Journal of
Applied Remote Sensing

RemoteSensing.SPIEDigitalLibrary.org

# General purpose graphic processing unit implementation of adaptive pulse compression algorithms

Jingxiao Cai
Yan Zhang

SPIE.

Jingxiao Cai, Yan Zhang, "General purpose graphic processing unit implementation of adaptive pulse compression algorithms," *J. Appl. Remote Sens.* **11**(3), 035009 (2017),
doi: 10.1117/1.JRS.11.035009.

# General purpose graphic processing unit implementation of adaptive pulse compression algorithms

**Jingxiao Cai[a,b,*] and Yan Zhang[a,b]**
[a]Advanced Radar Research Center, Intelligent Aerospace Radar Team, Norman, Oklahoma, United States
[b]University of Oklahoma, School of Electrical and Computer Engineering, Norman, Oklahoma, United States

**Abstract.** This study introduces a practical approach to implement real-time signal processing algorithms for general surveillance radar based on NVIDIA graphical processing units (GPUs). The pulse compression algorithms are implemented using compute unified device architecture (CUDA) libraries such as CUDA basic linear algebra subroutines and CUDA fast Fourier transform library, which are adopted from open source libraries and optimized for the NVIDIA GPUs. For more advanced, adaptive processing algorithms such as adaptive pulse compression, customized kernel optimization is needed and investigated. A statistical optimization approach is developed for this purpose without needing much knowledge of the physical configurations of the kernels. It was found that the kernel optimization approach can significantly improve the performance. Benchmark performance is compared with the CPU performance in terms of processing accelerations. The proposed implementation framework can be used in various radar systems including ground-based phased array radar, airborne sense and avoid radar, and aerospace surveillance radar.  [DOI: 10.1117/1.JRS.11.035009]

**Keywords:** airspace surveillance radar; adaptive pulse compression; general purpose graphic processing unit; parallel computing; radar data processing chain.

Paper 170310P received Apr. 11, 2017; accepted for publication Jul. 18, 2017; published online Aug. 17, 2017.

## 1 Introduction

There are many existing applications of graphic processing unit (GPU)-based radar processing implementations, such as synthetic aperture radar processing[1,2] and constant false alarm rate processing.[3,4] Many of the applications show the potential of acceleration using GPUs collaborating with CPUs,[5,6] such as one thousand times acceleration of image formation over using CPUs alone.[7] The performance of actual processing implementations has variations, and a well-organized approach to achieve optimal performance for surveillance radar processing has not been available.

The purpose of this study is initial implementation of a type of advanced high-level surveillance radar algorithms, called adaptive pulse compression (APC)[8–10] in a GPU environment. APC algorithms are based on existing pulsed compression (PC) algorithms in solid-state radar and offer reduced sidelobes and enhanced resolution. As such, they are important for downward-looking high-altitude airborne and space radars.[11] There are multiple versions of APC algorithms, including reiterative minimum-mean-square error (RMMSE)[8] and RMMSE based on matched filter (MF-RMMSE) output.[10] Implementation of APC algorithms in real time will allow for fast remote sensing image formation for weather observation.[12] The challenge of this implementation with parallel computing has been the reiterative nature of APC algorithms, as well as the latencies and memory constraints for matrix inversions. The new generation of

*Address all correspondence to: Jingxiao Cai, E-mail: anyech@ou.edu

**Table 1** Assumed surveillance radar system parameters.

| | |
|---|---|
| Waveform | LFM and phased coding |
| Scanning mode | Plan position indicator (PPI) scanning |
| # of azimuth (range profiles) | Varies in tests |
| # of range gates | 128 to 8192 |
| # of pulses per azimuth | Varies in tests |
| Target types | Weather and point targets |

GPU devices, on the other hand, provides an opportunity for breaking through some of these limitations. The assumed radar system parameters are summarized in Table 1.

As there has been no prior report of implementing APC algorithms using GPUs, this study provides the first performance benchmark and application to actually measured radar remote sensing data. Similar to other adaptive algorithms, the APC algorithm is based on the "basic" algorithms such as fast Fourier transform (FFT), matrix multiplication, and matrix inversion. Since the focus is investigating how the "basic" algorithms can be integrated optimally for surveillance radar processing, and how they impact on the overall performance of the radar processing chain, we use the most popular compute unified device architecture (CUDA) libraries in these implementations and do not seek to optimize the "basic" algorithms themselves in GPU. For example, we mainly use standard CUDA toolkit (Version 8.0) and global memory space allocations for the implementation. Utilizing these libraries may not be the most efficient approach for a particular device or algorithm, but it maximizes the reusability of the implementations on different platforms, especially for future upgrades.

Meanwhile, optimization of different "basic" algorithms using different advanced techniques is explored and some results are presented. The focus of this work is optimizing the key parameters in kernel configurations using different GPU devices and optimizing the trade-off between processor capability and data communication overheads.

## 2 Algorithms

This study focuses on three algorithms in the "pulse compression" category in Fig. 1. They are basic MF, RMMSE filter, and MF-RMMSE. These algorithms are key to the APC algorithm family and important to the resolutions and sidelobe levels of radar range processing.

### 2.1 *Matched Filter*

The benchmark test procedure for basic matched filtering is as follows: first, simulated ground truth, waveform, and returned signal are generated. Here, an LKP3 phase-coded waveform is

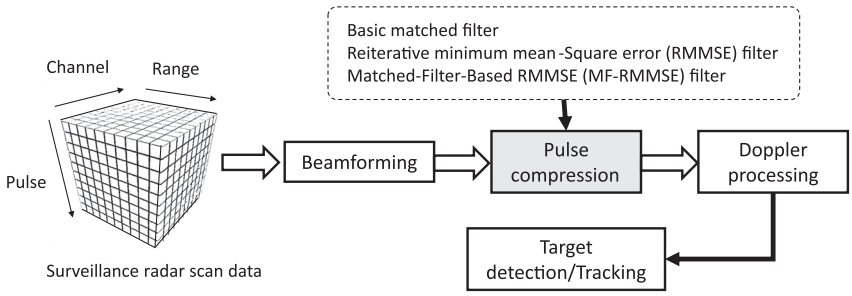


**Fig. 1** The basic processing chain of pulsed-Doppler radar.

used as the simulated surveillance radar waveform. Second, zero padding is applied to the waveform as well as the returned signal, to reach the nearest length $l = 2^a 3^b 5^c 7^d$, where $a$, $b$, $c$, and $d$ are integers, in order to achieve the optimal FFT performance.[13] Third, an MF is applied to the signals as described in Eqs. (1)–(5), with different implementations of FFT/inverse FFT (iFFT) and multiplication. Since the first two steps are common to all the implementations of the MF, time latencies are only measured from the third step

$$w_r(n) = w(-n), \tag{1}$$

$$W(N) = \mathrm{FFT}[w_r(n)], \tag{2}$$

$$S(N) = \mathrm{FFT}[s(n)], \tag{3}$$

$$M(N) = W(N) \times S(N), \tag{4}$$

$$m(n) = \mathrm{iFFT}[M(N)], \tag{5}$$

where $w$, $s$, and $m$ represent the waveform, returned signal, and MF output, respectively. Subscript $r$ means a reversed version of the respective data. Lower case letters represent data in time domain (before Fourier transform), whereas upper case letters represent data in frequency domain (after Fourier transform). FFT[·] and iFFT[·] represent FFT and iFFT computation. The pseudocode of MF is provided in Algorithm 1.

