# Optimized Minimum-Search for SAR Backprojection Autofocus on GPUs Using CUDA

Niklas Rother*, Christian Fahnemann*, Jan Wittler*, Holger Blume*
*Institute of Microelectronic Systems (IMS)
Leibniz University Hannover, Appelstrasse 4, 30167 Hannover, Germany
rother@ims.uni-hannover.de, fahnemann@ims.uni-hannover.de, blume@ims.uni-hannover.de

***Abstract*—Autofocus techniques for synthetic aperture radar (SAR) can improve the image quality substantially. Their high computational complexity imposes a challenge when employing them in runtime-critical implementations. This paper presents an autofocus implementation for stripmap SAR specially optimized for parallel architectures like GPUs. Thorough evaluation using real SAR data shows that the tunable parameters of the algorithm allow to counterbalance runtime and achieved image quality.**

***Index Terms*—backprojection, autofocus, SAR, GPU, CUDA**

## I. Introduction

Synthic Apeture Radar (SAR) data processing is a computationally expensive task. While frequency-domain based algorithms like the Range-Doppler-Alogrothm (RDA) or the Wavenumber-Domain-Algorithm (WKA) have a comparatively low complexity of $\mathcal{O}(N^2 logN)$, their requirement of straight flight paths makes them inappropriate for small airborne platforms. Time-domain based techniques like the Global-Backprojection-Algorithm (GBP) on the other hand can handle arbitrary trajectories, including excessive swaying [1], but have a complexity of $\mathcal{O}(N^3)$.

While the GBP can handle arbitrary trajectories, even small errors in the acquired position (in the dimension of one wavelength) can lead to severe errors in the generated image. Especially for small airborne platforms, the required accuracy in position tracking is a challenge. This gives rise to the idea of SAR autofocus algorithms, which correct small errors in position tracking to increase the image sharpness. The most common autofocus for time-domain backprojection was proposed by Duersch and Long [2]. They propose to use a coordinate-descent optimization to find a phase correction factor for every pulse. The algorithm is computationally very expensive, needing more than three times the computational costs of the backprojection algorithm.

Fahnemann, et al. [3] showed, that the autofocus algorithm by Duersch and Long is very well suited for the application of FPGAs and proposed hardware-related optimizations for an efficient parallel execution, termed Parallel Autofocus Optimization (PAFO).

To offer a high-quality live preview of the acquired SAR data, the image formation must be processed in real time. Because of the high data rate and very low compressibility of raw SAR data and the limited downlink bandwidth, live ground-based processing is often not possible. Instead, the image formation has to be done on-board. The resulting image can then be compressed by lossless or lossy image compression algorithms and sent to the ground station. The use of energy efficient and small processing platforms like FPGAs and embedded GPUs make real-time SAR data processing possible even on small airborne platforms with very stringent requirements on energy and payload size.

In this paper, the optimized parallel SAR autofocus algorithm (PAFO) by Fahneman et al. is implemented on an embedded GPU. The implementation features high image quality together with low processing runtime. Combined with the small form factor and the low energy requirements of an embedded GPU, the use on a small airborne platform for live image formation is feasible. Additionally, the implementation can be used to swiftly evaluate different parameters and optimizations in the context of a prospective FPGA implementation.

This paper is organized as follows. In section II a short overview of the GPB and PAFO algorithm is given. Section III describes the GPU implementation of the aforementioned algorithms, which is then evaluated in section IV. A summary of the findings is given in section V.

### A. Related work

Due to the inherent parallel nature of SAR image formation, the use of GPUs for this task has found widespread application. Fasih et al. [4] achieved a 60× improvement compared to a CPU implementation on SAR backprojection using GPUs. Benson et al. [5] used a cluster of multiple GPUs to reach further performance gains. Both used NVIDIA Tesla GPUs, whose high power consumption ($> 200\,\mathrm{W}$) is forbidding on small airborne platforms.

The use of embedded GPUs for SAR imaging was explored by Fatica et al. [6]. Their implementation is based on frequency-domain algorithms and does not include an autofocus technique.

The impact of using either single or double precision on power consumption and image quality was studied by Portillo et al. [7]. They found the single precision variant to be more energy efficient with little impact on image quality.

