remote sensing

MDPI

Article

A Deep Pipelined Implementation of Hyperspectral Target Detection Algorithm on FPGA Using HLS

Jie Lei [1,*], Yunsong Li [1,*], Dongsheng Zhao [1], Jing Xie [1,*], Chein-I Chang [2], Lingyun Wu [1], Xuepeng Li [1], Jintao Zhang [1] and Wenguang Li [1]

1 State Key Laboratory of Integrated Services Networks, School of Telecommunications Engineering, Xidian University, Xi'an 710071, China; dongshengzhao@stu.xidian.edu.cn (D.Z.); lywu@stu.xidian.edu.cn (L.W.); xuepengli@stu.xidian.edu.cn (X.L.); jtzhang_1@stu.xidian.edu.cn (J.Z.); wgli@stu.xidian.edu.cn (W.L.)

2 Department of Computer Science and Electrical Engineering, University of Maryland, Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250, USA; cchang@umbc.edu

* Correspondence: jielei@mail.xidian.edu.cn (J.L.); ysli@mail.xidian.edu.cn (Y.L.); jingxie@stu.xidian.edu.cn (J.X.); Tel.: +86-029-8820-3116 (J.L.)

Received: 2 January 2018; Accepted: 22 March 2018; Published: 25 March 2018

check for updates

Abstract: Real-time target detection for hyperspectral images (HSI) has received considerable interest in recent years. However, owing to enormous data volume provided by HSI, detection algorithms are generally computationally complex, thus developing rapid processing techniques for target detection has encountered several challenging issues. It seems that using a deep pipelined structure can improve the detection speed, and implementing on field programmable gate arrays (FPGAs) can also achieve concurrent operations rather than run streams of sequential instruction. This paper presents a deep pipelined background statistics (DPBS) approach to optimizing and implementing a well-known subpixel target detection algorithm, called constrained energy minimization (CEM) on FPGA by using high-level synthesis (HLS). This approach offers significant benefits in terms of increasing data throughput and improving design efficiency. To overcome a drawback of HLS on implementing a task-level pipelined circuit that includes a feedback data path, a script based circuit design method is further developed to make connections between some of the modules created by HLS. Experimental results show that the proposed method can detect targets on a real-hyperspectral data set (HyMap Data) only in 0.15 s without compromising detection accuracy.

Keywords: hyperspectral image; deep pipelined background statistics; constrained energy minimization; high-level synthesis; real-time processing

1. Introduction

Hyperspectral remote sensing imaging acquires three-dimensional (3D) data including two spatial dimensions with space information of pixels and one spectral dimension with high-dimensional reflectance vectors [1]. The rich spectral information provided by HSI is very useful and has been widely used in a range of various applications such as ecology [2], agriculture [3], environmental [4] and geology [5], where target detection plays a crucial role [6–9]. There are many algorithms have been developed for target detection in HSI [1], such as matched filter (MF) [10], spectral angle mapper (SAM) [11], constrained energy minimization (CEM) [12], target-constrained interference-minimized filter (TCIMF) [13], adaptive coherence estimator (ACE) [14], matched subspace detector (MSD) [15], orthogonal subspace projection (OSP) [16], and sparsity-based target detector (STD) [17]. Among them, CEM along with its variants have been widely used for hyperspectral target detection. The effectiveness of CEM has been shown successfully in many applications such as reconnaissance, rescue, search and on-orbit processing [6]. For such applications, the high-speed data processing of HSI is generally required for finding targets on a timely basis. However, as a trade-off the volume of data in HSI has also become unmanageable with increasing spectral and spatial resolutions. In this case, CEM needs a significantly large number of complex matrix computations. The algorithm could be implemented on one of the widely used platforms like CPUs, GPUs [18], and FPGAs [19]. Among them, FPGAs have significant advantages in supporting parallel computation and customizable deep pipeline operation with the cost of low power consumption.