## 2.2 *RMMSE*

RMMSE is one of the APC algorithms based on the MMSE approach.[8] It performs significantly well in terms of recovering the truth data from the measurement. However, this outstanding

**Algorithm 1** Matched filter.

```
function MF (s, wf)
  Input: Two vectors of complex numbers s and wf.
  // s and wf represent the returned signal and waveform, respectively
  Output: One vector of complex numbers mf.
  // mf represents the match filter output
if length(s)! = length(wf) then
    zero padding the shorter vector to make length(x) == length(wf)
end
wf = reverse(wf.conj());          // reverse and conjugate wf
s = circRightShift(s, 1);         // do a circular right shift on s
w̄f = fft(wf)
s̄ = fft(s)
m̄f = s̄ × w̄f;                      // the multiplication is element-wise
mf = ifft(m̄f)
return mf
  // the red part of algorithm could be implemented with EIGEN(CPU) or CUDA(GPU)
```

performance comes with a price, which is a high computation load. The extremely high computation load limits the application of this algorithm, making it improbable to be applied on the massive size of data from various high definition and/or rapid updating observation tasks. The pseudocode of RMMSE is provided in Algorithm 2.

## 2.3 *MF-RMMSE*

MF-RMMSE is a modified version of RMMSE based on MF output which is proposed and derived in Ref. 10. It successfully reduces the computational load by the processing of MF outputs. The computation complexities of MF, RMMSE, and MF-RMMSE are listed in Table 2. The pseudocode of MF-RMMSE is provided in Algorithm 3.

**Algorithm 2** RMMSE.

function RMMSE $(s, wf, \text{num}_{\text{iter}})$

**Input:** Two vectors of complex numbers $s$ and $wf$.

// $s$ and $wf$ represent the **returned signal** and **waveform**, respectively

// $\text{num}_{\text{iter}}$ represents the number of iterations in RMMSE

**Output:** One vector of complex numbers.

Do zero-padding at the head of $s$; // $\hat{s} = [0 \ldots 0 \ s]$

Initialize noise matrix $R$; // $R$ is a diagonal matrix with the values of diagonal elements equal to noise power

initialize vector $\rho$; // $\rho$ is a vector of ones

for $\text{idx}_{\text{iter}} = 1$ to $\text{num}_{\text{iter}}$ do

  $\text{num}_{\text{cell}} = (2 \times \text{num}_{\text{iter}} - \text{idx}_{\text{iter}} - 1) \times [\text{length}(wf) - 1] + \text{length}(s)$

  for $\text{idx}_{\text{cell}} = 1$ to $\text{num}_{\text{cell}}$ do

    Initialize matrix $C$

    for $\text{idx}_{\text{in\_cell}} = 1$ to $\text{num}_{\text{in\_cell}}$ do

      Update matrix $C$ with zero padded shifted $wf$

    **end**

    $CR = C + R$

    $\hat{w} = CR \setminus wf$; // solving linear equations

    $\hat{w} = \hat{w} \times \rho[\text{idx}_{\text{cell}} + \text{length}(wf) - 1]$

    $\text{output}_{\text{temp}} = \hat{w}^{\intercal} \hat{s}[\text{subvector}]$

    // subvector starts at $\text{idx}_{\text{cell}} l + (\text{idx}_{\text{iter}} - 1) * [\text{length}(wf) - 1]$ with length of $wf$

  **end**

  $\rho = |\text{output}_{\text{temp}}|^2$; // element-wise operation

**end**

return $\text{output}_{\text{temp}}$

// the red part of algorithm could be implemented with **EIGEN(CPU)** or **CUDA(GPU)**

**Table 2** Computational complexity per range cell for different algorithms.

| Algorithms | Computation cost | Comments |
|---|---|---|
| MF | $O(N)$ | where $N$ is the length of waveform |
| RMMSE | $O(N^3)$ | |
| MF-RMMSE | $O(KN + K^3)$ | where $K$ is the length of filter used, and $K \ll N$ |

**Algorithm 3** MF-RMMSE.

function MF_RMMSE $(s, mf, \text{num}_{\text{iter}})$

**Input:** Two vectors of complex numbers $s$ and $mf$.

// $s$ and $wf$ represent the **returned signal** and **matched filter output**, respectively

// $\text{num}_{\text{iter}}$ represents the number of iterations in MF-RMMSE

**Output:** One vector of complex numbers.

$\rho$ = getMatrixRho();

$U$ = getMatrixU($s$);

$G$ = getMatrixG($s$);

$Sn$ = getMatrixSn($s$);

for $\text{idx}_{\text{iter}} = 1$ to $\text{num}_{\text{iter}}$ do

  **for** $\text{idx}_l = 1$ **to** length($mf$) **do**

    $G_2$ = updateMatrixG($S$, $G$, $\text{lim}_{\text{window}}$)

    // $\text{lim}_{\text{window}}$ is the size of processing window

    $D$ = getMatrixD($\rho$, $Sn$, $U$)

    $\hat{w} = \rho \cdot D \setminus G_2$; // solving linear equation

    $\text{output}_{\text{temp}}[\text{idx}_l] = \hat{w}^{\intercal} mf$; // vector multiplication

  **end**

  $\rho = |\text{output}_{\text{temp}}|^2$; // element-wise operation

**end**

return $\text{output}_{\text{temp}}$

// the red part of algorithm could be implemented with **EIGEN(CPU)** or **CUDA(GPU)**

// definitions about functions of *getMatrixs* are referred to the original article about MF-RMMSE[10]

## 3 GPU Testbed

In this section, the hardware and software environments of the GPU testbed will be described.

### 3.1 *Hardware*

The GPUs used in this study are TITAN Z and TITAN Xp, whose specifications are described in Table 3. In this study, only one of the two GK110 GPU cores of TITAN Z is used. Therefore, the

**Table 3** The GPUs we used in the current studies.

| Model | TITAN Z | TITAN BLACK | TITAN Xp |
|---|---|---|---|
| Code name | 2 × GK110 | GK110 | GP102 |
| Die size ($mm^2$) | 2 × 561 | 561 | 471 |
| Fab (nm) | 28 | 28 | 16 |
| Compute capability | 3.5 | 3.5 | 6.1 |
| Microarchitecture | Kepler | Kepler | Pascal |
| CUDA core config[a] | 2 × 2880:240:48 | 2880:240:48 | 3840:240:96 |
| Memory size (MB) | 2 × 6144 | 6144 | 12,288 |
| Memory bandwidth[b] | 2 × 336 | 336 | 547.7 |
| GFLOPS[c] | 8121.6 | 5120.6 | 12,150 |
| Release date | March 2014 | February 2014 | April 2017 |
| TDP (Watts) | 375 | 250 | 250 |

[a]Unified shader processors:texture mapping units:render output units.
[b]Gigabyte per second.
[c]Single precision.

TITAN Z used in this study will have the effective computation resource equivalent to TITAN BLACK, whose specification is also listed in Table 3. As a comparison, AMD FX 8150 is used as the reference CPU for this study and the CPU implementations utilize FFTW[14] and EIGEN[15] as counterpart libraries of CUDA.

### 3.2 *Software*

For the software environment configuration, Windows 10 is used as the operation system, NVIDIA driver version 382.05 and CUDA toolkit 8.0 [including CUDA fast Fourier transform (cuFFT) and CUDA basic linear algebra subroutines] are used as GPU computing support, MSVC++ 14.0 (Visual Studio 2015) and NVIDIA CUDA compiler (NVCC) 8.0 are used for compiling and linking C++ and CUDA C code, Eigen 3.3.3 is used as fundamental linear algebra library for CPU-based counterpart, and fastest Fourier transform in the West (FFTW) 3.3.5 is used as CPU-based FFT backend libraries for comparison.