## II. Parallel autofocus (PAFO) algorithm

The captured data of a standard strip-map SAR scenario consists of $M$ antenna positions. At each position, a radar pulse is transmitted, and the received echo is stored as a series of $N$ discrete range-bins[1], yielding a data set $\mathbf{D} \in \mathbb{C}^{M\times N}$ with elements $d_{m,n}$. The final SAR image $\mathbf{G} \in \mathbb{C}^{U\times V}$ with elements $g_{u,v}$ is then assembled using the backprojection formula

$$g_{u,v} = \sum_{m=0}^{M-1} d_{m,\tilde{n}} \cdot e^{i2k\Delta r} \tag{1}$$

where $\Delta r$ is the 3-D-space distance from the $m^{\text{th}}$ antenna position to the ground position of pixel $(u, v)$. Additionally, a constant wavenumber $k = 2\pi f_\text{c}/c$ (with carrier frequency $f_\text{c}$ and wave propagation speed $c$) is required to maintain phase coherent summation.

As denoted by the index $\tilde{n}$, the range-bin samples are interpolated using the fractional range-bin index

$$\tilde{n} = (\Delta r - r_0)/\delta r \tag{2}$$

Here, $r_0$ denotes the distance to the first range-bin ($n = 0$) and $\delta r$ is the size of a single range-bin.

### *A. Backprojection Autofocus [2]*

Imprecise measurement of the antenna position leads to erroneous $\Delta r$'s, which then lead to a defocused image. As shown in [2], for small error magnitudes the effect can be modeled using a pulse-dependent phase error. Using a vector of phase corrections $\vec{\Phi} \in [0, 2\pi)^M$ with elements $\phi_m$, each sample can be phase corrected. A focused image can then be obtained using

$$g_{u,v}(\vec{\Phi}) = \sum_{m=0}^{M-1} \left(d_{m,\tilde{n}} \cdot e^{i\phi_m}\right) \cdot e^{i2k\Delta r} \tag{3}$$

It is the autofocus algorithm's task to find the vector $\vec{\Phi}_\text{opt}$ which leads to the sharpest image. For this, a sharpness metric $S\colon \mathbb{R} \to \mathbb{R}$ is used to define a cost function

$$C(\vec{\Phi}) = -\sum_{u=0}^{U-1}\sum_{v=0}^{V-1} S\left(\left|g_{u,v}(\vec{\Phi})\right|\right) \tag{4}$$

as the negative sum of $S$, applied to all pixel magnitudes in the output image. The autofocus problem can therefore be formulated as the $M$-dimensional optimization problem

$$\vec{\Phi}_\text{opt} = \underset{\vec{\Phi}\in[0,2\pi)^M}{\arg\min}\; C(\vec{\Phi}) \tag{5}$$

In [2], a coordinate-descent approach is suggested to solve this problem. Starting with $\vec{\Phi}'_0 = \vec{0}$, $\phi_0$ is varied to minimize the cost function, while keeping the other elements constant. $\vec{\Phi}'_0$ is then updated with the found value, and the process is being repeated to find the value of $\phi_1$, etc. This yields the first iteration result $\vec{\Phi}'_1$. Depending on the required image quality, this vector can either be used as the final correction factor $\vec{\Phi}$, or as an initial guess for another iteration of the algorithm.

### *B. Optimization for parallel architectures [3]*

As presented in [3], several properties of the presented algorithm can be exploited to reach an efficient implementation on massively parallel architectures like FPGAs or GPUs.

*1) Exploiting linear characteristics of the backprojection:* The backprojection is a linear operation, it is therefore irrelevant if a phase correction is applied before the backprojection to the raw data, or afterwards to the pulse's contribution to every pixel.

Using the notation $\mathbf{G}_m$ for the backprojection of a single pulse only, the focused image thus can be written as

$$\mathbf{G}(\vec{\Phi}) = \sum_{m=0}^{M-1} \mathbf{G}_m \cdot e^{i\phi_m} \tag{6}$$

Hence, the resulting image $\mathbf{G}(\vec{\Phi}|\phi_m = \xi)$ and, more importantly, the cost function $C(\vec{\Phi}|\phi_m = \xi)$ for a candidate phase value $\phi_m = \xi$ can quickly be calculated using

$$\mathbf{G}(\vec{\Phi}|\phi_m = \xi) = \mathbf{G}(\vec{\Phi}|\phi_m = 0) + \mathbf{G}_m \cdot \left(e^{i\xi} - 1\right) \tag{7}$$

Assuming sufficient memory to store the partially focused image $\mathbf{G}(\vec{\Phi}')$ and the current single-pulse image $\mathbf{G}_m$, evaluation of the cost function for a candidate phase value can therefore be realized without performing a full backprojection of the image, reducing the number of required backprojections largely.

