

High-performance image reconstruction algorithm in CUDA C++ for ultra wideband multi-channel MIMO radar systems

Josh Perske, Harun Cetinkaya, Christopher Schwäbig,
and Sabine Gütemann

Fraunhofer Institute for High Frequency Physics and Radar Techniques FHR,
Fraunhoferstraße 20, 53343 Wachtberg

Abstract The exact measurement of process-relevant parameters and product properties are prerequisites for efficient and sustainable production. In addition to accuracy, industrial applications place tough demands on the real-time capability and achievable measurement rates of the sensor technology. In the past, radar signal processing was mainly done with the use of highly specialised hardware to achieve the necessary performance. Computer systems are used to perform simulations and to test new algorithms before being implemented under high effort. The resulting sensor systems are rigid, and their enhancement is time and cost consuming. With increasingly powerful graphics processing units (GPU) and the possibility to use them for general-purpose computing, a new approach is to outsource parts of the radar signal processing from the specialised hardware to commercially available computer systems. The main objective of this idea is to reduce the development time of new sensor systems, facilitate their modification and to increase the re-usability of produced code. This approach is tested with a new imaging radar algorithm, developed for a frequency modulated continuous wave (FMCW) radar system with a modular multiple input multiple output (MIMO) antenna array. The implementation of this algorithm is used to determine the boundaries of this new approach and involves a step-by-step optimisation process to improve the performance of the final result.

Keywords Imaging radar algorithm, FMCW radar, MIMO sensor system, back-projection, CUDA C++, GPU programming

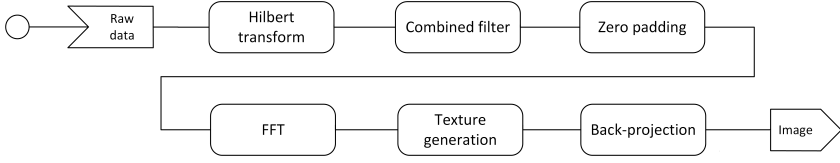


Figure 1: Activity diagram showing the steps of the radar imaging algorithm.

1 Sensor system

Based on recent research on MIMO imaging radar sensors and prior projects to measure the width of steel slabs in rolling mills [1] [2] [3] [4], a new sensor system is planned to not only measure the metal slab dimensions, but also to reconstruct high-resolution images of the material surface and to determine its speed.

The MIMO signal processing and its GPU implementation are based on a planned radar system with 197 real channels. The sensor operates according to the FMCW principle with a 3dB transmission range of 30GHz (119GHz – 149GHz). The sensor has a simulated resolution of 0.8mm along the vertical axis at a distance of 500mm. The combination of all transmit and receive channels provides a virtual array aperture of 1300mm distributed over 677 spatial positions with a virtual sampling distance of $2\lambda \approx 4.2\text{mm}$.

The transmitters are time-multiplexed, and only one transmitter is active at once. While the active transmitter is sending the chirp pulse, all other antennas act as receiver for the reflected radar signal.

2 Imaging algorithm

The input data for the algorithm is any number of pre-determined intermediate frequency (IF) signals $s_{\text{TxRx}}(n)$ with size N from any transmitter (Tx) and receiver (Rx). The raw data is transmitted via ethernet using an UDP-based data protocol. Those given, the imaging algorithm consists of the steps shown in figure 1.