## 4 Implementation and Optimization

Because both radar PPI scan and CUDA architecture have a hierarchy of three levels, it is reasonable to directly map those three layers as a guideline to distribute the computation resources. The initial mapping scheme is described in Fig. 2. However, after further investigations of particular algorithms, more realistic and efficient computation resources distribution schemes are discovered. In this section, the implementation and optimization schemes for particular algorithms will be discussed.

### 4.1 *Matched Filter*

The computation of MF only involves FFT and element-wise multiplication, thus it is feasible to take advantage of CUDA FFT library and THRUST to perform the computation without building customized CUDA kernels. The GPU version of MF algorithm follows the structure shown in Fig. 3. In this way, the impact of different kernel configurations on performance is less significant. As will be mentioned later in Sec. 5, different configurations of the kernel, i.e., THREADS

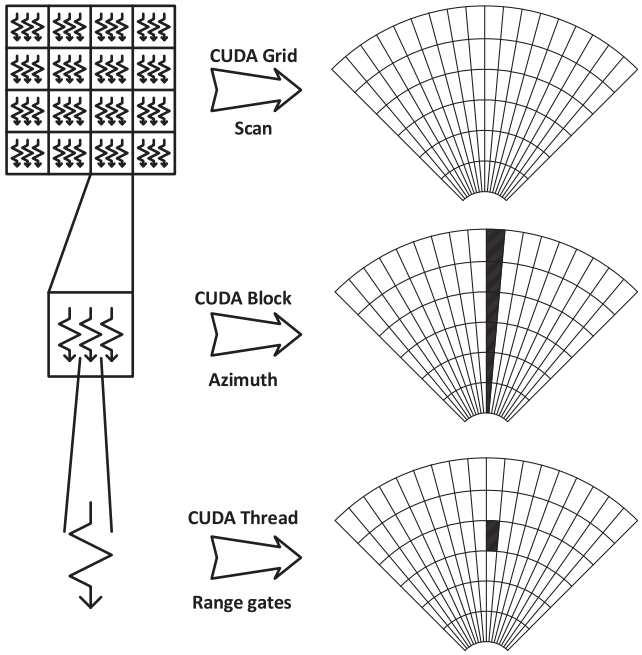


**Fig. 2** The hierarchy of CUDA abstract architecture and related radar PPI scan.

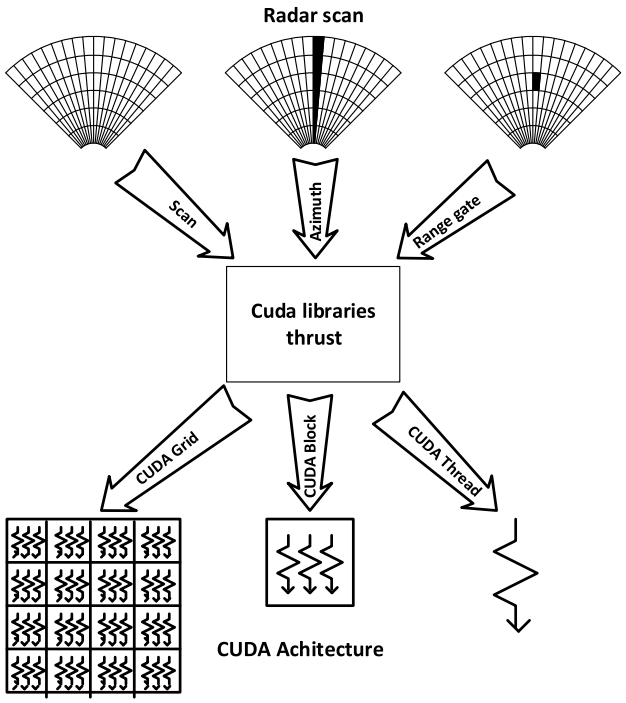


**Fig. 3** The configuration of processing power for MF.

PER BLOCK and BLOCKS PER GRID, will have a significant impact on the performance of the GPU version of other APC algorithms.

## 4.2 *RMMSE and MF-RMMSE*

RMMSE and MF-RMMSE are adaptive, iterative, and more complicated algorithms, and it would be better to utilize a divide-and-conquer technique, deploy the ability of nested

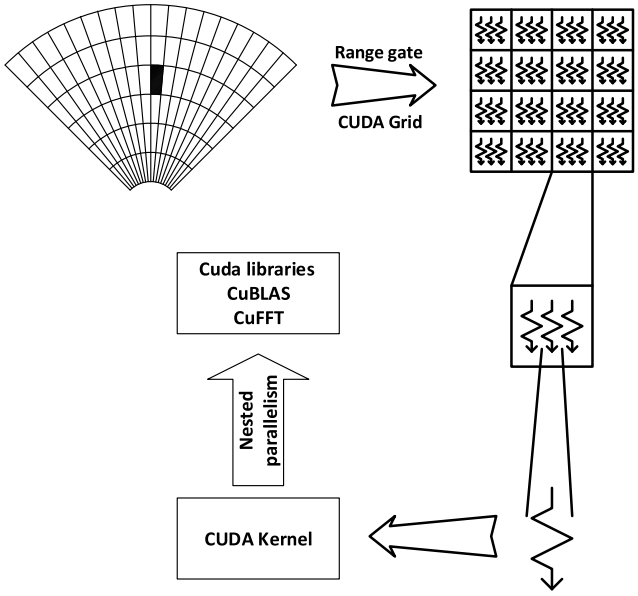


**Fig. 4** The configuration of processing power for RMMSE/MF-RMMSE.

parallelism of CUDA, and explore intrinsic parallelism of such algorithms, while it is applied on a small amount of data for each individual process. In addition, the high-level computation scheme of RMMSE and MF-RMMSE is depicted in Fig. 4. In this scheme, it is inevitable to manually configure the computing resource on the GPU. So, the key to reach the best acceleration performance is to optimally adjust the processing parameters with respect to specific data size. This will be further discussed in Sec. 5.3.

### 4.3 *Data Transfer*

One major issue of various GPGPU implementations is the bottleneck of data transfer between CPU (or host memory) to GPU (or device memory).[16] It would not be a fair performance comparison between CPU and GPU unless the cost of the data transfer has been treated appropriately. Because optimizing data transfer is beyond the scope of this work, the cost of data transfer is excluded from the GPU performance measurements in this paper, unless otherwise stated. A discussion of the overhead of data transfer will be given with a real-world application in Sec. 6.

### 4.4 *Memory Space Usage*

Shared memory is a type of memory in GPU which is faster but also smaller than global memory.[13] Utilizing shared memory on GPU would be beneficial in terms of processing speed in some applications.[17,18] However, because of the size restriction of shared memory, it is not feasible to utilize such memory in APC algorithms implementations. As described in Secs. 2.2 and 2.3, RMMSE and MF-RMMSE consist of several computations between fairly large size matrices and the smallest object of such algorithms is a large matrix, which could not be completely restored in the size restricted shared memory. Therefore, it would not benefit from utilizing shared memory without heavily altering the definitions and structures of these algorithms.