*2) Quadratic interpolation-based minimum search:* Different optimization algorithms can be used to find the optimal value for $\phi_m$ in each step of the autofocus algorithm. In [3] a quadratic interpolation-based minimum search is suggested, which allows parallel evaluation of different candidate values. This is well-suited to the inherently parallel architecture of FPGAs and can also be exploited on GPUs.

Analysis showed, that the impact of varying a single value $\phi_m$ of $\vec{\Phi}$ on the cost function $C(\vec{\Phi})$ is similar to a sinusoidal shape, especially when using $S : x \mapsto x^2$ or $S : x \mapsto x^4$. In the vicinity of the minimum, a sinusoidal shape may be approximated by a parabola. This gives rise to the idea of using quadratic interpolation to find the minimum.

The parallel minimum search works as follows. In a first step, the cost function $C$ is sampled at $\hat{S}$ equidistant points. All points can be evaluated in parallel. The found minimum value is then used to construct a narrower set of $\hat{S}$ sampling points around the minimum. This process is repeated $\hat{R}$ times. After the last iteration, the minimum index $\xi_0$ and its two neighbours $\xi_{-1}$ and $\xi_1$, together with the values $y_i = C(\vec{\Phi}'|\phi_m = \xi_i)$ for $i \in \{-1, 0, 1\}$ are used to construct a parabola. The whole process is visualized in Fig. 1. Exploiting the constant spacing of the sample points $\Delta\xi = \xi_0 - \xi_{-1} = \xi_1 - \xi_0$ and normalizing

[1]For other radar systems, like FMCW, a preprocessing step is necessary to transform the data to a pulse equivalent

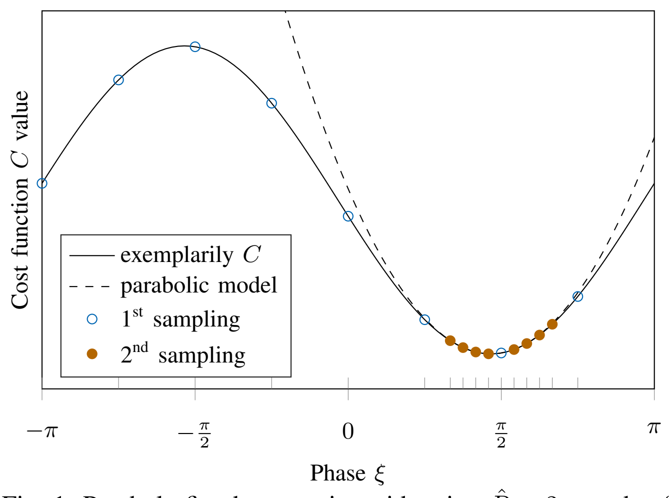

Fig. 1: Parabola fitted on an sinusoid, using $\hat{R} = 2$ rounds of $\hat{S} = 8$ samples [3].

the parabola offset to zero with $\tilde{y}_i = y_i - y_0$, the sought minimum can be found at

$$\phi_m = \xi_{\min} = \xi_0 + \frac{\Delta\xi}{2} - \Delta\xi \cdot \frac{\tilde{y}_1}{\tilde{y}_{-1} + \tilde{y}_1} \tag{8}$$

## III. GPU Implementation

The autofocus algorithm proposed in this paper was implemented on a Jetson AGX Xavier developer kit [8]. It features a Volta-generation GPU with 512 CUDA cores running at 1.5 GHz, as well as 8 ARMv8 CPU cores. On the Xavier platform, the CPU and GPU share the same physical memory, which allows for zero-copy data sharing between GPU and CPU. The software was targeting CUDA 10.0 and was running on the default Ubuntu Linux with kernel 4.9.

The implementation is organized around three memory buffers, each capable of storing a single image at full resolution $U \times V$. One buffer is used for the partially focused image $\mathbf{G}$ and two buffers $\hat{\mathbf{G}}_m$ and $\hat{\mathbf{G}}_{m+1}$ for the current and next single pulse projections, respectively. The use of two buffers for the single pulses makes it possible to start the projection of the next pulse, before the processing of the current one is finished. This yields a higher GPU occupancy, at the cost of higher memory requirements.

