IEICE TRANSACTIONS

on Fundamentals of Electronics, Communications and Computer Sciences

DOI:10.1587/transfun.2023EAP1137

Publicized:2024/07/23



A PUBLICATION OF THE ENGINEERING SCIENCES SOCIETY

The Institute of Electronics, Information and Communication Engineers

Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER

# Real-time Implementation of Joint Domain Localised Algorithm for High Frequency Surface Wave Radar using GPU

**Bowen ZHANG[†], Chang ZHANG[††], Non*members*, *Di YAO*[†††], M*ember*, and Xin ZHANG[†*a)], Non*member***

**SUMMARY** The performance of target detection and tracking is primarily limited by ionospheric interference in High Frequency Surface Wave Radar (HFSWR). Joint Domain Localised (JDL) has been proved to be an effective algorithm for ionospheric clutter suppression in HFSWR. However, the implementation of JDL in the traditional CPU platform cannot afford the real-time requirement in HFSWR. With the help of the tremendous parallel computational horsepower in GPU, in this paper we investigate the real-time implementation of JDL algorithm for HFSWR using Graphics Processing Unit (GPU). We also perform a comparative analysis in terms of the performance using the CPU-based implementation and the GPU-based implementation. Experimental result shows that the GPU-based implementation accelerates the computation by over 24.72 times as compared to the CPU-based implementation which meets the real-time requirement of HFSWR.

***key words:*** *HFSWR, ionospheric clutter, JDL, GPU*

## 1. Introduction

Generally, High Frequency Surface Wave Radar (HFSWR) transmits the high frequency band (3-30MHz) electromagnetic energy into a specific volume in space to search for targets, such as ships and aircrafts [1]-[3]. The target echoes data are then transmitted to the signal processing subsystem to extract target information such as range, velocity, angular position, and other target identifying characteristics [4]. Usually, the performance of target detection and tracking in HFSWR is primarily limited by ionospheric interference [5]-[7]. Ionospheric interference is characterized by a high degree of nonhomogeneity and nonstationarity, which makes its suppression difficult using conventional processing techniques.

Space-time Adaptive Processing (STAP) has been proved to be an efficient adaptive clutter suppressed algorithm which has enjoyed great success in HFSWR [8], [9]. Real-time implementation of STAP is considered impossible, as the computational cost of inverting a covariance matrix is considered too expensive [9]. Joint Domain Localised (JDL) algorithm which is a kind of partial STAP algorithm has been proved to be an effective clutter suppressed algorithm in HFSWR especially for ionospheric interference [10], [11]. Comparing with STAP, JDL can reduce the computing load quite a lot by transforming the huge channel-pulse dimension to the concerned angle-Doppler frequency domain [12]. However, HFSWR is a long distance detecting and high Doppler frequency resolution system which means the concerned region contains hundreds to thousands Doppler frequency units and hundreds range units. Combined with tens angle units, a huge three-dimension cube is organized. For each unit in this huge cube, JDL have to execute one time. In this case, the computing load for executing the whole cube for JDL is still quite heavy.

Originally, the Graphics Processing Unit (GPU) is a specialized circuit designed to accelerate computation for building and manipulating images. With CUDA which is a parallel computing and programming model developed by NVIDIA in 2006 [13], developers were able to dramatically speed up computing applications by harnessing the power of GPU. Hence, GPU has become a significant compute engine in the general computing, such as signal processing [14]-[16], computer vision and pattern recognition [17]-[19], machine learning [20]-[22] and data simulation [23]-[25]. A post-Doppler STAP extended factored algorithm (EFA) with a block training approach to estimate the required covariance matrices and through solving the linear system to achieve near-peak utilization using the compute unified device architecture (CUDA) framework GPU based implementation provided by NVIDIA [26].

In this paper, we investigate the real-time implementation of JDL algorithm for HFSWR. Owing to the implementation of JDL in the traditional CPU platform cannot afford the real-time requirement in HFSWR. Therefore, our team with the help of the tremendous parallel computational horsepower in GPU, we investigate the real-time implementation of JDL algorithm for HFSWR using GPU as specific novelty of the research. The remainder of this paper is organized as follows. Section 2 presents the JDL algorithm for HFSWR. Section 3 describes our real-time implementation of JDL algorithm using GPU in detail. Section 4 provides experimental results of our approach and evaluates its effectiveness. Finally, we summarize our work and make conclusion in Section 5. The JDL algorithm for HFSWR using GPU was completed together by Bowen ZHANG and Chang ZHANG. Di YAO completed the implementation and analysis of the CPU based JDL algorithm, while Xin ZHANG complete the algorithm experimental results. This paper was jointly written by the

†The authors are Dept. of Electronic and Information Engineering, Harbin Institute of Technology, Harbin, Heilongjiang 150001, China.
††The author is with Jiangsu Automation Research Institute (JARI), Jiangsu, 222000, China.
†††The School of Computer Science and Engineering, Northeastern University, Shenyang 110819, China.
*Songjiang Laboratory, Harbin Institute of Technology, Harbin 150001, China.
a)zhangxinhit@hit.edu.cn

four individuals mentioned above.

## 2. Joint Domain Localised (JDL) for HFSWR

### 2.1 Space-time data model for HFSWR

For HFSWR, the receiving array comprised of N isotropic, point sensors separated by a distance of $d$, receiving an incident plane wave, as shown in Fig. 1.