### 4.5 *Statistical Kernel Optimization*

One key factor, which has a critical impact on performance, is the configuration of CUDA kernel parameters. As shown in Fig. 5, there are three layers of both the CUDA abstract architecture and the GPU physical architecture.[16] The parameters BLOCKS PER GRID and THREADS PER BLOCK are user definable but the ratio between the GPU, streaming multiprocessors

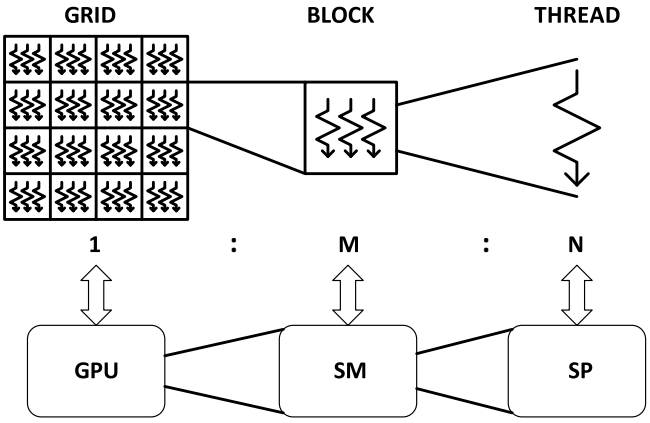


**Fig. 5** The hierarchy of CUDA abstract architecture and related GPU physical architecture.

(SMs), and streaming processors is fixed for a specific GPU device. In addition, the choice of BLOCKS PER GRID and THREADS PER BLOCK to get optimal performance depends on the type of algorithm to be implemented and the type of GPU to be used. There is another "layer" between the thread and grid existing for CUDA architecture called warp. A warp consists of 32 fixed threads and it is the basic operation group for SM to execute.[19] However, since the warp is not configurable by users, the layer of warp is omitted for seeking the optimal configuration in this research.

For our research, a statistical sampling and searching method is first used to determine the optimal configuration of CUDA kernel parameters for each algorithm implementation scenario (with variations in the size of data), based on testing several combinations of parameters in a given range for each scenario. Time latencies, aka the performance, for each set of parameters chosen on every individual scenario were recorded. Next, we ran the CPU counterpart of the algorithm on respective scenarios and recorded time latencies. Finally, we computed the GPU/CPU time latency ratios for each set of parameters on every individual scenario. By running statistically large samples of the parameter combinations, we will have a "performance map" for selecting the optimal kernel configuration. This technique will be illustrated in Figs. 10 and 13 for RMMSE and MF-RMMSE algorithms, respectively.

## 5 Benchmark Results

In this section, the APC algorithms' implementations described in Sec. 4 will be tested. Benchmark results of various data configurations and CUDA settings for each algorithm will be provided.

### 5.1 *Matched Filter*

According to the results shown in Fig. 6, the GPU implementation based on cuFFT is 5 to 30 times faster than the CPU implementation based on FFTW, as long as the dataset is sufficiently large. The performances of TITAN Z and TITAN Xp are nearly the same as that of TITAN Xp, holding a slight edge over TITAN Z.

Figure 7 shows us another view on the performance comparison of the MF. Comparing the performances of TITAN Xp and TITAN Z in Fig. 7, although TITAN Xp performs better than TITAN Z overall, trends of the amount of acceleration achieved with regard to the number and length of range profiles are the same for both of the GPUs used.

The white dashed line in Fig. 7 represents the equal performance between CPU and GPU platforms. It shows that a speed up of 30 times is achievable in several configurations of source profiles, as it could be seen that the acceleration curve is not quite smooth. The main reason for this phenomenon is that the cuFFT library is optimized when the length of processing time series is $l = 2^a 3^b 5^c 7^d$, where $a$, $b$, $c$, and $d$ are integers, and in addition, the smaller the prime factor is,

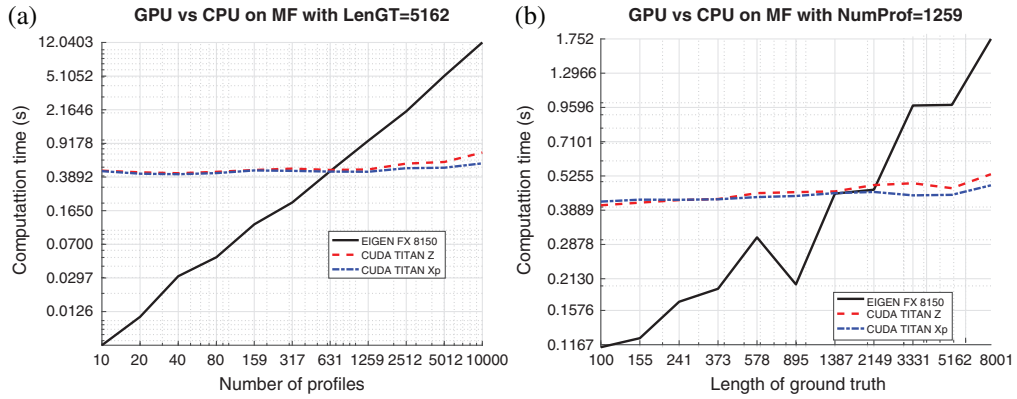


**Fig. 6** The performance of MF computing based on various libraries and processor usage. (a) Time latency used versus number of pulses, while length of data is fixed to 5162 sample points. (b) Time latency used versus length of range profile, while number of pulses is fixed to 1259.

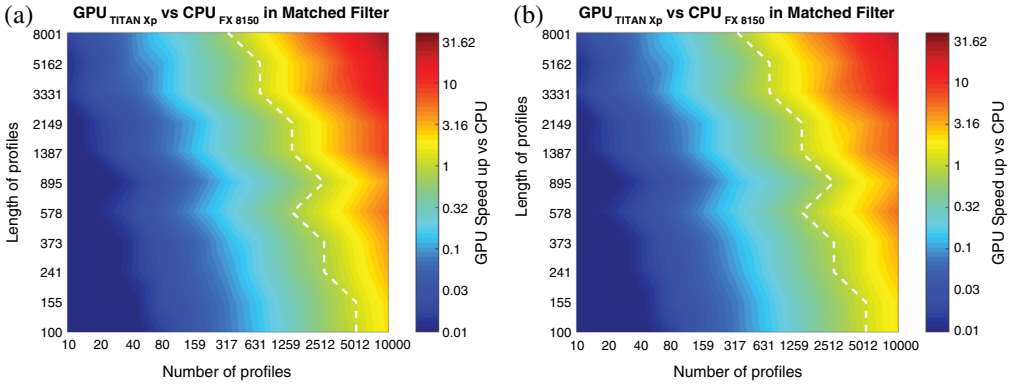


**Fig. 7** The performance comparison of MF based on GPU and CPU platforms. (a) TITAN Xp and (b) TITAN Z.

the better the performance would be.[13] Thus, along with the axis of the number of profiles, the acceleration performance of the GPU versus CPU monotonically increases. However, along with the axis of length of the profiles, the acceleration performance reaches its peak when the length of the profile is close to $2^a$. It is obvious that the larger the size of the data, the larger acceleration ratio can be obtained. However, the size of on-board memory limits further acceleration. When the memory size limit is reached, multiple data transfers are required to circumvent such a limit, and the data transferring process could be the bottleneck of such an implementation. However, the cost of data transfer can be compensated by utilizing another CUDA stream, and a more sophisticated computing scheme, by overlapping the computing and data transfer stream. Further acceleration would be expected and it will be exploited in future experiments.

## 5.2 *RMMSE*

The performance comparison of RMMSE implementations based on GPU and CPU platforms is demonstrated in Fig. 8. The results indicate that the length of the waveform (which is a transmitted pulse) has a larger impact on processing time compared with the length of ground truth (which is the impulse response of the range profile), which is implied in the description of its computation complexity listed in Table 2, and the GPU-based platform performs better when the length of either parameter mentioned above is larger, as it can be seen that 10 times acceleration is expected when the data size is sufficiently large.

