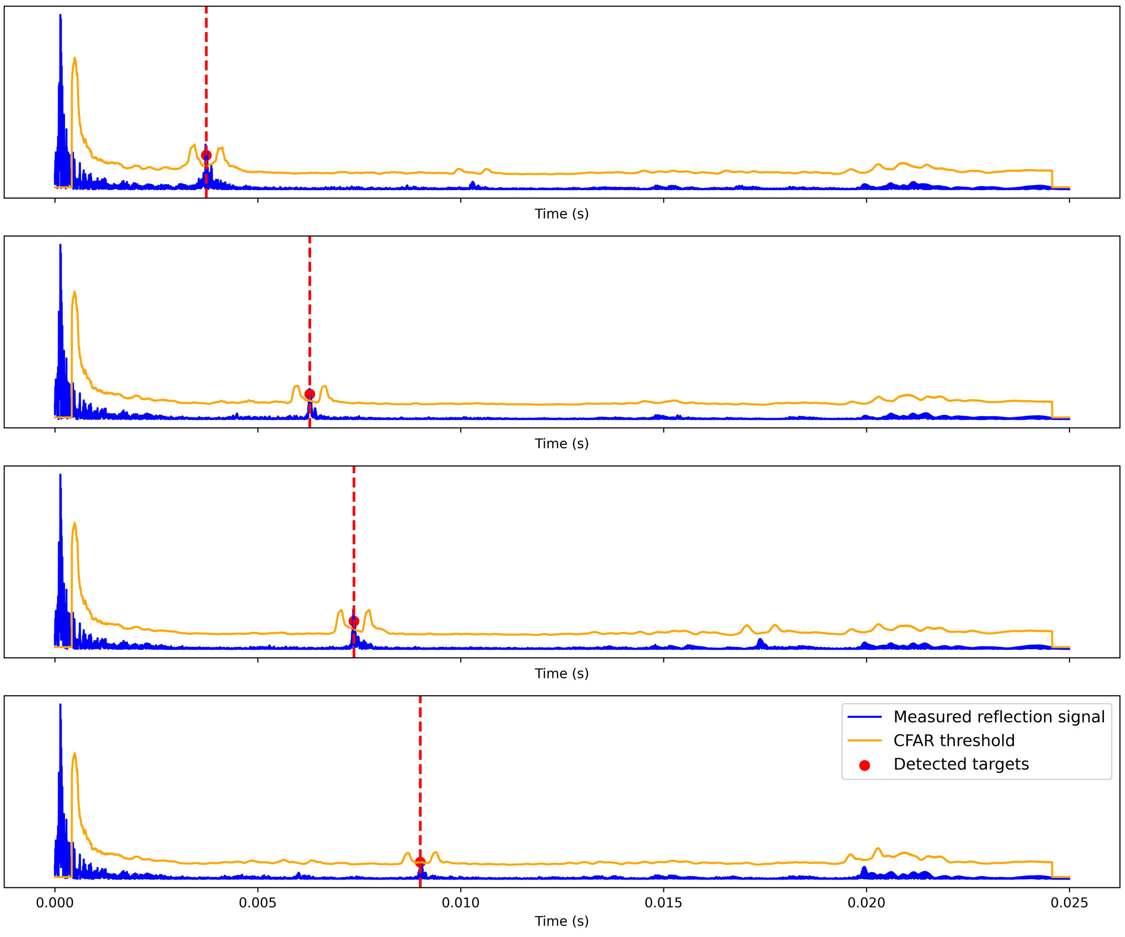
With:

- $B_\times$: multiplicative bias, a scaling factor to make the threshold stricter or more lenient.
- B_+: additive bias, a fixed offset that is added to the threshold.

$$\text{Detected peaks} = \forall (X[k] \geq \alpha[k]) \tag{1.4}$$

Figure 1.2 shows four reflection signals measured using the exponential sine sweep method. For each signal, the reflective wall is positioned progressively farther away. This is clearly visible, as each measured reflection has a longer time of arrival. To detect these arrival times, we used the CFAR peak-picking algorithm described above. It's clearly visible that the adaptive threshold follows the shape of the received signal. We carefully chose the parameters so that only the first reflections are detected. With slight changes to these parameters, it is also possible to detect the other reflections. However, these additional reflections do not contain much useful information, as they are primarily caused by the measurement setup in a room. They represent second-order reflections and backscatter.