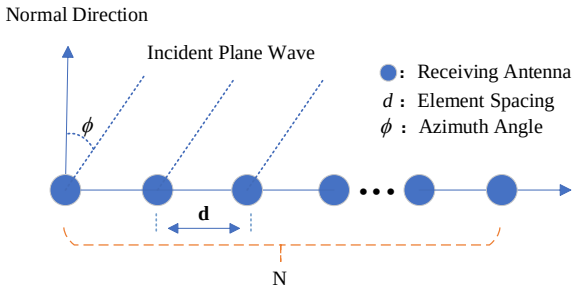



**Fig. 1** Receiving array antenna of HFSWR

The reflected target echoes are collected and sampled at N array channels and M successive pulses in a coherent processing interval (CPI) for each range unit. For a given l-th range bin, the space–time snapshot vector is defined

$$\boldsymbol{X}_l = \left[\boldsymbol{x}_1 \quad \boldsymbol{x}_2 \quad \cdots \quad \boldsymbol{x}_M\right]^T \in C^{MN\times 1} \tag{1}$$

where $\boldsymbol{x}_i \in C^{N\times 1}$ is used to denote the spatial snapshot of data corresponding to the $i$-th pulse repetition interval (PRI).

By dividing each pulse repetition interval (PRI) into $K$ separate range bins, the corresponding space-time snapshots are compiled into one $N\times M\times K$ data-cube, as shown in Fig. 2.

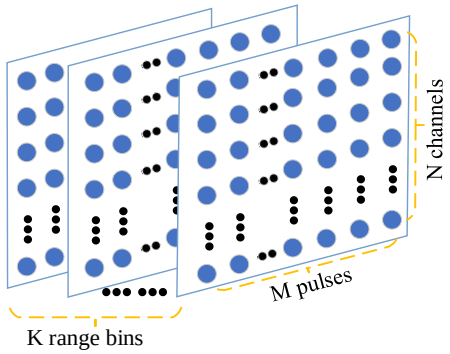



**Fig. 2** A 3-dimensional representation of a data-cube in JDL

The space–time snapshot vector $\boldsymbol{X}_l$ is used to construct the estimated covariance matrix $\boldsymbol{R}$

$$\boldsymbol{R} = \boldsymbol{X}_l \boldsymbol{X}_l^H \in C^{MN\times MN} \tag{2}$$

and the weight $\boldsymbol{w}$ for STAP is given as.

$$\boldsymbol{w} = \boldsymbol{R}^{-1}\boldsymbol{v} \tag{3}$$

where $\boldsymbol{v} \in C^{MN\times 1}$ is the space-time steering vector corresponding to a target at azimuth angle $\phi_t$ and Doppler frequency $f_t$. This space-time steering can be written as follows.

$$\boldsymbol{v} = \boldsymbol{b}(f_t) \otimes \boldsymbol{a}(\phi_t) \tag{4}$$

where $\otimes$ represents the Kronecker product of two vectors, $\boldsymbol{a}(\phi_t)$ is a space steering vector defined by

$$\boldsymbol{a}(\phi_t) = [1 \; e^{j2\pi\frac{d}{\lambda}\sin(\phi_t)} \; e^{j(2)2\pi\frac{d}{\lambda}\sin(\phi_t)} \; \cdots \; e^{j(N-1)2\pi\frac{d}{\lambda}\sin(\phi_t)}]^T \tag{5}$$

and $\boldsymbol{b}(f_t)$ is a time steering vector defined by

$$\boldsymbol{b}(f_t) = [1 \; e^{j2\pi f_t/f_R} \; e^{j(2)2\pi f_t/f_R} \; \cdots \; e^{j(M-1)2\pi f_t/f_R}]^T \tag{6}$$

where $\lambda$ is wavelength and $f_R$ is pulse repetition frequency (PRF).

Real-time Implementation of STAP is considered impossible, as the computation cost of inverting a $MN\times MN$ dimensional matrix, ($O(MN^3)$), is considered too expensive for large values of $M$ and $N$.

### 2.2 Joint Domain Localised (JDL)

JDL algorithm was first introduced by Wang and Cai[27]. By using a transformation matrix $\boldsymbol{T}$, it can transform the space-time signal vector $\boldsymbol{X}_l$ from channel-pulse domain data to the partial angle-Doppler frequency domain data $\tilde{\boldsymbol{X}}_l$, which is as follows

$$\tilde{\boldsymbol{X}}_l = \boldsymbol{T}^H \cdot \boldsymbol{X}_l \tag{7}$$

and the corresponding transformed space-time steering vector can be written as.

$$\tilde{\boldsymbol{v}} = \boldsymbol{T}^H \cdot \boldsymbol{v} \tag{8}$$

Usually we called the partial angle-Doppler frequency domain as localised processing region (LPR). Adaptive processing is restricted to the LRP. Fig. 3 shows a LPR contains $\eta_a = 3$ angle units and $\eta_d = 3$ Doppler units which with the center of Doppler frequency $f_0$, angle $\varphi_0$ and range gate $k$. So the transformation matrix $\boldsymbol{T}$ can be written as

$$\boldsymbol{T} = [\boldsymbol{b}(f_{-1}), \boldsymbol{b}(f_0), \boldsymbol{b}(f_1)] \otimes \left[\boldsymbol{a}(\varphi_{-1}), \boldsymbol{a}(\varphi_0), \boldsymbol{a}(\varphi_1)\right] \tag{9}$$