As shown in Fig. 2 three kernels (special functions executed on the GPU) are used for the backprojection (GBP), the evaluation of the cost function and the update of the partially focused image $\mathbf{G}$. The determination of the $\hat{S}$ sampling points and the quadratic interpolation are performed on the CPU, because these tasks do not benefit from parallel processing.

TABLE I: Found optimal processing granularity used for the processing kernels

| Kernel | Pixels/Thread | Threads/Block |
|---|---|---|
| Initial GBP | $2 \times 2$ | $16 \times 16$ |
| Single Pulse GBP | $2 \times 2$ | $4 \times 4$ |
| Evaluate Cost Function | $1 \times 1$ | $1 \times 32$ |
| Update Image | $2 \times 2$ | $8 \times 8$ |

The GPU kernels are executed using many threads in parallel, organized in a 2D grid, which naturally fits the two dimensional structure of the images. The used optimal granularity is shown in Table I. For the backprojection and the image update every thread processes an image tile of $2 \times 2$ pixels instead of a single pixel. For image sizes not divisible by the thread count, a hardware feature of the GPU is used to discard out-of-bound writes.

The cost function kernel evaluates all $\hat{S}$ sampling points iteratively. Similar to the proposed FPGA architecture [3], this saves memory bandwidth, as the values for $g_{u,v}$ and $\hat{g}_{u,v}$ have to be loaded only once. The iterative processing makes the kernel runtime dependent on $\hat{S}$. This is different to the FPGA architecture concept, where an increase of $\hat{S}$ can be handled with more parallel hardware blocks, increasing the chip area, but not the runtime.

The calculated per-pixel cost values are summed up in a tree-like reduction scheme using the CUB library [9]. The final reduction step is then performed on the CPU, using Kahan summation to avoid floating point errors. This technique uses a separate running compensation to reduce precision loss when small numbers are added together in a large sum [10].

## IV. Evaluation

The described architecture was evaluated using an in-house dataset captured using an ultra-wideband FMCW sensor in a laboratory demonstrator setup [11]. To simulate the effects of an unstable flight path and/or errors in position tracking, the recorded path is distorted using a high-frequency sine function. The scene, as shown in Fig. 3, can be classified as municipal and features multiple houses as well as a strong corner reflector near the center of the scene. Image quality was measured using peak signal-to-noise ratio (PSNR) defined as

$$\mathrm{MSE} = \frac{1}{U \cdot V} \sum_{u=0}^{U-1} \sum_{v=0}^{V-1} (|g_{u,v}| - |h_{u,v}|)^2 \tag{9}$$

$$\mathrm{PSNR~(dB)} = 10 \cdot \log_{10} \left( \frac{\max(|h_{u,v}|)^2}{\mathrm{MSE}} \right) \tag{10}$$

for the image $\mathbf{G}$ compared to a reference image $\mathbf{H}$.

Algorithm run time was measured using CUDA *events*, which implement performance counters directly on the GPU. All runtimes are averaged over 10 runs. The used dataset consists of 742 range lines with 1404 range bins each and was projected to an $512 \times 512\,\mathrm{px}^2$ image. The power budget of the Jetson AGX Xavier board was set to 15 W.

### A. Impact of floating point precision on image quality

The GPU can operate on floating point numbers with either single, double or half precision. The impact on performance and image quality is shown in Table II. The choice of data width not only impacts the time needed for the actual arithmetic operations, but also the required memory bandwidth, which explains the more than twofold increase from float to double. Half precision arithmetic has a very limited range, which must be taken into account to avoid overflows.