2.1 Signal pre-processing

The pre-processing involves a Hilbert transform function \mathcal{H} to get the analytic input signal and the application of a complex combined filter function $\underline{w}_{\text{TxRx}}(n)$ with zero padding. In consideration of these two functions, the signal after the pre-processing step is given by (1).

$$\underline{s}_{\text{TxRx}}(n) = \begin{cases} \underline{w}_{\text{TxRx}}(n)[s_{\text{TxRx}}(n) + i\mathcal{H}(s_{\text{TxRx}}(n))] & \text{if } n < N \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

For the actual implementation, the analytic signal is approximated through a Fast Fourier transform (FFT) by setting all negative frequencies in the signal spectrum to zero followed by an inverse FFT [5]. The zero padding then appends a specified number of zeros to the filtered IF-signal to get a spectrum with lower peaks but higher distance resolution during the image reconstruction. With the speed of light c_0 and the radar bandwidth B , the enhanced step size of the range axis after the zero padding Δd is given by (2). The possible length N_P of the padded signal $\underline{s}_{\text{TxRx}}(n)$ will be determined through tests with the finished system.

$$\Delta d = \frac{c_0 N}{2BN_P} \quad (2)$$

2.2 Combined filter

The combined filter $\underline{w}_{\text{TxRx}}(n)$ in (3) consists of several sub-filters for different tasks. The calibration $\underline{w}_{\text{cal}}(n)$ removes channel response from the measurement signal, the Hamming filter $H(n)$ and the Kaiser filter K_{TxRx} suppress side lobes and noise in the imaging area and the multiplicity value M_{TxRx} equalises the illumination level along the aperture of the MIMO array. The effect of the different filters is shown in figure 2.

$$\underline{w}_{\text{TxRx}}(n) = \underline{w}_{\text{cal}}(n) \frac{H(n)K_{\text{TxRx}}}{M_{\text{TxRx}}} \quad (3)$$

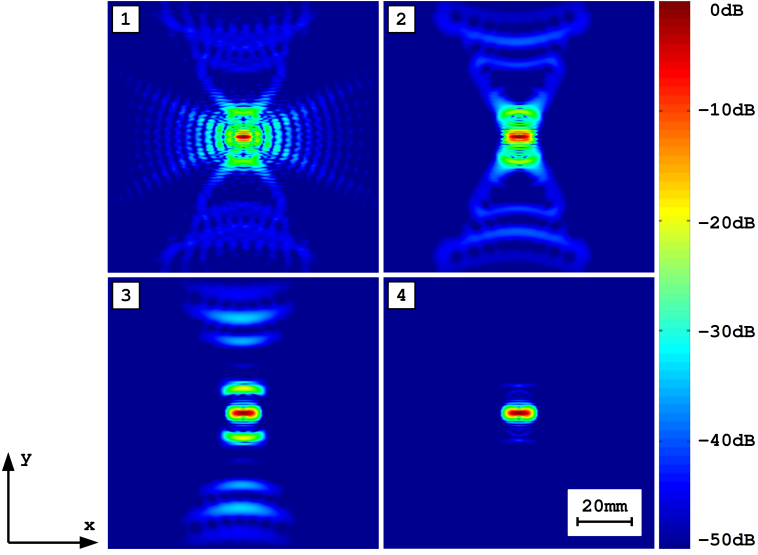


Figure 2: Output data $|I(x, y)|$ of the imaging algorithm with 970 simulated input signals ($B = 160\text{GHz} - 120\text{GHz}$, $N = 1024$, $N_p = 8192$) for a single point reflector with (1) No filters (2) Hamming filter (3) Hamming and Kaiser filter (4) Hamming, Kaiser and Multiplicity filter.

The Hamming filter (4) is applied as a window function over the whole signal length in range direction reducing side lobes and noise along the x-axis of the reconstructed image.

$$H(n) = 0.54 - 0.47\cos\left(\frac{2\pi n}{N-1}\right) \quad (4)$$

The Kaiser filter is not applied over the signal length but over the y-axis of the antenna array. A virtual antenna position V_{TxRx} is calculated as the midpoint between the transmitter position P_{Tx} and receiver position P_{Rx} . The virtual y-position of each pair is then used to calculate the Kaiser value K_{TxRx} in (5). Here, I_0 is the zeroth-order modified Bessel function of the first kind.

$$K_{\text{TxRx}} = \frac{I_0 \left[\pi \alpha \sqrt{1 - \left(2 \frac{V_{\text{TxRx},Y} - \min(V_Y)}{\max(V_Y) - \min(V_Y)} - 1 \right)^2} \right]}{I_0(\pi \alpha)} \quad \text{with} \quad (5)$$

$$\alpha = 4.0$$

The multiplicity value M_{TxRx} for each antenna pair is generated by counting the number of overlapping virtual antenna positions within a threshold radius around V_{TxRx} .