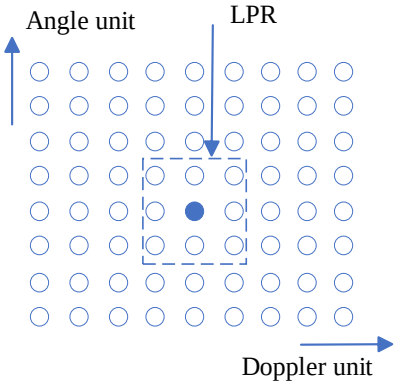



**Fig. 3** An example of localised processing region

and the covariance matrix $\tilde{\boldsymbol{R}}$ in JDL is defined as

$$\tilde{\boldsymbol{R}} = \frac{1}{P}\sum_{0}^{P-1} \tilde{\boldsymbol{X}}_l \tilde{\boldsymbol{X}}_l^H \in C^{\eta_a\eta_d \times \eta_a\eta_d} \quad (10)$$

where $P$ is the number of weight training data samples employed. To ensure the expectation of the ratio of the adaptive SNIR to optimum SNIR is greater than 0.5, $P$ should satisfy $P \geq 2\eta_a\eta_d$. And $l$=$k$±2, $k$±3,…, $k$±($P$/2+1) with one range protected unit[9, 12]. Forming LPR significantly reduced the number of unknowns while remaining maximal gain against noise, that is from $\boldsymbol{R} \in C^{MN\times MN}$ to $\tilde{\boldsymbol{R}} \in C^{\eta_a\eta_d\times\eta_a\eta_d}$. The lower degrees of freedom lead to a corresponding reduction in required sample support as well as computational cost.
So the optimal weights can be expressed as

$$\tilde{\boldsymbol{w}} = \tilde{\boldsymbol{R}}^{-1}\tilde{\boldsymbol{v}} \quad (11)$$

and the result of JDL algorithm is as follows

$$\boldsymbol{y} = \tilde{\boldsymbol{w}}^H \tilde{\boldsymbol{X}}_l \quad (12)$$

## 3. JDL Algorithm parallel Implementation

### 3.1 The Math Kernel Library of CPU

The implementation process of JDL algorithm on CPU and GPU is the same, but the difference is that GPU has stronger computing power than CPU, and using GPU can greatly improve the efficiency of engineering implementation. As CPU does not provide the ability to make the cells calculating parallelly. We had to loop the processing to process the whole three-dimensional data-cube. By simulating on the CPU and implementing the JDL algorithm with the same batch of echo data as the GPU. And calculate the average time cost of implementing the JDL algorithm on the CPU.

The library functions that implement the JDL algorithm have been widely used on CPU. The Intel® Math Kernel Library [29] includes the Basic Linear Algebra Subprograms (BLAS) routines that provide standard building blocks for performing basic vector and matrix operations. The Level 1 BLAS perform scalar, vector and vector-vector operations, the Level 2 BLAS perform matrix-vector operations, and the Level 3 BLAS perform matrix-matrix operations. By applying the matrix operation functions involved in this library, the JDL algorithm can be implemented on CPU, such as *cblas_cgemm()* perform matrix-matrix multiplication operations, *cblas_cgemv()* perform general matrix-vector multiplication operations, *cblas_chemv()* perform the Hermitian matrix-vector multiplication operations, etc.

### 3.2 NVIDIA CUDA Programming Model

CUDA is a general-purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs [13]. Usually, a modern NVIDIA GPU hardware consists of thousands of elementary processing units, called CUDA cores, divided in blocks called Streaming Multiprocessors (SM). Each SM has a constant memory and a shared memory, which have much lower latency than the device global memory, as shown in Fig. 4.

In CUDA programming, the function that can be executed in CUDA threads is called kernel. A kernel is

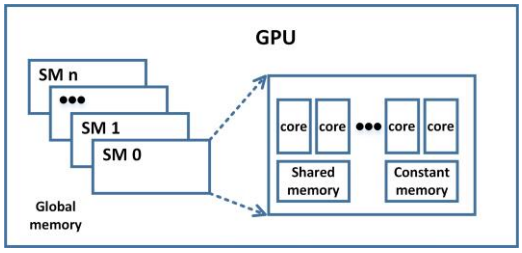



**Fig. 4** The composition of GPU

defined using the *__global__* declaration specifier. A kernel can be executed by multiple equally-shaped thread blocks. There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same SM and must share the limited memory resources of that SM. In principle, blocks are organized into a one-dimension, two-dimensional, or three-dimensional grid of thread blocks as illustrated by Fig. 5 Each block within the grid can also be identified by a one-dimensional, two-dimensional, or three-dimensional grid of threads. The number of the thread blocks in a grid is usually dictated by the size of the data being processed. The core of CUDA are three key abstractions, that is a hierarchy of thread groups, shared memories and barrier synchronization.

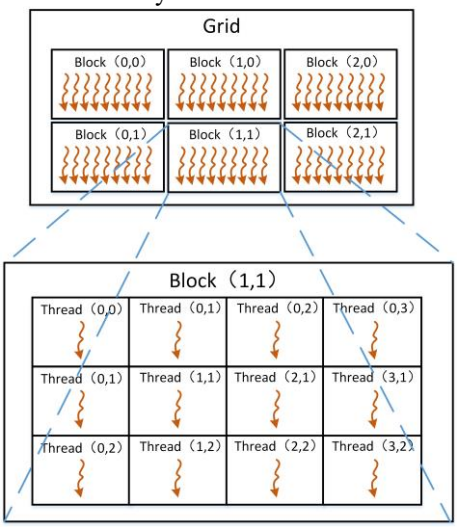



**Fig. 5** Grid of thread blocks in CUDA

### 3.3 Implementation

As the limited global memory resources of one GPU card, assuming a $N$×$M$×$K$ data-cube can be executed in one GPU card, where $N$ is the number of angle units, $M$ is the number of Doppler frequency units and $K$ is the number of range units. The transformed space-time signal vector *Host_X* is formed by $N$×$M$×$K$ cells. As the LPR is formed by 3 angle

units and 3 Doppler units, in each cell, the input data is formed by the extraction of adjacent Doppler frequency units and adjacent angle units which formed a vector with the size $9\times1$ in Eq. (7). The transformed steering vector *Host_V* should also be prepared by Eq. (8).