Table 1.1 shows that the measured distances closely match the chosen expected distances. The small deviations can be attributed to measurement inaccuracies and factors such as absorption. Overall, the results confirm that the applied measurement and peak-detection method is reliable for determining distances based on time of arrival data.



Figuur 1.2: Measured multiple reflections for multiple distances with CFAR detection

Expected distances (m)	Measured time instances (s)	Measured distances (m)
1.3	0.003732	1.280076
2.2	0.00628	2.15404
2.5	0.007368	2.527224
3	0.009004	3.088372

Tabel 1.1: Overview of chosen distances, measured time instances, and calculated measured distances.

1.4 Implementation

1.4.1 Reference CPU code

We first wrote a baseline CFAR code that iterates serially over a one-dimensional array of size 6250. Each iteration computes the average of all the reference cells, using the guard cells as offset. We also used bias parameters for the multiplication and addition to adjust the detection threshold. We average the computing time over 1000 iterations to eliminate the effect of other processes running on the machine.

1.4.2 GPU Implementation

This code assigns each CUT to a separate thread. The threads are organized into blocks of 1024 threads, and blocks are processed by Streaming Multiprocessors (SM's). Each thread loads the relevant reference and guard-cell samples from global memory, computes the threshold, and compares the value of the CUT with this threshold.

1.5 Optimizations

1.5.1 Block size optimization

Choosing the block size directly affects the ratio of active warps (blocks of 32 threads) per SM's, and therefore overall throughput.

When analysing the amount of unused threads in this configuration, we see that we have 6 full blocks of 1024 threads ($6 \cdot 1024 = 6144$) and one block with only 106 used threads. This causes an overhead of 918 unused threads, which is obviously not optimal.
To tackle this issue, we first looked at using blocks sizes as powers of 2 (64, 128, 256, 512) where the overhead is low. In following table a comparison was made between these possibilities:

Block Size	**Blocks Needed**	**Threads Allocated**	**Unused Threads**	**Efficiency**
64	98	6272	22	99.65%
128	49	6272	22	99.65%
256	25	6400	150	97.66%
512	13	6656	406	93.90%
1024	7	7168	918	87.19%

Tabel 1.2: Comparison of thread utilization for 6250 elements across different block sizes

Based on this table, we see that block sizes of 64 and 128 threads are the most efficient, but it would be obviously more sensible to use 128 blocks. We see that our initial block size was not optimal and allows for great improvements.

Initial tests were performed, for which you can find the results in following table:

Threads per Block	**GPU Time (ms)**
64	0.025828
128	0.026890
256	0.025835
512	0.025953
1024	0.030950

Tabel 1.3: GPU Computing Time Comparison for Different Block Sizes

This again shows that the computing time increases when the efficiency drops.

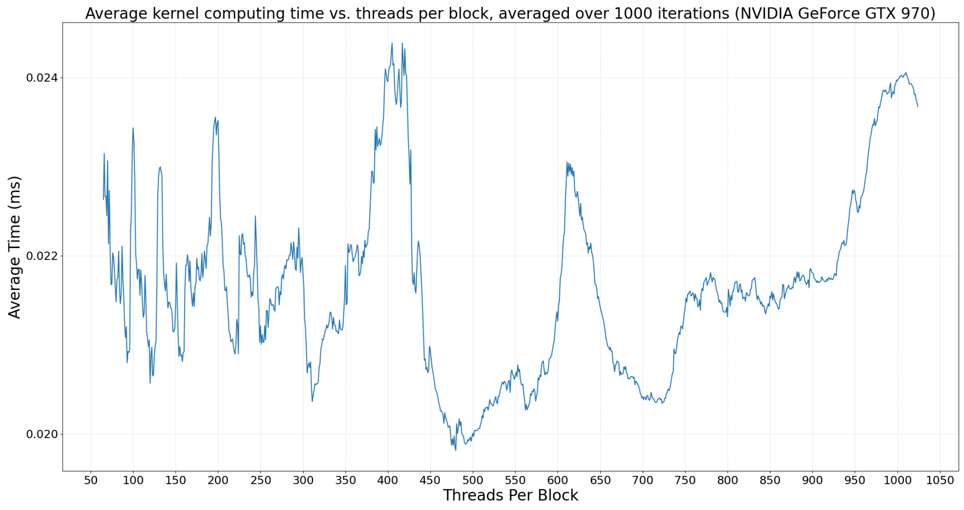
Whilst doing these computations, we looked into our hardware in detail. The GPU used for this project was the NVIDIA GeForce GTX 970. Whilst it was first introduced in 2014 and many new generations have been introduced by now, its still capable of delivering strong computing performances. We will now detail the specifications of this GPU [4]:

- **Architecture:** NVIDIA Maxwell

- **CUDA Cores:** 1664 CUDA cores
- **Streaming Multiprocessors:** 13 SM's
- **Compute Capability:** 5.2 (Maxwell generation)
- **Base Clock:** 1050 MHz
- **Memory:** 4GB DDR5
- **Memory Interface:** 256-bit
- **Memory Bandwidth:** 224.4 GB/s
- **Shared memory size:** 96 kB

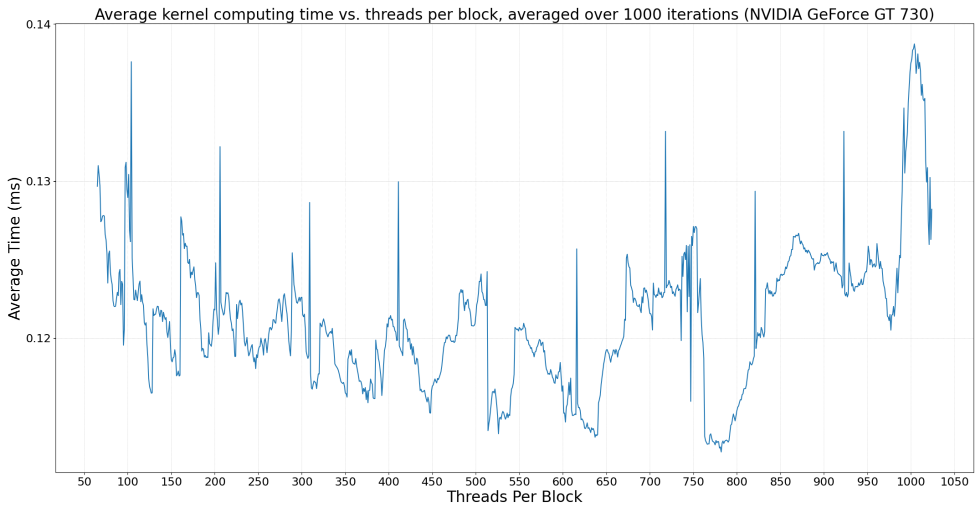
The most important value that we can utilize from these specifications is the amount of SM's. When executing a kernel, each block gets assigned to one SM. We thought it could therefore be useful to use a block size which would create 13 blocks to compute our input array. This array contains 6250 elements, and through simple calculations, we find following result: amount of elements per SM = amount of threads per block = $\frac{6250}{13} = 480.77$. Rounding up gives 481 threads in one block. This is our expected optimal block size. When computing this block size, we indeed found an improvement compared to previous attempts.

In order to make sure this new block size was indeed the most optimal, we performed the computation for each block size between 64 and 1024, again averaging over 1000 iterations. The graph in Figure 1.3 shows the results. We indeed find that our calculated block size of 481 threads is the most optimal configuration in our specific use case. Additionally, we note that there are local minima at the powers of 2 previously explored, especially at 128 threads per block we see a clear minimum.



Figuur 1.3: Computing time in function of the amount of threads per block for our GPU

Next, we wanted to test our code when using a different GPU, for which we used the lab computers, equipped with the NVIDIA GeForce GT 730 GPU. This GPU has 2 SM's [5]. The graph speaks for itself, as the minimum is at a completely different block size.



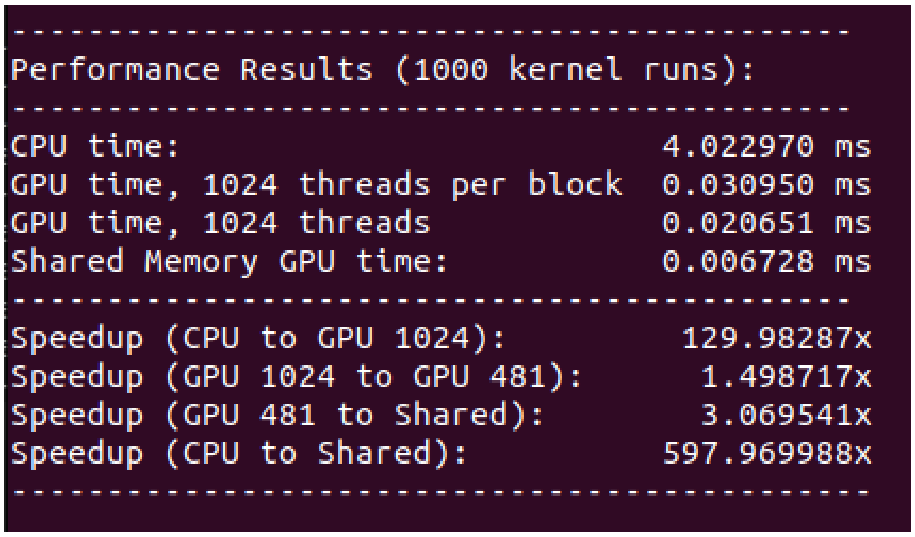
Figuur 1.4: Computing time in function of the amount of threads per block for the lab GPU