Figure 9 shows another view on the performance comparison of RMMSE implementations. The white dashed line in Fig. 9 represents the equal performance between the CPU and GPU

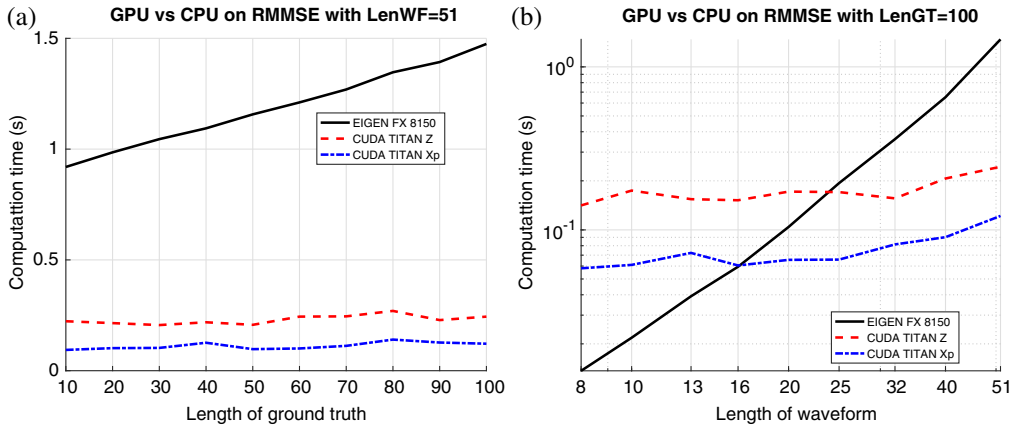


**Fig. 8** The performance comparison of RMMSE based on GPU and CPU platforms. Length of (a) waveform = 51 and (b) ground truth = 100.

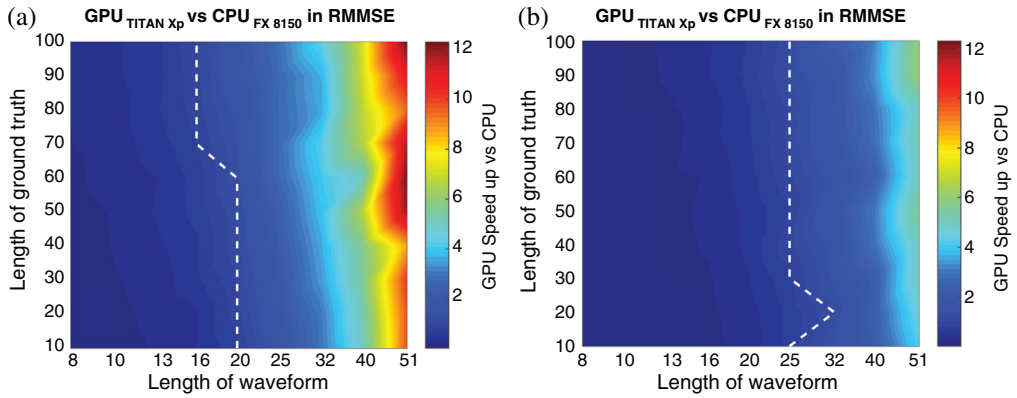


**Fig. 9** The performance comparison of RMMSE based on GPU and CPU platforms. (a) TITAN Xp and (b) TITAN Z.

platforms. It shows that 12 times speed up is achievable in several configurations of range profiles.

As mentioned in Secs. 4.1 and 4.5, the configuration of CUDA computing resources, i.e., THREADS PER BLOCK and BLOCKS PER GRID, has a major impact on the performance of the GPU version of algorithms. Plus, for more sophisticated algorithms, such as RMMSE and MF-RMMSE, utilizing a customized kernel is necessary to perform matrix-wise and element-wise manipulation as introduced in Secs. 2.2 and 2.3. Thus, properly distributing the GPU computing resources according to the problem data size is the key to achieve the optimum performance of the GPU implementations of these algorithms.

Figure 10 shows the impacts of different configurations of GPU computing resources on the acceleration performance of RMMSE algorithm implementations. The black circles in each figure represent the largest achievable speed up of the GPU compared with the CPU counterpart in specific lengths of waveforms, whereas the length of the ground truth is fixed at 100 sample points for all the cases, since it has less impact on the processing time as discussed earlier.

It can be seen from Fig. 10 that there exists an optimized computing resource configuration for each specific data size. For the application of RMMSE investigated in this study, we can conclude that the best configurations of CUDA are 4 or 8 for THREADS PER BLOCK and 16 or 32 for BLOCKS PER GRID for different lengths of waveforms.

As for the performance of different generations of GPU platforms, comparing results in Figs. 8–10, TITAN Xp (Pascal) consistently outperforms TITAN Z (Kepler) in this test. In general, TITAN Xp runs twice as fast as TITAN Z regardless of the size of the dataset.

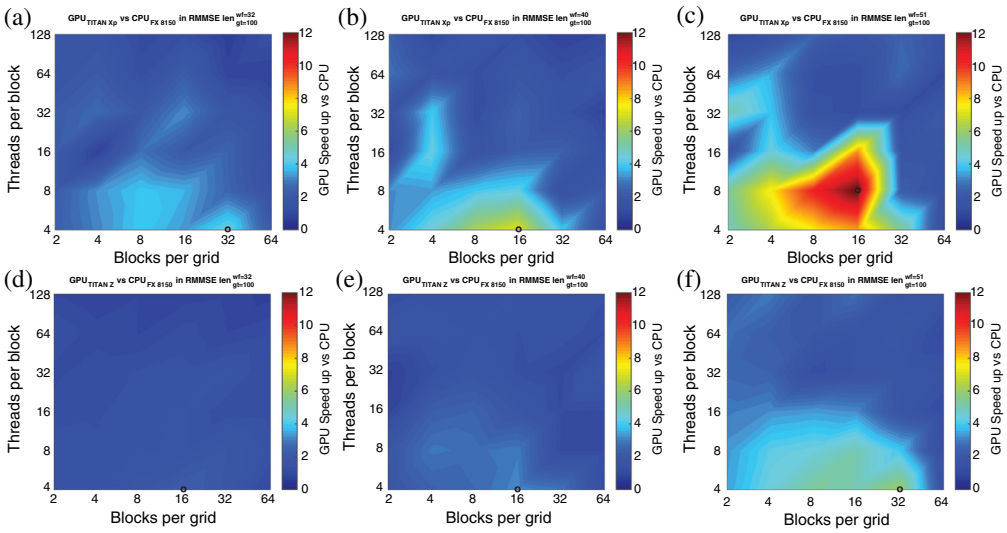


**Fig. 10** The performance comparison of RMMSE between GPU and CPU implementations with various CUDA configurations and length of waveforms when length of ground truth is fixed to 100 sample points. Lengths of waveforms are from 32 to 51. The black circle represents the configuration when maximum speed up is achieved under current data format. (a)–(c) TITAN Xp, len_wf = 32, 40, 51 and (d)–(f) TITAN Z, len_wf = 32, 40, 51.

## 5.3 *MF-RMMSE*

The performance comparison of MF-RMMSE implementations based on GPU and CPU platforms is demonstrated in Fig. 11. As can be seen, the acceleration is not as significant as the implementation of RMMSE, which is shown in Fig. 8. As described in Sec. 2.3, one reason for this phenomenon is the way MF-RMMSE is designed, which is different from that of RMMSE.[10] It utilizes a processing window that is much smaller than the length of the waveform, and it effectively reduces the impact of the length of the waveform on processing time. In addition, as mentioned in Sec. 5.2 and implied in Table 2, the advantage on performance of a GPU over a CPU is more significant when the length of the waveform is large. Thus, GPU implementations may be less beneficial for MF-RMMSE than the original RMMSE. However, similar to the observation of the performance result of RMMSE, GPU implementation of the MF-RMMSE algorithm performs better when the length of the waveform grows longer.