Our scheme for implementing the JDL algorithm takes the following steps in the code, as shown in Fig. 6.

```
Input:  Host_X,  Host_V                    Output:  Host_Y
Algorithm
1. cudaMemcpy (Device_X, Host_X, cudaMemcpyHostToDevice) ;
2. cudaMemcpy (Device_V, Host_V, cudaMemcpyHostToDevice) ;
Step 1: Calculate the covariance matrix :  R̃_i = X̃_l X̃_l^H
3. matrixMulXxhBatch<<<grid_xxh, thrd_xxh>>>(Device_C, Device_X) ;
4. cudaDeviceSynchronize () ;
Step 2: Sum the covariance matrix :  R̃ = (1/P) Σ_{i=0}^{P-1} R̃_i, P ≥ 2η_a η_d
5. sumCovarianceMatrix<<<grid_sum, thrd_sum>>>(Device_R, Device_C) ;
6. cudaDeviceSynchronize () ;
Step 3: Calculate the inverse matrix of R̃
7. cublas<t>getrfBatched (handle, 9, Device_R, 9, Pivot, info, batch) ;
8. cublas<t>getriBatched (handle, 9, Device_R, 9, Pivot, Device_InvR, 9, info, batch) ;
9. cudaDeviceSynchronize () ;
Step 4: Calculate the matrix :  w̃ = R̃^{-1} ṽ
10. matrixMulRvBatch<<<grid_Rv, thrd_Rv>>>(Device_W, Device_InvR, Device_V) ;
11. cudaDeviceSynchronize () ;
Step 5: Calculate the final result :  y = w̃^H X̃_l
12. matrixMulWxBatch<<<grid_Wx, thrd_Wx>>>(Device_Y, Device_W, Device_X) ;
13. cudaDeviceSynchronize () ;
14. cudaMemcpy (Host_Y, Device_Y, cudaMemcpyDeviceToHost) ;
```

**Fig. 6** Parallel implementation of JDL algorithm in GPU

By transmitting *Host_X* and *Host_V* from the CPU memory to the GPU global memory (line 1, 2), the implementation begin. For step 1, a kernel function *__global__ void matrixMulXxhBatch()* (line 3) is used to calculate $N\times M\times K$ cells of matrix multiplication $\tilde{\boldsymbol{R}}_i=\tilde{\boldsymbol{X}}_l\tilde{\boldsymbol{X}}_l^H$. The grid of thread blocks for Step 1 is shown in Fig. 8. That is a two-dimensional grid of threads in each block is identified by *dim3 thrd_xxh(9,90)* and a one-dimensional grid of blocks in this grid is identified by *dim3 grid_xxh(Q, 1),* where $Q=\frac{N\times M\times K}{10}$. As shown in Fig. 7, in each block, we divide the threads grid into 10 regions, each region is formed by 81 threads with the size of $9\times9$ which maps to one cell of $\tilde{\boldsymbol{R}}_i$.

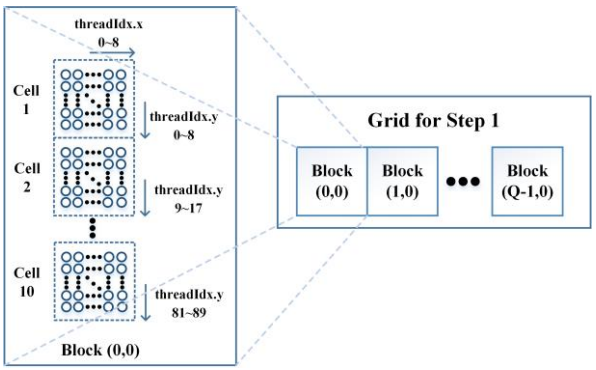



**Fig. 7** Grid of thread blocks in Step 1

For step 2, a kernel function *__global__ void sumCovarianceMatrix() (*line 6) is used to calculate Eq. (10), as $\eta_a=\eta_d=3$, in order to satisfy (10), we choose $P=18$. The grid of thread blocks in step 2 is as follows, *dim3 thrd_sum(81,10)* and *dim3 grid_sum (*$\frac{M\times K}{10}$*, N)*, shown in Fig. 8. In each block, we divide the threads grid into 10 regions, each region is formed by 81 threads with the size of $81\times1$ to calculate one cell of $\tilde{\boldsymbol{R}}$. In this grid, we divide the blocks into N regions, each region calculate the $M\times K$ cells of $\tilde{\boldsymbol{R}}$ in one angle unit.