2.3 Image reconstruction

The image reconstruction steps use a back-projection algorithm to map the filtered input signals onto a two-dimensional plane. Given by the physical properties of the FMCW radar, peaks in the IF-signal spectrum correspond to the presence of a reflecting object in the sensors' field-of-view. Therefore, the first step is to perform the FFT of the IF-signal. In (6) the signal response of an antenna pair $\underline{S}_{\text{TxRx}}(x, y)$ is calculated for any position in the target area with the distance $d_{\text{TxRx}}(x, y)$ between the antennas and the pixel position.

$$\underline{S}_{\text{TxRx}}(x, y) = \mathcal{FFT} [\underline{S}_{\text{TxRx}}(n)] \left(\frac{d_{\text{TxRx}}(x, y)}{\Delta d} \right) \quad (6)$$

Since the FFT is a discrete function, it is not possible to calculate the value of $\underline{S}_{\text{TxRx}}(x, y)$ directly. Therefore, it is necessary to interpolate the result of the FFT at the target distance to get a continuous function. The efficient implementation of this interpolation through GPU texture memory is one of the main aspects of the optimisation process described in this work.

Finally, the signal response of each antenna pair is superimposed to get the combined reflection intensity for any target position. With the application of a phase correction value $\Phi(d)$, the reflectivity function $\underline{I}(x, y)$ can be represented as in (7). Here, f_0 refers to the starting frequency of the FMCW radar chirp.

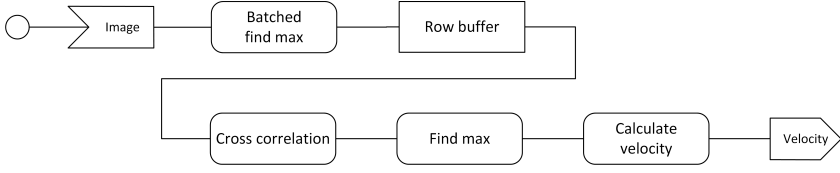


Figure 3: Activity diagram showing the steps of the velocity estimation algorithm.

$$\begin{aligned} \underline{I}(x, y) &= \sum_{\text{TxRx}} \underline{\Phi}[\underline{d}_{\text{TxRx}}(x, y)] \underline{S}_{\text{TxRx}}(x, y) \quad \text{with} \\ \underline{\Phi}(d) &= e^{-i \frac{2\pi f_0}{c_0} d} \end{aligned} \quad (7)$$

3 Velocity estimation

The velocity estimation is based on the imaging algorithm. With a MIMO array arranged along the movement axis of the observed object, the shift of the object is determined through the cross-correlation of two consecutive measurements. The steps of the velocity estimation in figure 3 involve the reduction of the image data to an one-dimensional function (8), from which the shift of an object is determined through a cross-correlation with the previous image data (9).

$$I_{\max}(y) = \max_{x \in D_x} (|\underline{I}(x, y)|) \quad \text{with } D_x = [x_{\min}, x_{\max}] \quad (8)$$

The cross-correlation is calculated between each $I_{\max,i}$ and the previous $I_{\max,i-1}$ to determine the shift of the observed object. Here, \star is the short notation for the cross-correlation.

$$\Delta y = \underset{y \in D_y}{\operatorname{argmax}} (I_{\max,i}(y) \star I_{\max,i-1}(y)) \quad \text{with } D_y = [y_{\min}, y_{\max}] \quad (9)$$

Finally, the velocity is calculated with the timestamps t_i and t_{i-1} of the received data (10).

$$v = \frac{\Delta y}{t_i - t_{i-1}} \quad (10)$$

4 Implementation details

The approach of this project is not only to implement the new algorithm in an efficient way, but also to ensure the re-usability of the developed code by creating the foundation for a general-purpose CUDA signal processing library. With that in mind, the focus during the development is on modularity, the creation of clean and safe code and a proper documentation.