Figure 12 shows another view on the performance comparison of MF-RMMSE. The white dashed line in Fig. 12 represents the equal performance between the CPU and GPU implementations. It shows that a 2.5 times speed up is achievable in several configurations of the source

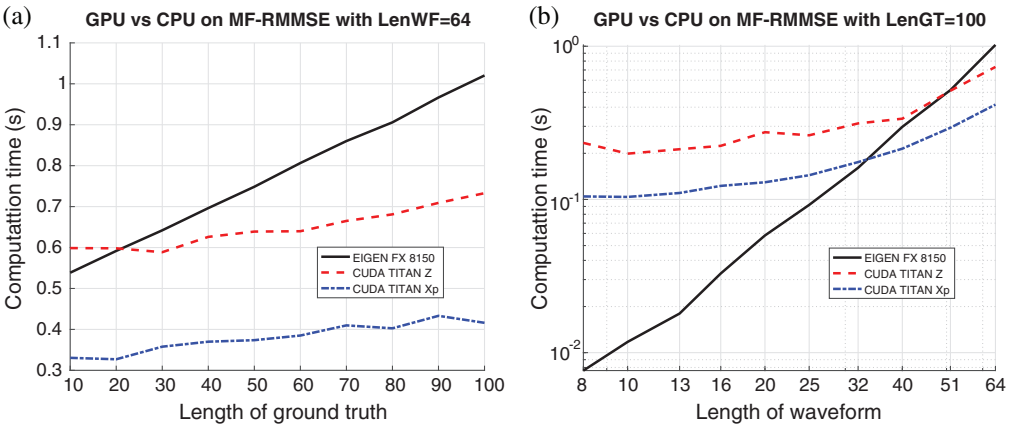


**Fig. 11** The performance comparison of MF-RMMSE based on GPU and CPU platforms. Length of (a) waveform = 51 and (b) ground truth = 100.

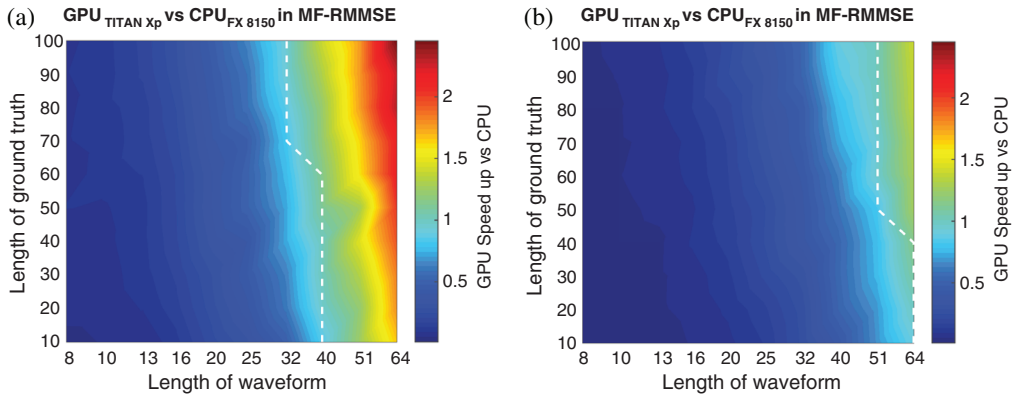


**Fig. 12** The performance comparison of MF-RMMSE based on GPU and CPU platforms. (a) TITAN Xp and (b) TITAN Z.

profiles. Although the MF-RMMSE algorithm utilizes MF outputs,[10] the computation cost of an MF is negligible compared to MF-RMMSE. Thus, compared with Fig. 7, the acceleration curve in Fig. 12 is much smoother.

Similar to Fig. 10, Fig. 13 summarizes the impacts of the GPU kernel resources on the acceleration performance of MF-RMMSE implementations. The black circles in each figure represent the largest achievable speed up of the GPU compared with the CPU counterpart having specific lengths of waveforms, whereas the length of the ground truth is fixed at 100 sample points for all the cases, since it has less impact on the processing time as discussed earlier.

It can be seen from Fig. 13 that an optimized computing resource configuration may be achieved for each specific data size. For the application of MF-RMMSE investigated in this study, we can conclude that the best configurations of CUDA are 4 for THREADS PER BLOCK and 64 or 128 for BLOCKS PER GRID for different lengths of waveforms. The trend is similar to what could be found in the testing results of RMMSE in Sec. 5.2.

As for the performance of different generations of GPU platforms, comparing results in Figs. 11–13, TITAN Xp (Pascal) consistently outperforms TITAN Z (Kepler) in this test. For general comparison, algorithms implemented on TITAN Xp execute twice as fast as TITAN Z regardless of the sizes of the datasets.

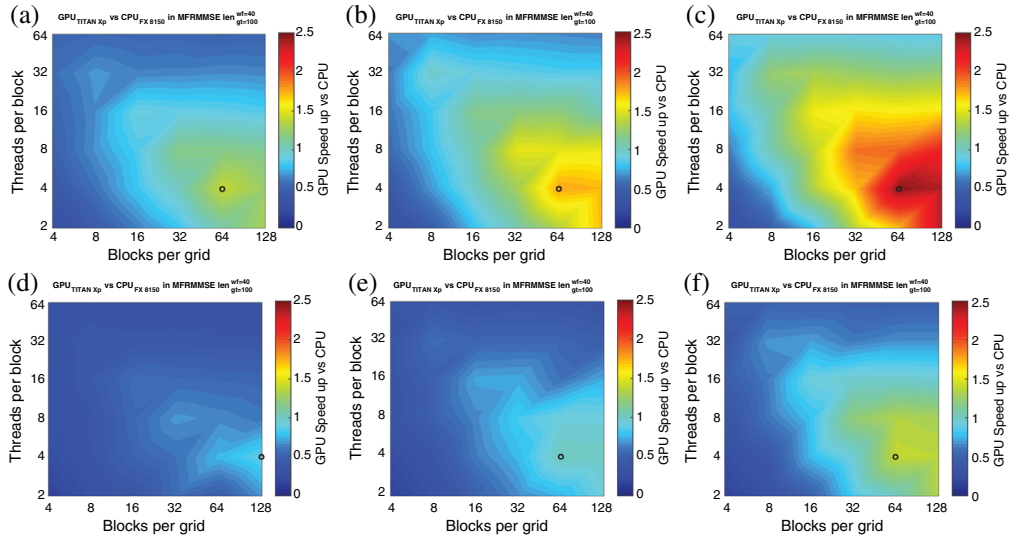


**Fig. 13** The performance comparison of MF-RMMSE between GPU and CPU implementations with various CUDA configurations and length of waveforms when length of ground truth is fixed to 100 sample points. Lengths of waveforms are from 40 to 64. The black circle represents the configuration when maximum speed up is achieved under current data format. (a)–(c) TITAN Xp, len_wf = 40, 51, 64 and (d)–(f) TITAN Z, len_wf = 40, 51, 64.

## 6 Application to Real/Measured Data

MF is implemented on NASA's HIWRAP radar measurement data[10] with both the CPU-based method (EIGEN) and GPU-based method (CUDA). From the MF outputs depicted in Fig. 14, the two results are almost identical except for those positions (range points from 400 to 570) with low-signal power levels. Figure 15 shows averaged MF outputs of EIGEN and CUDA among 10,000 range profiles, respectively. From these two figures, it is clearer that the only noticeable difference between EIGEN and CUDA results is located within the low-signal return region. Figure 16 shows the absolute and relative errors of CUDA output compared with EIGEN output. The errors are calculated in linear scale not dB. Although there is no way to determine whether the EIGEN or CUDA output is more "correct," the two outputs are similar to each other, thus they are able to validate each other. From Fig. 16(a), it could be observed that the errors are consistent in the same level. However, Fig. 16(b) shows that the errors are more noticeable when the signal-to-noise ratio is lower. Magnitude-square coherence between CUDA and EIGEN outputs is also demonstrated in Fig. 17. From the coherence results in those figures, it can be concluded that the results from CUDA and EIGEN outputs are identical.