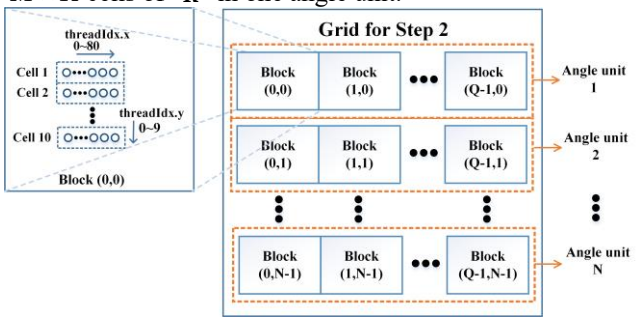



**Fig. 8** Grid of thread blocks in Step 2

For step 3, to calculate the inverse matrix of $\tilde{\boldsymbol{R}}$, we use cuBLAS API library [28] which is an implementation of Basic Linear Algebra Subprograms (BLAS) on top of the CUDA runtime. The *cublas<t>getriBatched()* function (line 8) performs the inversion of matrix $\tilde{\boldsymbol{R}}_i$ for i=0, … ,batchSize-1. Prior to calling *cublas<t>getriBatched()*, the matrix $\tilde{\boldsymbol{R}}_i$ must be factorized first by using the routine *cublas<t>getrfBatched()* function (line 7). The *cublas<t>getrfBatched()* function performs the LU factorization of each $\tilde{\boldsymbol{R}}_i$ for i=0, … ,batchSize-1. Following the LU factorization, *cublas<t>getriBatched()* function uses forward and backward triangular solvers to complete inversion of matrix $\tilde{\boldsymbol{R}}_i$ for i=0, … ,batchSize-1.

For step 4, a kernel function *__global__ void sumCovarianceMatrix()* (line 10) is used to calculate the $N\times M\times K$ cells of the optimal weights $\tilde{\boldsymbol{w}}$. The grid of thread blocks in step 4 is as follows, *dim3 thrd_Rv(9,100)* and *dim3 grid_sum(*$\frac{M\times K\times N}{100}$*, 1).*

For step 5, a kernel function *__global__ void matrixMulWxBatch()* (line 12) is used to calculate the result of JDL algorithm. The grid of thread blocks in Step 5 is as follows, *dim3 thrd_Wx(1024,1)* and *dim3 grid_Wx(*$\frac{M\times K\times N}{1024}$*, 1).* By transmitting the result *Device_Y* from the GPU global memory to the CPU memory *Host_Y*, the implementation of JDL on GPU is accomplished.

For the implementation of JDL on GPU, there are three key points as follows:

1) As the threads in kernel function are not synchronized while they are running, the *cudaDeviceSynchronize()* function should be used at the end

of each kernel function for data synchronization.

2) Shared memory [29] is used in every kernel functions (Step 1, 2, 4, 5). Because it is on-chip, shared memory has much higher bandwidth and much lower latency than global memory. By using shared memory, we can get maximum performance of the JDL implementation.

3) The data for the JDL implementation are double-precision. In step 3, only when the data are double-precision, the rank of $\tilde{\boldsymbol{R}}_l$ is full rank, then we can get the correct result of step 3. In this case, double-precision performance (FP64) should be considered for the JDL implementation.

The JDL algorithm steps are shown in Fig. 6, but the actual data needs to be cycled due to the limitation of GPU memory capacity. The number of cycles is equal to the ratio of the total amount of data to the memory capacity. The computation process can be optimized according to different computing graphics cards to improve the utilization rate of memory and reduce the number of cycles. This is the special feature of this paper that uses GPU computing.

### 3.4 Comparison between the JDL and existing method

In order to improve the target detection ability in ionospheric clutter, a series of STAP algorithm have been proposed to suppress clutter. The dimensionality reduction STAP processing methods include JDL processing method, D3 processing method, and D3-JDL processing method [30]. The dimensionality reduction STAP reduces the degree of freedom (DOF) of the system, thereby reducing the clutter degree of freedom in the angle-Doppler region to be processed, achieving the goal of reducing the demand for training samples and reducing computational complexity.

D3 algorithm processed on the cell under test data only, which does not statistical processing on the range domain. Therefore, D3 has no requirement for the correlation of range domain. D3 method obtains training data samples by smoothing and removing target data information in the Doppler domain and angle domain respectively. By seek the optimal weight vector and constructs the space-time steering to achieve the dimensionality reduction. Ideally, the Doppler and angle information of the target can be completely removed, but there may be mismatch issues due to sampling, which will not have a significant impact on the processing result. The JDL processing and the D3-JDL are both better than the D3 processing in terms of both homogeneous clutter and non-homogeneous clutter from the performance of clutter suppression, whereas the D3 processing method is more suitbale for suppressing non-homogeneous clutter [31].

For the comparison between JDL and other method, there are two aspects as follows:

1) Clutter suppression performance. From the perspective of clutter suppression, whether it is stationary or nonstationary clutter, the JDL processing method and D3-JDL method have good suppression effects, while the D3 processing method performs poorly in suppressing stationary clutter, mainly highlighting its suppression performance for nonstationary clutter.

2) Computational complexity. The JDL processing method has the smallest computational complexity, while the D3-JDL method has the largest computational complexity.

Overall, the JDL processing method is more suitable for High Frequency Radar systems.

## 4. Experimental results

To evaluate the proposed method, the experiment was conducted on a Linux 64bit machine (Centos 6.5) with 2x Intel Xeon E5-2609V2 4 core @ 2.5GHz, 64GB DDR3 memory, a 512GB Crucial SSD and NVIDIA Tesla K40c 2880 cores. The FP64 of Tesla K40c is 1.42Tflops which accommodate our needs regarding parallel computation. The data-cube in this experiment is formed by 369 Doppler frequency unit, 200 range units and 31 angle units. The LPR in JDL is formed by 3 angle units and 3 Doppler frequency units. So the transformed space-time signal vector *Host_X* is formed by $29 \times 367 \times 200$ cells and the transformed steering vector *Host_V* is also formed by $29 \times 367 \times 200$ cells.