The software is written in C++17 and is completely object-oriented. The different parts involve an advanced memory management, data handling, error handling and a flexible structure for the implementation of new arithmetic operations. Since CUDA code uses global definitions for the kernel and device functions and in some cases even needs global variables, all those relics from CUDA C are hidden behind proxy classes and never exposed to the user of the library. In order *to make error handling systematic, robust, and non-repetitive* [6, E.2], this library replaces the return-based error handling from the CUDA Runtime API with a throw-based error handling with dedicated exception classes.

4.1 Memory organisation

The CUDA Runtime API provides C-like `cudaMalloc` and `cudaFree` functions for memory allocation and deallocation. This technique is still supported but outdated in modern C++ [6, R.10] and therefore, those functions are wrapped in the class `DeviceArray` to perform an automatic allocation and deallocation in its Constructor and Destructor. Avoiding manual memory allocation reduces the risk of leaks and simplifies the memory management.

Another abstraction layer shown in figure 4 is the implementation of different `DeviceData` subclasses. Those subclasses hold additional information about the data dimension and provide methods to access subareas of the allocated memory without manual pointer manipulation.

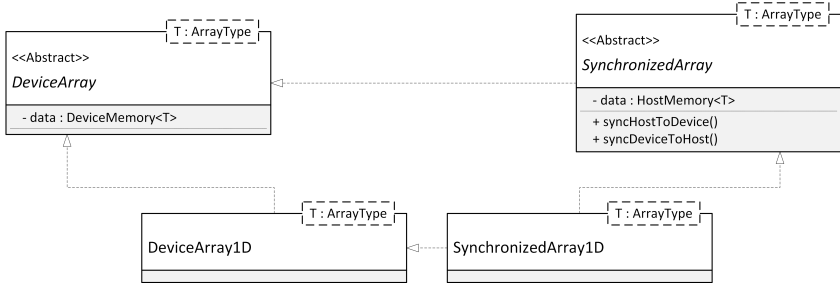


Figure 4: Simplified class diagram of the implemented memory management. Classes derived from `DeviceArray` inherit the automatic device memory allocation. Classes derived from `SynchronizedArray` inherit the automatic host memory allocation and synchronisation methods.

In addition, the `SynchronizedDeviceArray` classes allocate host memory and allow the bi-directional synchronisation of host and device memory. The `cudaMallocHost` method is used to enable a faster data transfer during the synchronisation. Virtual inheritance is used to resolve the diamond pattern in this design.

All memory management classes are templates to be usable with different data types without code duplication. Those templates are specialised through explicit template specialisation, which generates the source code for a specific selection of data types during compile time. This technique is mainly used to restrict the usage of the template classes to only implemented and tested data types. This explicit form of generating source code from templates during compilation (template metaprogramming) is not recommended in the C++ Core Guidelines [6, T.120,T.121], except for the emulation of concepts. Although this library would effectively benefit from using concepts, the Nvidia Nvcc compiler does not support this new C++20 feature yet.