For the computation time evaluation, since the MF is a relatively simple algorithm, it is expected that the data transferring process between the CPU and GPU would be the bottleneck for such an application as the result demonstrated in Fig. 18. In this experiment, to fully utilize the parallel computing ability of the GPU, data transfer from host (CPU) memory to device (GPU) memory is done by a burst of all the data included (to be specific, copy all data

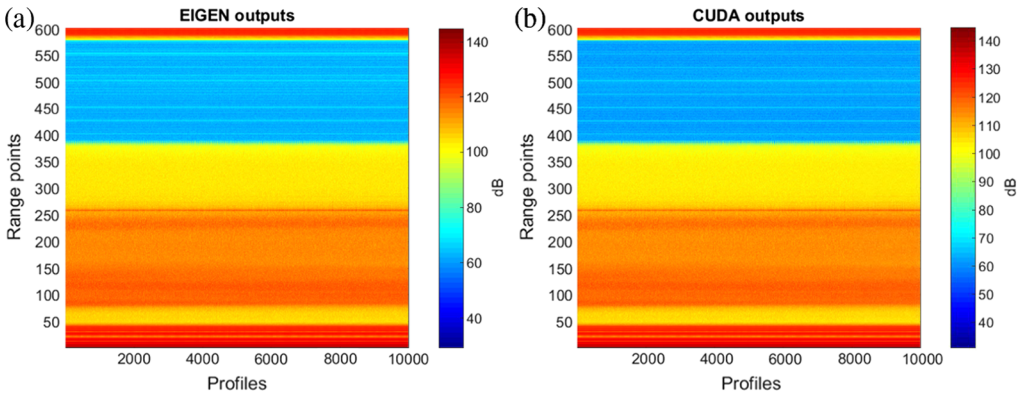


**Fig. 14** Output results of pulse compression implementation using NASA's HIWRAP radar data. MF output of (a) EIGEN and (b) CUDA.

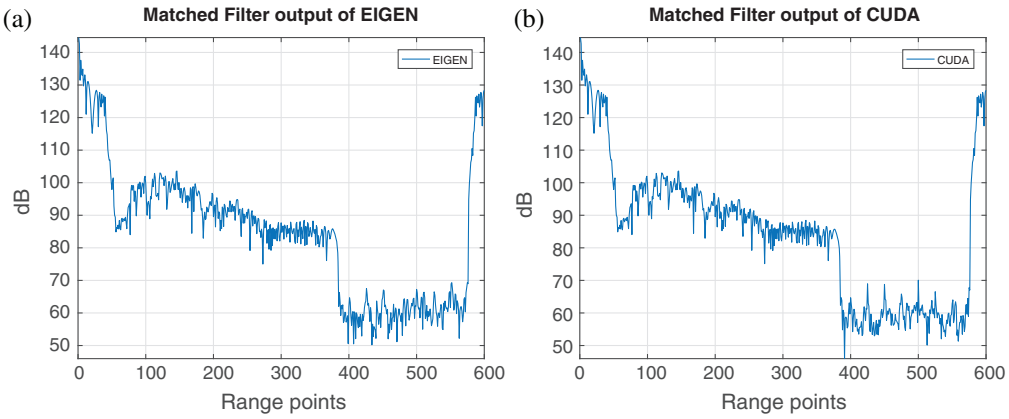


**Fig. 15** Averaged output results of pulse compression implementation using NASA's HIWRAP radar data. Averaged MF output of (a) EIGEN and (b) CUDA.

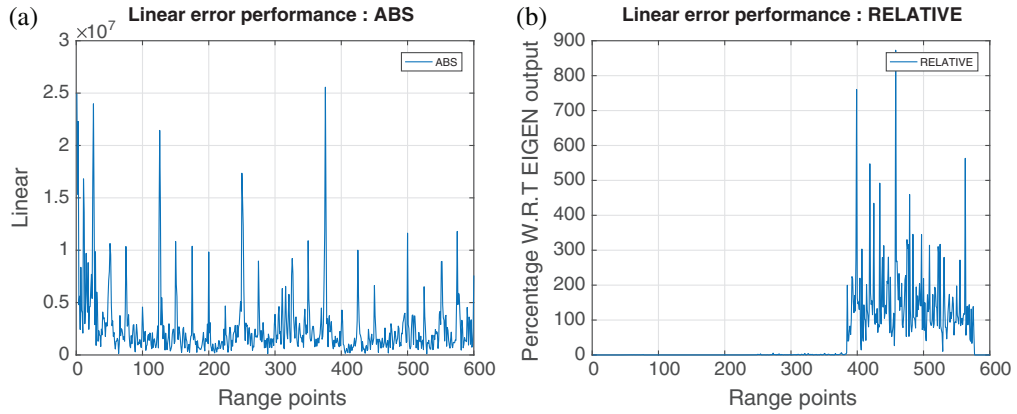


**Fig. 16** Error results of pulse compression implementation using NASA's HIWRAP radar data. (a) Absolute linear error performance and (b) relative linear error performance.

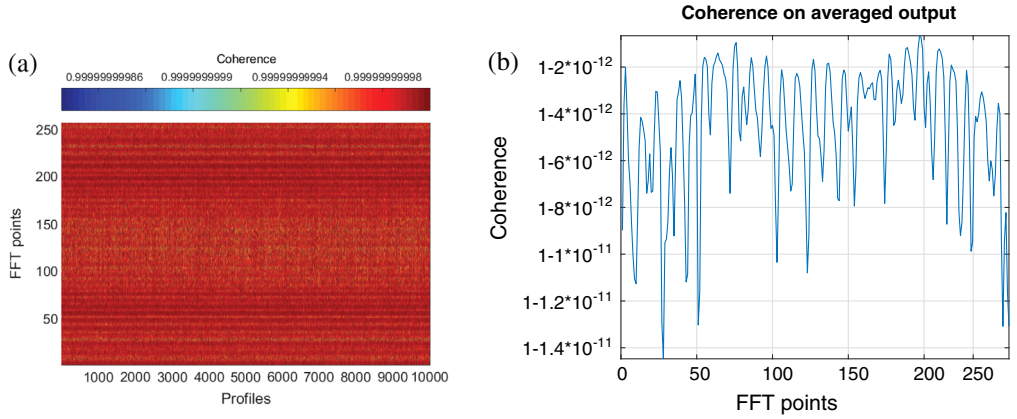


**Fig. 17** Coherence results of pulse compression implementation using NASA's HIWRAP radar data. Magnitude-square coherence estimate between (a) EIGEN and CUDA outputs and (b) averaged EIGEN and CUDA outputs.