### 4.1 CPU-based implementation of JDL

For the implementation of JDL on Intel E5-2609V2, we use Intel Math Kernel Library (MKL) which provides abundant math libraries, such as BLAS and LAPACK linear algebra routines, fast Fourier transforms, vectorized math functions [32, 33]. As it does not provide the ability to make the cells calculating parallelly. We had to loop the processing to process the whole three-dimensional data-cube. Table 1 shows statistical properties of our CPU implementation of JDL.

**Table 1** Average time cost for JDL on Intel E5-2609V2

| | Average time cost (ms) | | | |
|---|---|---|---|---|
| | $\tilde{\boldsymbol{R}}_l = \tilde{\boldsymbol{X}}_l \tilde{\boldsymbol{X}}_l^H$ | $\tilde{\boldsymbol{R}} = \sum_{i=0}^{P-1} \tilde{\boldsymbol{R}}_i / P$ | $\tilde{\boldsymbol{w}} = \tilde{\boldsymbol{R}}^{-1} \tilde{\boldsymbol{v}}$ | $y = \tilde{\boldsymbol{w}}^H \tilde{\boldsymbol{X}}_l$ |
| Each step | $1.02 \times 10^{-4}$ | $6.95 \times 10^{-4}$ | $1.08 \times 10^{-2}$ | $1.03 \times 10^{-4}$ |
| One cell | $1.17 \times 10^{-2}$ | | | |
| Whole cube | 24904.6 | | | |

### 4.2 GPU-based implementation of JDL

For NVIDIA Tesla K40c, it can make 367×200×6 cells processed together. So only 5 loops are needed for the whole data-cube. Before the implementation of JDL on GPU, we had to initialize the GPU global memory, and then copy the data from CPU memory to GPU global memory. When the implementation of JDL is over, the data should also be copied from GPU global memory to CPU memory. Thus, the time cost of copying data should be considered. Table 2 shows statistical properties of our GPU implementation of JDL.

Comparing with CPU platform, GPU provides parallel processing more effectively with few loops. It was clear that

the GPU-based implementation can improve the speedup to 24.72 times, thus indicating that GPUs are very well suited to JDL algorithm. Fig. 9 shows partial enlargement of the range-Doppler maps (RDMaps) obtained in practical HFSWR before and after JDL processing. And the Signal to Clutter Ratio increased with the JDL method as shown in Fig.9. As indicated in Fig. 9(a), the ionospheric interference covers part of the RDMaps, the target in this area cannot be detected. After JDL processing, the target can be detected directly, as shown in Fig. 9(b). The results show that the presented algorithm is effective. indicate

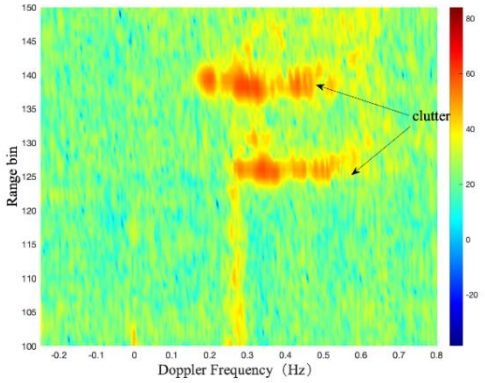
**Table 2** Average time cost for JDL on Tesla K40c

| | Average time cost (ms) | | | | |
|---|---|---|---|---|---|
| | Loop 1 | Loop 2 | Loop 3 | Loop 4 | Loop 5 |
| Copy data from CPU to GPU | 113.62 | 113.95 | 113.88 | 113.98 | 113.42 |
| $\tilde{\boldsymbol{R}}_l = \tilde{\boldsymbol{X}}_l \tilde{\boldsymbol{X}}_l^H$ | 2.23 | 2.20 | 1.99 | 2.00 | 1.99 |
| $\tilde{\boldsymbol{R}} = \sum_{i=0}^{P-1} \tilde{\boldsymbol{R}}_i / P$ | 8.70 | 8.66 | 7.98 | 7.97 | 7.95 |
| $\tilde{\boldsymbol{R}}^{-1}$ | 66.62 | 65.48 | 65.20 | 65.03 | 65.55 |
| $\tilde{\boldsymbol{w}}$ | 8.46 | 8.49 | 8.49 | 8.50 | 8.48 |
| $y = \tilde{\boldsymbol{w}}^H \tilde{\boldsymbol{X}}_l$ | 1.39 | 1.39 | 1.39 | 1.39 | 1.39 |
| Copy data from GPU to CPU | 2.07 | 2.10 | 1.66 | 2.07 | 1.64 |
| Whole cube | 1007.31 | | | | |

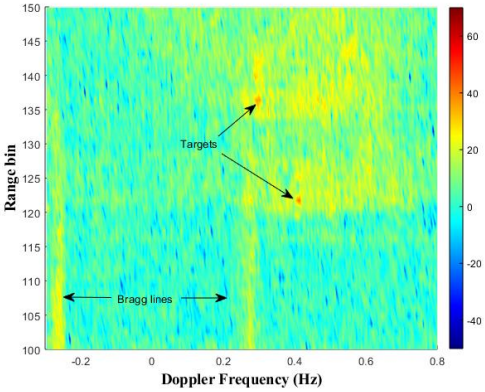
The CPU has a large amount of memory, but the storage space of the GPU is limited, it is necessary to use a cyclic approach for data processing in GPU which contribute to the significant differences in the definitions of tables and columns between Table 1 and Table 2.