Currently, great progress has been made in implementing target detection algorithms on FPGAs. For example, Chang described a new FPGA design by using the Coordinate Rotation Digital Computer (CORDIC) algorithm to solve the matrix inversion problem of the classical CEM [20]. This method of computing the inverse of a large matrix is unable to support fast target detection. Yang utilized Streaming Background Statistics (SBS) structure with an idea of continuously updating the inverse of the correlation matrix on FPGA [21]. Despite that a pixel-by-pixel processing design is realized using fewer hardware resources, its data processing speed is not high. Recently, Gonzalez C. et al. proposed an FPGA implementation of the automatic target-generation process based on an orthogonal subspace projector (ATGP-OSP) using the pseudoinverse operation [22], where Gauss-Jordan elimination method was selected for computing the inverse of a small square matrix whose size is no more than 32×32. Unfortunately, the Gauss-Jordan elimination method consumes too much logic resources for solving the large matrix inversion problem required by the CEM.

Although FPGAs gain much attention, it is still not widely deployed for accelerating many algorithms that require high computational complexity such as CEM. The main reason is that the conventional development methods of FPGAs, which are based on register transfer level (RTL) hardware description, are much more difficult than that of CPUs or GPUs. It commonly requires great efforts in achieving highly efficient results on FPGAs. Furthermore, a design method based on RTL for FPGAs lacks portability and flexibility compared to those based on C/C++ for CPUs or GPUs. To close this gap, FPGA vendors and developers have begun to take advantage of high-level synthesis (HLS) to work on FPGA applications. HLS is able to convert high abstraction languages such as C, C++ and SystemC into VHDL/Verilog hardware description language (HDL) for RTL-level circuit design. According to the user-defined constraints and C code style, the efficiency of the converted RTL designs are quite different. Until now, studies on the acceleration of hyperspectral data processing algorithms with HLS are already available. Santos proposed a novel adaptive and predictive algorithm for lossy hyperspectral image compression algorithm [23] and Lossy Compression for Exomars (LCE) algorithm [24] described in Vivado HLS. Domingo R. et al. proposed a hyperspectral image spatial-spectral classifier accelerator using Intel FPGA SDK for OpenCL [25]. What's more, HLS is popularly utilized for hardware acceleration of deep learning algorithms like convolution neural networks (CNN) [26,27]. However, no research work has been reported for implementing CEM on FPGA using HLS.

In this work, a DPBS-CEM algorithm is developed to be implemented on FPGA using HLS for real-time hyperspectral target detection. Like SBS-CEM, the inverse matrix is gradually updated according to a Sherman-Morrison formula [28]. Different from using sliding windows, DPBS-CEM takes advantage of cumulative windows instead to greatly reduce the number of calculations. As for the issue of removing data dependency in updating inverse matrices, separate memories are proposed to store the results of the successive inverse matrices, which make sure the operations on adjacent pixels can be processed independently. As a consequence, a deep pipelined implementation of DPBS-CEM can be further developed, which has an extraordinary performance improvement in terms of data throughput. Experimental results demonstrate that the proposed algorithm has the capability of operating at a high-speed rate of more than 200 MHz on FPGA. Setting the same clock frequency, the algorithm can also achieve a significant speed-up of near $7.3\times$ than SBS-CEM [21] with no compromise for detection accuracy.

The contributions of this paper can be summarized as follows.

- A novel deep pipelined architecture is proposed to accelerate the proposed DPBS-CEM algorithm on FPGA using HLS. It outperforms the previous work designed with RTL in terms of data throughput performance.
- A solution is derived to remove the data dependency existing in SBS-CEM for updating the inverse matrices, by allowing four adjacent pixels to update their own individual inverse matrices that are stored in four different memories.
- The proposed structure can be simply rebuilt to support diverse HSI implementations with different spatial resolution and number of spectral bands through several parameters modified under HLS. Most importantly, the framework can support various operation modes including split/non-split data and local/global detection. It is easily adapted to match multiple rates of hyperspectral imagery.
- Last but not least, alternative solutions to the problems of feedback and high fanout are also provided.