4.2 Arithmetic operations

All arithmetic functions and various memory operations are implemented using a visitor pattern. The design shown in figure 5 implements the different operations as visitors in dedicated classes which get called by the memory objects. This design avoids hard binding

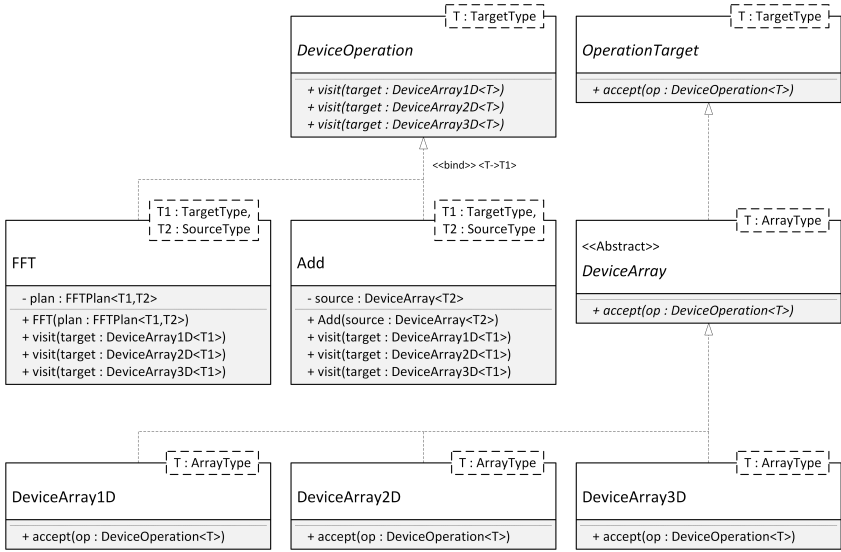


Figure 5: Simplified class diagram of the implemented visitor pattern for the arithmetic operations. Any `DeviceOperation` can be applied on any `OperationTarget`. The two operations `FFT` and `Add` are examples of concrete `DeviceOperation` classes.

between the implemented operations and the data on which those are applied and facilitates the implementation of new operations.

An alternative to this design would be a Utility Class implemented as Singleton or with the use of static methods. There is a wide discussion about the usage of Utility Classes, and in general they are not seen as good practice. They break the principles of object-oriented programming by having only one instance of the class, which comes with several downsides compared to a regular instantiable class. Moreover, the tight coupling to the Utility Class prevents to switch this dependency by creating a subclass and extending its functionality.

4.3 Optimisations

The final implementation of the image processing algorithm is the result of a longer optimisation process, and a selection of the interme-

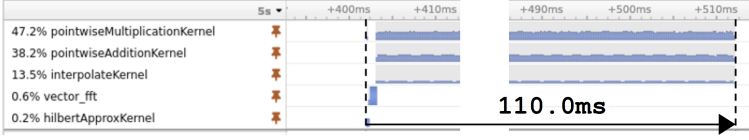


Figure 6: Nvidia Nsight Systems timeline for the unoptimised algorithm. Many small kernels are launched and executed sequentially.

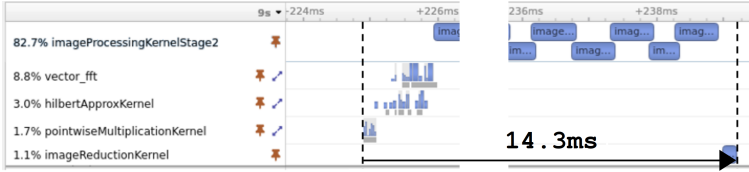


Figure 7: Nvidia Nsight Systems timeline for the second optimisation stage. A specialised image processing kernel per data block massively reduces the overhead through many small kernel launches.

diated stages of this process is presented and compared in this section. Those stages of the development process are described to explain the different design decisions and their effect on the overall performance. For comparison, all described tests are performed on the same hardware with 970 simulated input signals ($N = 1024, N_p = 8192$) and a target area size of 512×512 pixel. The execution time is measured with the analysis software Nvidia Nsight Systems, and the GPU activity is observed with Nvidia Nsight Compute.

The first attempt to implement the image processing does not involve any specialised kernel functions but only uses general arithmetic operations. This stage exclusively aims to check the general functionality and to determine the upper bound of performance improvement. All operations in figure 6 are executed sequentially with a total execution time of 110.0ms.