## 5. Conclusion

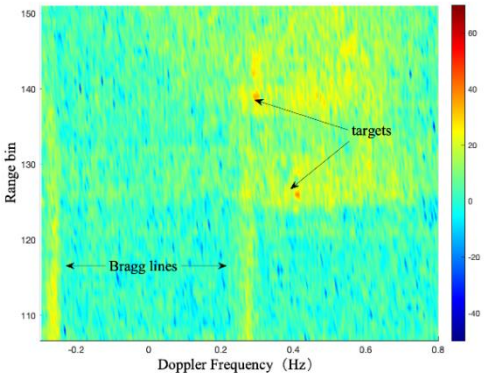
JDL has been proved to be an effective algorithm for ionospheric clutter suppression for HFSWR. As the computational cost of JDL is considered too expensive. In HFSWR system, the real-time implementation of JDL is impossible in the CPU-based platform. In this paper, we proposed a real-time implementation of JDL algorithm for HFSWR. The experiment results confirmed that the proposed method accelerates the computation by over 24.72 times as compared to the CPU-based implementation which meets the real-time requirement of HFSWR. By comparing Fig. 9 (a) and Fig. 9 (b), it was clear that after JDL algorithm processing the clutter can be greatly suppressed and targets can be directly detected. Meanwhile the clutter suppression result of the proposed GPU-based method is show in Fig. 9 (c), which obtains the completely same results comparing with the conventional CPU-based method. Although the detection performance of GPU-based implementation same from the CPU that the parallel computing power of GPU far exceeds that of CPU. Therefore, by applying JDL algorithm into real HFSWR system, the ability of the target detection is significantly improved.




(a) RDMap of HFSWR without JDL




(b) RDMap of the conventional CPU-based with JDL




(c) RDMap of the GPU-based with JDL

**Fig. 9** RDMaps comparation of HFSWR based on GPU and GPU

## Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant 62201563, Grant 61701140 and Grant 61171182, and research project fund of Songjiang Laboratory (No. SL20230204).

### References

[1] Y. Qiang, et al.: "An approach to detecting the targets of aircraft and ship together by over-the-horizon radar," 2001 CIE International Conference on Radar Proceedings (2001) 95 (DOI: 10.1109/ICR.2001.984631)