The remainder of this paper is organized as follows. Section 2 briefly discusses CEM and SBS-CEM used for target detection. Section 3 describes the principle of DPBS-CEM in great detail. The FPGA implementation of DPBS-CEM is presented in Section 4. Section 5 conducts a detailed performance analysis via extensive experiments. Finally, conclusions along with some remarks were drawn in Section 6.

2. Related Algorithms

In this section, the principles of the classical CEM and SBS-CEM algorithms are described. Besides, the problems of implementing these two algorithms in practical applications are also analysed.

2.1. CEM Algorithm

2.1.1. Principle of the CEM Algorithm

Let $\mathbf{X} \in \mathbf{R}^{W\times H\times L}$ denote a HSI with $W \times H$ pixels (row of $\mathbf{X}$) and L spectral bands (column of $\mathbf{X}$). We may interpret $\mathbf{X}$ either as a collection of L 2D images (or bands) of size N ($N = W \times H$), or as a collection of $W \times H$ spectral vectors of size L. The entire data matrix $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, ..., \mathbf{x}_N]$, where $\mathbf{x}_i$ is the ith sample pixel vector $\mathbf{x}_i = (x_{i1}, x_{i2}, ..., x_{iL})^{\mathrm{T}}$ for $1 \le i \le N$ and the signature $\mathbf{d} = (d_1, d_2, ..., d_L)^{\mathrm{T}}$ of target is known. The basic purpose of CEM is to design a linear finite impulse response (FIR) filter with L filter coefficients denoted by an L-dimensional vector $\mathbf{w} = (w_1, w_2, ..., w_L)^{\mathrm{T}}$ that minimizes the energy of the the output y_i $(1 \le i \le N)$ with the following constraint.

$$s.t. \mathbf{d}^{\mathrm{T}}\mathbf{w} = 1 \tag{1}$$

$$\min_{\mathbf{w}} \left(\mathbf{w}^{\mathrm{T}}\mathbf{R}\mathbf{w}\right) \tag{2}$$

where $\mathbf{R} = \frac{1}{N}\left[\sum_{i=1}^{N} \mathbf{x}_i \mathbf{x}_i^{\mathrm{T}}\right]$ is the global correlation matrix of $\mathbf{X}$. The weighting vector $\mathbf{w}$ solved for Equation (1) and Equation (2) is given by

$$\mathbf{w} = \frac{\mathbf{R}^{-1}\mathbf{d}}{\mathbf{d}^{\mathrm{T}}\mathbf{R}^{-1}\mathbf{d}} \tag{3}$$

which yields the CEM described by

$$y_i = \delta_{CEM}(\mathbf{x}_i) = (\mathbf{w}_{CEM})^{\mathrm{T}}\mathbf{x}_i \tag{4}$$

2.1.2. Problem Analysis

The classical CEM algorithm is a global subpixel target detector, which uses all the pixels in HSI to calculate the correlation matrix. After the correlation matrix is obtained, the process of calculating the inverse matrix is executed through many complicated steps via QR decomposition. This is a typical large matrix inversion problem, which may be the main cause of a significant latency up to obtaining the final results.

2.2. SBS-CEM Algorithm

To accelerate the task of target detection using CEM, some researchers choose to calculate local detection by using a partial set of pixel vectors instead of all data sample vectors [6,29,30]. For instance, Yang [21] proposed an FPGA-based implementation of SBS-CEM by using a new matrix inversion method to perform the correlation operation and the inversion operation simultaneously.