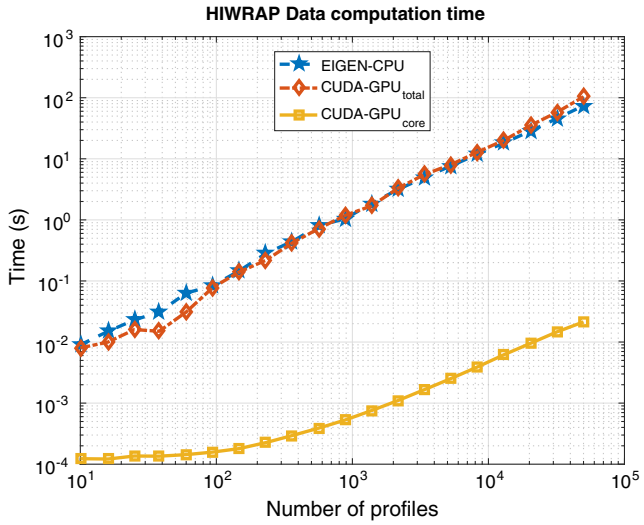


**Fig. 18** Time consumption of CPU and GPU versus number of range profiles handling in HIWRAP data, whereas GPU time consumption without data transfer is demonstrated separately.

from "THRUST host vector" to "THRUST device vector" in one command). It could be observed that, taking account of the overhead of data transfer between the CPU and GPU, the time consumption of the GPU-based approach is nearly comparable to the CPU-based approach. However, if the time of the GPU computing part is recorded separately from the data transferring part, the "true" GPU time consumption is far less than the CPU counterpart. To implement this evaluation specifically for the GPU computing part, a timer based on CUDA events is used around the GPU computing part as recommended by an official document from NVIDIA.[16] In this experiment, where only MF is implemented, the time consumption of the CPU-based approach is as much as 3466 times larger than the "true" GPU-based approach computing time consumption. Here, in this experience, AMD FX-8150 and NVIDIA TITAN Z are used as CPU and GPU devices, respectively.

## 7 Summary and Conclusions

In this study, we explored the feasibility of GPGPU-based implementation of advanced pulse compression algorithms, which is a key element and usually a bottleneck of an end-to-end radar data processing chain. For APC, GPGPU-based solutions show great potentials of accelerating such resource-demanding algorithms without introducing additional processing error. During the investigation of the accelerating ability of the GPGPU, a correlation is discovered between the computing resource distribution on GPU hardware and acceleration performance compared with CPU-based platforms under different sizes and structures of data being handled. For future reference, an optimal way is proposed to allocate resource on GPUs for better performance based on empirical data. In addition, as a major obstacle of various GPGPU implementations, the overhead of the data transfer between CPU (host) and GPU (device) might be enormous for some data heavy applications. One instance of such data-intensive applications is analyzed, and the result indicates that a more sophisticated and efficient memory management scheme is required to unleash the parallel computing capability of GPU in such applications. As the GPGPU platform is also available in an embedded system, for future experiments, the revised APC algorithms could be also implemented on a system-on-chip (SoC) platform "Tegra X2 (Parker)," which is a lightweight computing solution distributed by NVIDIA with its peak computation power at over 1500 GFLOPS per unit. In addition, the thermal design power (TDP) of this SoC is just 15 watts and it is able to be expanded to clusters for more computing power and to meet the low space, weight, and power requirement of many applications. Compared with the TITAN Xp we used in our experiment, which comes with 12150 GFLOPS, X2 provides 1/8 of the computation power of its desktop counterpart. While comparing with another popular approach for pulse compression which is a DSP-focused, high-performance, embedded computing platform,[20] the potential computation ability of the GPGPU approach [1500 GFLOPS-Tegra X2 (Parker)] is significantly higher than it of the DSP approach (160 GFLOPS-TI C6678). However, with fewer overheads, the DSP-based approach could be more efficiently used than the GPGPU counterpart, which might mitigate the performance gap between GPGPU and DSP (1500 GFLOPS versus 160 GFLOPS). Nevertheless, the ease of transforming existing codes into GPGPU enabled codes, compared with DSP approach which requires heavy modification of existing CPU-based codes, would still make the GPGPU a better high-performance solution for existing algorithms.

## Disclosures

Mr. Cai has nothing to disclose.

## References

1. T. Balz and U. Stilla, "Hybrid GPU-based single-and double-bounce SAR simulation," *IEEE Trans. Geosci. Remote Sens.* **47**(10), 3519–3529 (2009).
2. F. Zhang et al., "Accelerating spaceborne SAR imaging using multiple CPU/GPU deep collaborative computing," *Sensors* **16**(4), 494 (2016).

3. W. Xue and Y. Jiang, "Simulation of fast threshold CFAR processing algorithm with CUDA," in *Int. Conf. on Computer Distributed Control and Intelligent Environmental Monitoring*, pp. 621–624, IEEE, Changsha, Hunan, China (2012).
4. C. Venter, H. Grobler, and K. AlMalki, "Implementation of the CA-CFAR algorithm for pulsed-Doppler radar on a GPU architecture," in *IEEE Jordan Conf. on Applied Electrical Engineering and Computing Technologies*, pp. 1–6, IEEE, Amman, Jordan (2011).
5. J. Cai and Y. Zhang, "Acceleration of generalized adaptive pulse compression with parallel GPUs," *Proc. SPIE* **9461**, 946107 (2015).
6. J. Cai et al., "Acceleration of advanced radar processing chain and adaptive pulse compression using GPGPU," in *Proc. of the 24th High Performance Computing Symp.*, p. 7, Society for Computer Simulation International, Pasadena, California (2016).
7. C. Fallen et al., "GPU performance comparison for accelerated radar data processing," in *Symp. on Application Accelerators in High-Performance Computing*, pp. 84–92, IEEE, Knoxville, Tennessee (2011).
8. S. D. Blunt and K. Gerlach, "Adaptive pulse compression via MMSE estimation," *IEEE Trans. Aerosp. Electron. Syst.* **42**(2), 572–584 (2006).
9. Z. Li et al., "Super-resolution processing for multi-functional LPI waveforms," *Proc. SPIE* **9077**, 90770M (2014).
10. Z. Li et al., "Fast adaptive pulse compression based on matched filter outputs," *IEEE Trans. Aerosp. Electron. Syst.* **51**(1), 548–564 (2015).
11. L. Li et al., "High-altitude imaging wind and rain airborne radar (HIWRAP)," in *IEEE Int. Geoscience and Remote Sensing Symp.*, Vol. **3**, p. 354, IEEE, Boston, Massachusetts (2008).
12. F. Kong et al., "Real-time radar signal processing using GPGPU (general-purpose graphic processing unit)," *Proc. SPIE* **9829**, 982914 (2016).
13. Nvidia Corporation, "CUDA C programming guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (31 May 2017).
14. M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. IEEE* **93**(2), 216–231 (2005) Special issue on "Program Generation, Optimization, and Platform Adaptation."
15. G. Guennebaud et al., "EIGEN v3," 2010, http://eigen.tuxfamily.org (31 May 2017).
16. Nvidia Corporation, "CUDA C best practices guide," http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html (22 November 2016).
17. L. Ma, R. D. Chamberlain, and K. Agrawal, "Analysis of classic algorithms on GPUs," in *Int. Conf. on High Performance Computing & Simulation (HPCS '14)*, pp. 65–73, IEEE (2014).
18. L. Ma, "Modeling algorithm performance on highly-threaded many-core architectures," PhD Thesis, Washington University, St. Louis (2014).
19. M. Wolfe, "Understanding the CUDA Data Parallel Threading Model a Primer," PGI Insider, 2010, https://www.pgroup.com/lit/articles/insider/v2n1a5.html (31 May 2017).
20. X. Yu et al., "An implementation of real-time phased array radar fundamental functions on a DSP-focused, high-performance, embedded computing platform," *Aerospace* **3**(3), 28 (2016).

**Jingxiao Cai** received his bachelor's degree in 2012 from Sichuan University and his master's degree at the University of Oklahoma in 2017. He is a PhD candidate in electrical engineering at the University of Oklahoma. His experiences include weather radar meteorology, adaptive pulse compression, HPC on GPGPU, and machine learning. His work focuses on the conjunctions of those fields. Currently, he is working on applying machine intelligence to turbulence detection.

**Yan Zhang** is current associate professor and presidential professor of the School of Electrical and Computer Engineering, University of Oklahoma. He is the faculty leader of the Intelligent Aerospace Radar Team (IART) and one of the founding faculty members of the Advanced Radar Research Center (ARRC).