[2] S. Park, et al.: "Compact HF surface wave radar data generating

simulator for ship detection and tracking," IEEE Geoscience and Remote Sensing Letters 14 (2017) 969 (DOI: 10.1109/LGRS.2017.2691741)
[3] Y. Wei, et al.: "Experimental analysis of a HF hybrid sky-surface wave radar," IEEE Aerospace and Electronic Systems Magazine 33 (2018) 32 (DOI: 10.1109/MAES.2018.170036)
[4] G. Cirillo, et al: "Echo simulator systems for exomars 2016 radar doppler altimeters tests," 2015 IEEE Metrology for Aerospace (MetroAeroSpace) (2015) 513 (DOI: 10.1109/MetroAeroSpace.2015.7180710)
[5] L. Sevgi, et al.: "An integrated maritime surveillance system based on high-frequency surface-wave radars, part 1: theoretical background and numerical simulations," IEEE Antennas and Propagation Magazine 43 (2001) 28 (DOI: 10.1109/74.951557).
[6] L. Huang, et al: "Ionospheric interference suppression in HFSWR," IEEE Conference on Industrial Electronics and Applications (2006) (DOI: 10.1109/ICIEA.2006.257236)
[7] H. Zhou, et al: "Ionospheric clutter suppression in HFSWR using multilayer crossed-loop antennas," IEEE Geoscience and Remote Sensing Letters 11 (2014) 429 (DOI: 10.1109/LGRS.2013.2264531)
[8] W.L. Melvin: "A STAP overview," IEEE Aerospace and Electronic Systems Magazine 19 (2004) 19 (DOI: 10.1109/MAES.2004.1263229).
[9] R. Klemm: "Applications of Space-Time adaptive processing, part VII: over-the-horizon radar applications," IET Radar, Sonar and Navigation Series 14 (2004) 603 (DOI: 10.1049/PBRA014E).
[10] K.P. Ong, et al: "Angular bin compression for joint domain localized (JDL) processor," Twelfth International Conference on Antennas and Propagation (ICAP 2003) (2003) 353 (DOI: 10.1049/cp:20030086)
[11] R.S.Adve, et al: "Practical joint domain localised adaptive processing in homogenoeous and nonhomogeneous environments. Part I : Homogeneous environments," IEE Proceedings-Radar, Sonar and Navigation 147 (2000) 57 (DOI: 10.1049/ip-rsn:20000035)
[12] X. Zhang, et al: "Ionospherci clutter suppression method based on STAP," Systems Engineering and Electronics 35 (2013) 1177
[13] NVIDIA Corporation: CUDA C Programming Guide 10.2 (2019) https://docs.nvidia.com/cuda/cuda-c-programming-guide
[14] M. D. Mccool: "Signal Processing and General-Purpose Computing and GPUs [Exploratory DSP]," IEEE Signal Processing Magazine 24 (2007) 109 (DOI: 10.1109/MSP.2007.361608 )
[15] C. Zhang, et al: "High frequency radar signal processing based on the parallel technique," IET International Radar Conference 2015 (2015) 2903 (DOI: 10.1049/cp.2015.1246)
[16] S. P. Mohanty: "GPU-CPU multi-core for real-time signal processing," 2009 Digest of Technical Papers International Conference on Consumer Electronics (2009) P-1-2 (DOI: 10.1109/ICCE.2009.5012160)
[17] R. Benenson, et al: "Pedestrian detection at 100 frames per second," 2012 IEEE Conference on Computer Vision and Pattern Recognition (2012) 2903 (DOI**:** 10.1109/CVPR.2012.6248017)
[18] F. James, M. Steve: "Using graphics devices in reverse: GPU-based Image Processing and Computer Vision," IEEE International Conference on Multimedia and Expo (2008) (DOI: 10.1109/ICME.2008.4607358)
[19] D. Pavel, et al: "Pattern Recognition in EEG Cognitive Signals Accelerated by GPU," International Joint Conference CISIS'12-ICEUTE´12-SOCO´12 Special Sessions (2013) 477 (DOI: 10.1109/COASE.2007.4341818)
[20] H. Jang, et al: "Neural network implementation using CUDA and OpenMP," 2008 Digital Image Computing: Techniques and Applications (2008) 155 (DOI: 10.1109/DICTA.2008.82)
[21] P. Li, et al: "HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms," IEEE International Conference on Networking, Architecture and Storage (NAS),(2015) 347 (DOI: 10.1109/NAS.2015.7255222)
[22] L. Baldini, et al: "Predicting GPU Performance from CPU Runs Using Machine Learning," IEEE 26th International Symposium on Computer Architecture and High Performance Computing (2014) 254 (DOI: 10.1109/SBAC-PAD.2014.30)
[23] F. Zhang, et al: "Multiple mode SAR raw data simulation and parallel acceleration for Gaofen-3 Mission," IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing 11 (2018) 2115 (DOI: 10.1109/JSTARS.2017.2787728)
[24] P. Kang: "GPU-accelerated stochastic simulation of biochemical networks," IEICE Trans.INF. & SYST E101-D (2018) 786 (DOI: 10.1587/transinf.2017EDL8218)
[25] N. Kath, et al: "Robust GPU-based Virtual Reality Simulation of Radio Frequency Ablations for Various Needle Geometries and Locations," International Journal of Computer Assisted Radiology and Surgery (2019) 1 (DOI: 10.1007/s11548-019-02033-w)
[26] T. M. Benson, R. K. Hersey, and E. Culpepper, "GPU-based space-time adaptive processing (STAP) for radar," in 2013 IEEE High Performance Extreme Computing Conference (HPEC), (2013) (DOI: 10.1109/HPEC.2013.6670341)
[27] H. Wang and L. Cai: "On adaptive spatial-temporal processing for airborne surveillance radar systems," IEEE Transactions on Aerospace and Electronic Systems 30 (1994) 660 (DOI: 10.1109/7.303737)
[28] NVIDIA Corporation: cuBLAS API library, https://docs.nvidia.com/cuda/cublas.html, accessed Feb.13. 2019.
[29] NVIDIA Corporation: CUDA C Programming Guide 10.1, https://docs.nvidia.com/cuda/cuda-c-programming-guide.html, accessed Feb.13. 2019.
[30] Y. Eunjung, C. Joohwan, R. Adve, and C. Jonghoon, "A hybrid D3-Sigma Delta STAP algorithm in non-homogeneous clutter," in 2007 IET International Conference on Radar Systems, 15-18 Oct. 2007 2007.
[31] M. Li, G. Sun, and Z. He, "Direct Data Domain STAP Based on Atomic Norm Minimization," in 2019 IEEE Radar Conference (RadarConf), 22-26 April 2019 (DOI: 10.1109/RADAR.2019.8835701)
[32] Intel : Intel® Math Kernel Library Developer Reference,https://software.intel.com/en-us/mkl-developer-reference-c.html, accessed Feb.15. 2019.
[33] Intel : BLAS and Sparse BLAS Routines, https://software.intel.com/en-us/mkl-developer-reference-c-blas-and-sparse-blas-routines.html, accessed Feb.15. 2019.

**Bowen ZHANG** received his B.Sc. degree in Yanbian University, Yanji, China in 2022 . Now, he is a Ph.D. student with the School of Electronic and Information Engineering, Harbin Institute of Technology, Harbin, Heilongjiang,150001, China. His research interests include array signal processing, clutter and interference suppression, and radar signal processing.

**Chang ZHANG** received his B.Sc. degree in electronic information engineering, and the M.Sc. degree in information and communication engineering from Harbin Institute of Technology, Harbin, China, in 2009 and 2020, respectively. He is currently a research assistant in Jiangsu Automation Research Institute (JARI), Lianyungang, China. His research interests include radar signal processing and parallel computing.

**Di YAO** received the B.Sc. degree in electronic information engineering and the M.Sc. degree in underwater acoustic engineering from Harbin Engineering University, Harbin, China, in 2011 and 2014, respectively, and the Ph.D. degree in information and communication engineering from the Harbin Institute of Technology, Harbin, in 2019.

He is currently an associate professor with the School of Computer Science and Engineering, Northeastern University, Shenyang 110819, China. His research interests include array signal processing, clutter suppression, and radar signal processing.

**Xin ZHANG** received the B.Sc., M.Sc., and Ph.D. degrees in information and communication engineering from the Harbin Institute of Technology, Harbin, China, in 2005, 2011, and 2016, respectively. He is currently a associate Professor with the School of Electronics and Information Engineering, Harbin Institute of Technology. His research interests are in radar signal processing, clutter suppression, space-time adaptive processing, and compressed sensing.