1.5.2 Shared memory utilization

Global memory accesses on GPUs cause hundreds of cycles of latency, and especially with uncoalesced access, this causes an underutilization of the available bandwidth. The CFAR algorithm has regular, overlapping and predictable memory access patterns: neighbouring CUTs threads read overlapping reference-cell regions. To exploit this data reuse and reduce global memory access, we placed each block's window data into on-chip shared memory. Each block loads its required data into shared memory, with boundary margins to handle the outer cells under tests. Access to the shared memory improves parallelism and reduce redudancy, meaning that our computation gets more optimized.

1.6 Results

In this chapter, the results of the optimizations are shown. From the terminal screenshot in Figure 1.5, we can see that using our original GPU implementation made a huge improvement compared with the CPU computation time. Our additional improvements ultimately led to a 597x acceleration, which is a very good result.



```
------------------------------------------
Performance Results (1000 kernel runs):
------------------------------------------
CPU time:                          4.022970 ms
GPU time, 1024 threads per block  0.030950 ms
GPU time, 1024 threads            0.020651 ms
Shared Memory GPU time:           0.006728 ms
------------------------------------------
Speedup (CPU to GPU 1024):          129.98287x
Speedup (GPU 1024 to GPU 481):        1.498717x
Speedup (GPU 481 to Shared):          3.069541x
Speedup (CPU to Shared):            597.969988x
-------------------------------------------
```

Figuur 1.5: Computing time results

1.7 Shortcomings and possible improvements

- Shared memory per SM (96 kB on our GPU) contraints the maximum amount of reference cells and guard cells. We could expect that for RF application, where the amount of samples is a lot higher, we might have to utilize tiling to compute the CFAR.
- Spawning kernels could increase granularity in regions of interest, like when a peak is starting to get detected. This could make our positioning more accurate.
- We see the implementation of an RF-based test setup as a possible improvement in accuracy and detection range. The use of our GPU code would then make more sense, as the amount of samples also increases.

1.8 Conclusion

We conclude that our proposed optimization achieves an approximate improvement of 597 times compared to the CPU implementation. Whilst the setup used for the measurements only produces an array of 6250 elements 40 times per second, which the CPU can still process in real time, we can easily see this algorithm being used for radar applications. Here, due to the much greater propagation speed, a lot more samples need to be captured in order to estimate distance. In a situation with hundreds of thousands samples per 25 ms, the use of the GPU would be required to provide real time measurement.

[1] L.L. Scharf en C. Demeure. *Statistical Signal Processing: Detection, Estimation, and Time Series Analysis*. Addison-Wesley series in electrical and computer engineering. Addison-Wesley Publishing Company, 1991. ISBN: 9780201190380. URL: `https://books.google.be/books?id=y_dSAAAAMAAJ`.

[2] Daan Delabie e.a. „Techtile: a Flexible Testbed for Distributed Acoustic Indoor Positioning and Sensing". In: *2022 IEEE Sensors Applications Symposium (SAS)*. 2022, p. 1–6. DOI: `10.1109/SAS54819.2022.9881342`.

[3] M.A. Richards. *Fundamentals of Radar Signal Processing, Second Edition*. McGraw-Hill Education, 2014. ISBN: 9780071798327. URL: `https://books.google.be/books?id=i5KAMAEACAAJ`.

[4] TechPowerUp. *NVIDIA GeForce GTX 970 Specs*. Accessed: 2025-05-09. n.d. URL: `https://www.techpowerup.com/gpu-specs/geforce-gtx-970.c2620`.

[5] TechPowerUp. *NVIDIA GeForce GT 730 Specs*. Accessed: 2025-05-09. n.d. URL: `https://www.techpowerup.com/gpu-specs/geforce-gt-730.c1988`.