As a first optimisation step, the input data is divided into different blocks to enable the parallel execution of multiple kernel functions on the GPU. Each processing block generates partial image data, which is combined to the final image when all processing blocks are finished. The assignment of a dedicated CPU thread and CUDA stream to each block reduces the overall execution time to 94.3ms.

The most obvious drawback of the previous implementation is the high amount of small kernel launches, which comes with a large overhead compared to a single specialised kernel. The calculation of the antenna distance, the interpolation of the FFT data and the application of the complex phase correction consists of five kernel launches for each signal. The idea is now to combine those steps in one kernel launch per block and to loop over the signals inside the kernel function. The result in figure 7 is a reduction of the overall execution time down to 14.3ms.

The most significant proportion of the execution time is still caused by the specialised image processing kernel, and reducing its execution time will have a large impact on the overall execution time. Further optimisations and the analysis of a single kernel need a deeper look into the GPU hardware. The kernel analysis tool Nvidia Nsight Compute measures various metrics of a kernel on the hardware layer. That includes bandwidth measurements, the utilisation of the different memory types, cache and hit-rate analysis and the utilisation of the different GPU pipelines.

The analysis of the image processing kernel reveals a very low L2 cache hit-rate of 5.66% and a very high utilisation of the GPU integer multiplication and floating point operation pipeline (FMA). Both problems are targeted by transferring the FFT data into a batched read-only 1D-texture in which each row is filled with the complex spectrum of a single input signal. Thus, the GPU is able to perform more aggressive caching by ignoring possible write operations and predicting the memory access for adjacent rows. Another huge advantage of the texture memory is the ability to perform a hardware interpolation during memory access which relieves the FMA unit. The analysis of the optimised kernel shows a L1 and L2 cache hit-rate of over 95%, a balanced utilisation of the used GPU pipelines and an overall execution time of 6.1ms.

5 Conclusion

The new imaging radar algorithm was successfully implemented in CUDA C++ and was used to perform initial simulations and to estimate the expected signal processing time of the sensor system. Besides

general optimisations, like the usage of streams and the step-by-step kernel runtime optimisation, the main outcome is the suitability of the GPU texture memory for the back-projection algorithm. A first sensor prototype is in the making and will be used for further tests and to verify the results of this work. The code developed during this work is now the foundation for a general-purpose CUDA signal processing library, which will be used and extended in further projects.

References

1. C. Schwäbig, S. Wang, and S. Gütgemann, "Development of a millimetre wave based sar real-time imaging system for three-dimensional non-destructive testing," *tm - Technisches Messen*, vol. 88, no. 7-8, pp. 488–497, 2021.
2. J. Romstadt, H. Papurcu, A. Zaben, S. Hansen, K. Aufinger, and N. Pohl, "Comparison on spectral purity of two size d-band frequency octuplers in mimo radar mmics," in *2021 IEEE BiCMOS and Compound Semiconductor Integrated Circuits and Technology Symposium (BCICTS)*, 2021, pp. 1–4.
3. E. Tolin, M. A. Campo, H. Cetinkaya, R. Herschel, S. Gütgemann, C. Krebs, and S. Bruni, "Uwb millimeter-wave 1d mimo array for non-destructive testing," in *2021 15th European Conference on Antennas and Propagation (EuCAP)*, 2021, pp. 1–5.
4. M. Ortner, Z. Tong, and T. Ostermann, "A millimeter-wave wide-band transition from a differential microstrip to a rectangular waveguide for 60 ghz applications," in *Proceedings of the 5th European Conference on Antennas and Propagation (EUCAP)*, 2011, pp. 1946–1949.
5. L. Marple, "Computing the discrete-time "analytic" signal via fft," *IEEE Transactions on Signal Processing*, vol. 47, no. 9, pp. 2600–2603, 1999.
6. B. Stroustrup and H. Sutter. (2021) C++ core guidelines. Last visited 29.09.2022 15:48. [Online]. Available: <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>