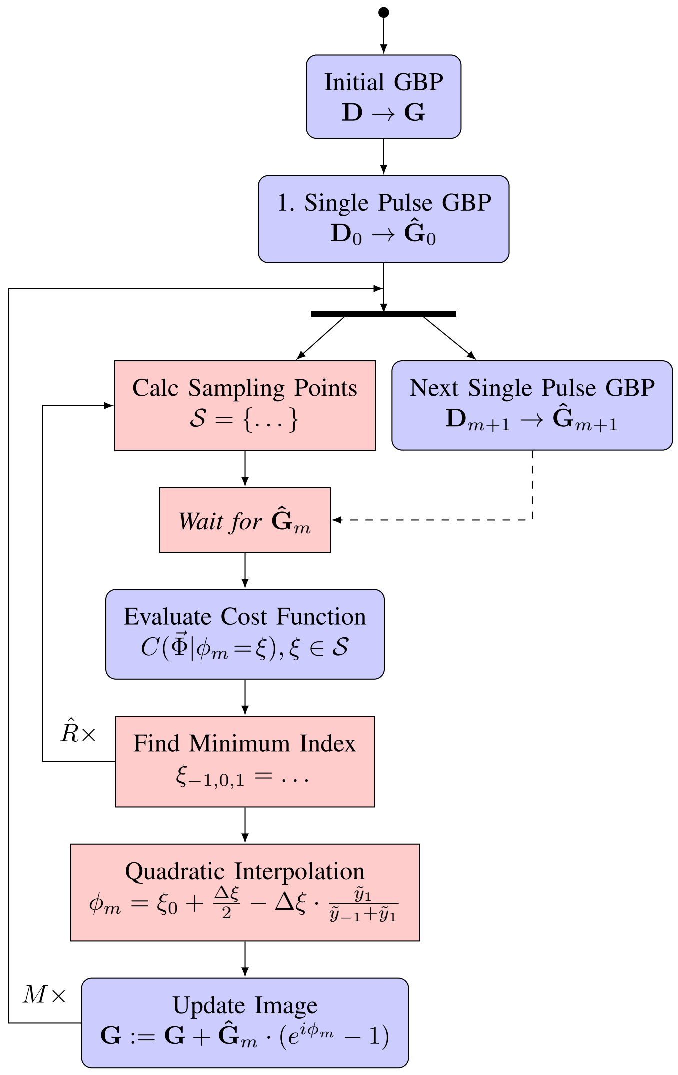
Initial GBP
$\mathbf{D} \rightarrow \mathbf{G}$
1. Single Pulse GBP
$\mathbf{D}_0 \rightarrow \hat{\mathbf{G}}_0$
Calc Sampling Points
$\mathcal{S} = \{\dots\}$
Next Single Pulse GBP
$\mathbf{D}_{m+1} \rightarrow \hat{\mathbf{G}}_{m+1}$
*Wait for* $\hat{\mathbf{G}}_m$
Evaluate Cost Function
$C(\vec{\Phi}|\phi_m = \xi), \xi \in \mathcal{S}$
$\hat{R}\times$
Find Minimum Index
$\xi_{-1,0,1} = \dots$
Quadratic Interpolation
$\phi_m = \xi_0 + \frac{\Delta\xi}{2} - \Delta\xi \cdot \frac{\tilde{y}_1}{\tilde{y}_{-1} + \tilde{y}_1}$
$M\times$
Update Image
$\mathbf{G} := \mathbf{G} + \hat{\mathbf{G}}_m \cdot (e^{i\phi_m} - 1)$

Fig. 2: Flow chart of the proposed autofocus architecture, showing the parts executed on the GPU and CPU.

Using single precision has, compared to double precision, very little impact on image quality (PSNR $> 75$ dB). This is in line with the findings of Portillo et al. [7]. The runtime is cut more than tenfold. Analysis of the double precision code with the NVIDIA Visual Profiler showed, that the implementation fails to reach a high GPU occupancy in this case. If a higher image quality is desired, only the summation reduction can be executed using double precision. This increases the

TABLE II: Effect of floating point precision on the image quality in different parts of the algorithm. The all-double version was used as a reference. Results for $\hat{S} = 8$ and $\hat{R} = 2$.

| Algorithm part | | | | |
|---|---|---|---|---|
| GBP | Storage | Reduction | PSNR (dB) | Runtime (ms) |
| double | double | double | $\infty$ | 10 087.36 |
| float | float | double | 81.33 | 2808.10 |
| float | float | float | 76.90 | 607.40 |
| float | half | float | 73.65 | 510.50 |

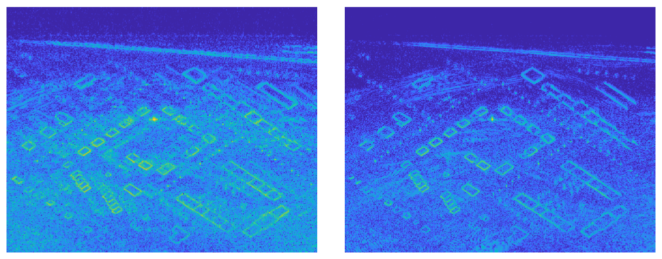
(a) before autofocus processing (b) after autofocus processing

Fig. 3: Image results of an in-house dataset [11] $\hat{R} = 3, \hat{S} = 12$. An artificial high-frequency path deviation was added to the image to simulate an unstable flight path.