2.2.1. Principle of the SBS-CEM Algorithm

Unlike the classical CEM algorithm, SBS-CEM takes the inverse of the correlation matrix of the K-group pixel vectors to replace the entire pixel vectors. More specifically, SBS-CEM can be described as follows.

$$\mathbf{R}_n = (1/K)\left[\sum_{i=n-K}^{n-1} \mathbf{x}_i\mathbf{x}_i^{\mathrm{T}}\right] \tag{5}$$

$$\mathbf{S}_n = \left[\sum_{i=n-K}^{n-1} \mathbf{x}_i\mathbf{x}_i^{\mathrm{T}}\right] \tag{6}$$

Now, $\mathbf{S}_n^{-1}$ is the inverse of the correlation matrix of the K-group pixel vectors. The Sherman-Morrison formula is used to derive the following two formulas.

$$\mathbf{C}^{-1} = \left(\mathbf{S}_n + \mathbf{x}_n\mathbf{x}_n^{\mathrm{T}}\right)^{-1} = \mathbf{S}_n^{-1} - \frac{\mathbf{S}_n^{-1}\mathbf{x}_n\mathbf{x}_n^{\mathrm{T}}\mathbf{S}_n^{-1}}{\mathbf{x}_n^{\mathrm{T}}\mathbf{S}_n^{-1}\mathbf{x}_n + 1} \tag{7}$$

$$\mathbf{S}_{n+1}^{-1} = \left(\mathbf{C} - \mathbf{x}_{n-K}\mathbf{x}_{n-K}^{\mathrm{T}}\right)^{-1} = \mathbf{C}^{-1} - \frac{\mathbf{C}^{-1}\mathbf{x}_{n-K}\mathbf{x}_{n-K}^{\mathrm{T}}\mathbf{C}^{-1}}{\mathbf{x}_{n-K}^{\mathrm{T}}\mathbf{C}^{-1}\mathbf{x}_{n-K} - 1} \tag{8}$$

Based on this streaming framework, the inverse matrix can be updated by using Equations (7) and (8). When applying the Sherman-Morrison formula, the initial value of $\mathbf{S}_0^{-1}$ should be set. Let $\mathbf{S}_0^{-1} = \beta \cdot \mathbf{I}$; then $\mathbf{S}_{K+1}$ can be expressed as:

$$\mathbf{S}_{K+1} = (1/\beta)\cdot\mathbf{I} + \mathbf{x}_1\mathbf{x}_1^{\mathrm{T}} + \mathbf{x}_2\mathbf{x}_2^{\mathrm{T}} + \cdots + \mathbf{x}_K\mathbf{x}_K^{\mathrm{T}} \tag{9}$$

Among them, the matrix $(1/\beta)\cdot\mathbf{I}$ does not affect the performance of the detector. On the contrary, it makes the detection results be more stable [31]. The detection equation of the SBS-CEM algorithm is then derived as:

$$SBS - CEM\,(\mathbf{x}) = \frac{K\left(\mathbf{x}^{\mathrm{T}}\mathbf{S}^{-1}\mathbf{d}\right)}{K\left(\mathbf{d}^{\mathrm{T}}\mathbf{S}^{-1}\mathbf{d}\right)} = \frac{\mathbf{x}^{\mathrm{T}}\mathbf{S}^{-1}\mathbf{d}}{\mathbf{d}^{\mathrm{T}}\mathbf{S}^{-1}\mathbf{d}} \tag{10}$$

Since the pixel to be detected is located in the middle of the window, SBS-CEM can also be expressed as:

$$SBS - CEM\,(\mathbf{x}_{n-K/2}) = \frac{\mathbf{x}_{n-K/2}^{\mathrm{T}}\mathbf{S}_n^{-1}\mathbf{d}}{\mathbf{d}^{\mathrm{T}}\mathbf{S}_n^{-1}\mathbf{d}} \tag{11}$$

2.2.2. Problem Analysis

Sliding window problem. Compared to the classical CEM algorithm, SBS-CEM does not need the full image data sample vectors to compute the correlation matrix. Instead, a local region of the image defined by a sliding window is utilized to capture the local statistics. The size of the sliding window is fixed and set to L^2 (square of the number of spectral bands) in the SBS-CEM algorithm. The fixed size of the sliding window requires the compute-intensive task of calculating the Sherman-Morrison formula to be performed twice (as shown in Equations (7) and (8)) for each update of the inverse matrix. However, the extra calculation of the Equation (8) does not provide appreciable improvements of the target detection accuracy according to our experimental results.