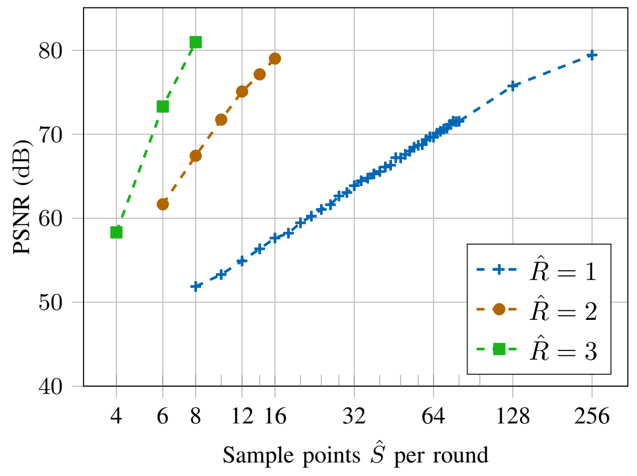

Fig. 4: Resulting image quality for different sampling parameters $\hat{S}$ and $\hat{R}$.

PSNR to 81 dB, but also leads to a prolonged runtime. The opposite effect can be achieved by using half precision for the intermediate results $\mathbf{G}$ and $\hat{\mathbf{G}}_m$; this version shows a slightly lower PSNR (74 dB) accompanied by a slightly shorter runtime. Using half precision arithmetic in other cases leads to overflows.

### *B. Impact of algorithm parameters $\hat{S}$ and $\hat{R}$*

The number of samples evaluated per round $\hat{S}$ and the number of sampling rounds $\hat{R}$ are the main tuning parameters of the algorithm. As shown in Figs. 4, 5 and 6, these parameters can be used to counterbalance runtime and achieved image quality. The reference image used for the PSNR comparison was generated using $\hat{S} = 12, \hat{R} = 3$. All computations were done using single precision numbers.

The image quality is linearly correlated with the iteration number, matching the simulations of Fahnemann et al. [3]. As expected, configurations leading to the same number of effective samples, like $\hat{S} = 64, \hat{R} = 1$ and $\hat{S} = 8, \hat{R} = 2$ result in about the same image quality, but differ in runtime, since the latter only evaluates $2 \times \hat{S} = 16$ samples.

Unlike in the FPGA architecture concept [3], an increase of $\hat{S}$ does increase the runtime nearly linearly because the

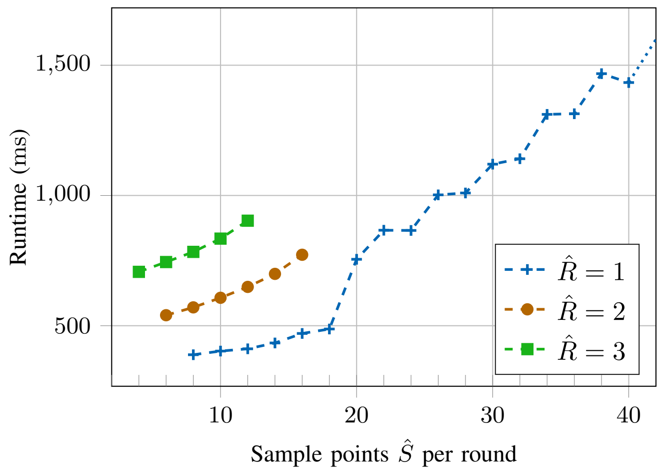

Fig. 5: Algorithm runtime for different sampling parameters $\hat{S}$ and $\hat{R}$.

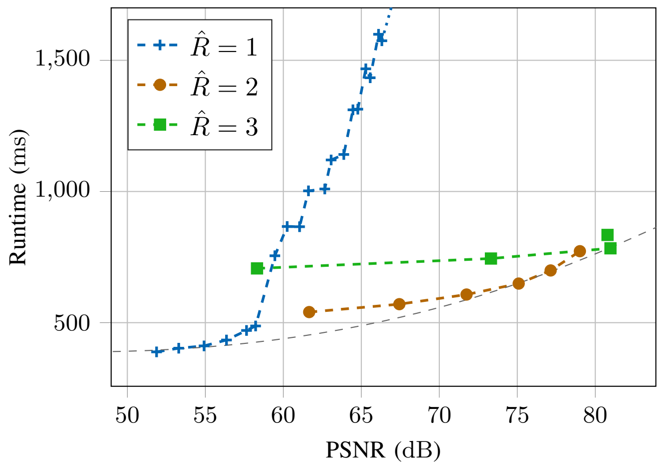

Fig. 6: Image quality to runtime comparison. The dashed gray line shows the Pareto frontier.

samples are evaluated iteratively. In the FPGA concept all samples can be evaluated in parallel using dedicated hardware blocks, therefore an increase in $\hat{S}$ does only increase the required hardware elements and not the runtime. For $\hat{R} = 1$, at $\hat{S} = 20$ the runtime increases abruptly. This can be attributed to increasing register pressure and/or shared memory requirements.

The Pareto frontier in Fig. 6 shows that for the most useful quality section of 60 dB to 80 dB PSNR the optimal runtime can be achieved using $\hat{R} = 2$. If an image quality of 55 dB PSNR is sufficient, a faster processing can be reached using only a single sampling round.

## V. Conclusion

This paper presents an efficient parallel SAR autofocus implementation on a GPU based on the FPGA architecture concept by Fahnemann et al. [3]. The algorithm exposes tunable parameters to counterbalance desired runtime and image quality. Thorough evaluation of the implementation using real SAR data was carried out, showing the intended scaling behavior.

The GPU implementation additionally serves as a working prototype for a prospective FPGA implementation and allows quick exploration of algorithm changes and parameter choices.

## Acknowledgment

The authors acknowledge the support by the WTD-81.

## References

[1] A. Sommer and J. Ostermann, "Explicit motion compensation for backprojection in spotlight SAR," in *2016 17th International Radar Symposium (IRS)*, IEEE, 2016, pp. 1–4.

[2] M. I. Duersch and D. G. Long, "Backprojection autofocus for synthetic aperture radar," 2013.

[3] C. Fahnemann, D. Fallnich, A. Sommer, F. Cholewa, and H. Blume, "Hardware-optimized minimum-search for SAR backprojection autofocus on FPGAs," in *2019 International Radar Conference*, IEEE, 2019, pp. 1–5.

[4] A. Fasih and T. Hartley, "GPU-accelerated synthetic aperture radar backprojection in CUDA," in *2010 IEEE Radar Conference*, IEEE, 2010, pp. 1408–1413.

[5] T. M. Benson, D. P. Campbell, and D. A. Cook, "Gigapixel spotlight synthetic aperture radar backprojection using clusters of GPUs and CUDA," in *2012 IEEE Radar Conference*, IEEE, 2012, pp. 853–858.

[6] M. Fatica and E. Phillips, "Synthetic aperture radar imaging on a CUDA-enabled mobile platform," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, IEEE, 2014, pp. 1–5.

[7] R. Portillo, S. Arunagiri, P. J. Teller, S. J. Park, L. H. Nguyen, J. C. Deroba, and D. Shires, "Power versus performance tradeoffs of GPU-accelerated backprojection-based synthetic aperture radar image formation," in *Modeling and Simulation for Defense Systems and Applications VI*, International Society for Optics and Photonics, vol. 8060, 2011, p. 806 008.

[8] NVIDIA, *Jetson AGX Xavier developer kit.* [Online]. Available: https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit.

[9] D. Merrill, *CUDA UnBound (CUB) library.* 2018. [Online]. Available: http://nvlabs.github.io/cub/.

[10] W. Kahan, "Further remarks on reducing truncation errors," *Communications of the ACM*, vol. 8, no. 1, p. 40, 1965.

[11] M. Wielage, F. Cholewa, P. Pirsch, and H. Blume, "Experimental violation of the start-stop-approximation using a holistic rail-based UWB FMCW-SAR system," in *Proceedings of EUSAR 2016: 11th European Conference on Synthetic Aperture Radar*, VDE, 2016, pp. 1–4.