Data dependency problem. The problem of data dependency exists in the process of updating the inverse matrix where the calculation for $\mathbf{S}_{n+1}^{-1}$ cannot be started until the $\mathbf{S}_n^{-1}$ is available. Unless $\mathbf{S}_n^{-1}$ is ready, it is not possible to compute $\mathbf{S}_{n+1}^{-1}$. SBS-CEM divides the process of updating the inverse matrix into several stages to reduce its complexity. Unfortunately, under such circumstance, several stages' time consumption has to spend waiting for each inverse matrix updating. This computation overhead would be the major bottleneck of the SBS-CEM's data throughput performance.

3. Algorithm Optimization

3.1. Principle of Algorithm Optimization

To solve the problems described above for SBS-CEM, an optimized algorithm is proposed in this section. The two main improvements are proposed to deal with the use of sliding windows and to remove data dependency.

Non-sliding window. We choose not to use sliding windows to update calculations of the inverse matrix, which is quite different from the SBS-CEM algorithm. With no requirement for moving out the oldest pixel, the Equation (8) can be removed and thus a large number of calculations can be therefore reduced. When a new pixel vector $\mathbf{x}_n$ is loaded into the window, we can obtain the output value $\mathbf{S}_{n+1}^{-1}$ by Equation (12).

$$\mathbf{S}_{n+1}^{-1} = \left(\mathbf{S}_n + \mathbf{x}_n\mathbf{x}_n^{\mathrm{T}}\right)^{-1} = \mathbf{S}_n^{-1} - \frac{\mathbf{S}_n^{-1}\mathbf{x}_n\mathbf{x}_n^{\mathrm{T}}\mathbf{S}_n^{-1}}{\mathbf{x}_n^{\mathrm{T}}\mathbf{S}_n^{-1}\mathbf{x}_n + 1} \tag{12}$$

Data segmentation for deep pipeline. As mentioned above, the SBS-CEM algorithm runs calculations of matrix inversions in serial. Since data dependency exists between $\mathbf{S}_{n+1}^{-1}$ and $\mathbf{S}_n^{-1}$, there is a great increase in processing time. To solve this problem, we need to complete the computation of Equation (12) in four stages and apply pipeline optimization for achieving pipeline acceleration. However, updating the inverse matrix between adjacent pixels is not independent, which prevents the use of the optimization strategy of deep pipeline. If we want to achieve a deep pipelined design, we have to make sure there is no feedback or iterations among the stages. In this case, we solve the data dependency by means of data segmentation. As a result, the current input pixel can be processed directly with no need of waiting for the previous pixel to be completed. By making the inverse calculations between neighbouring pixels independent, we are able to carry out a deep pipelined architecture, which can achieve 8× speed-up compared to SBS-CEM in theory.

Table 1, derived from the evaluation of hardware calculation, shows that the number of computations for each stage is different, but the number of clock cycles consumed by each stage is approximately equal after being parallelized. Where $\mathbf{x}_n$ $\left(\mathbf{P} = \mathbf{x}_n^{\mathrm{T}}\right)$ represents a column of $\mathbf{X}$, T and Q are scalars. $\mathbf{S}_n^{-1}$ is denoted by $\mathbf{U}$, which is an L-dimensional matrix. In addition, the detail procedure of DPBS-CEM algorithm is shown as Algorithm 1.

Table 1. Four stages of the inverse matrix update.

Stage Number	Formula	Flop $(\times : \pm)$	Parallelism	Clock Cycles
1	$\mathbf{h} = \mathbf{U}\mathbf{p}^{\mathrm{T}}$	$(L^2 : L^2)$	L	L
2	$T = \mathbf{p}\mathbf{h}$	$(L : L)$	1	L
3	$\mathbf{F} = \mathbf{h}\mathbf{h}^{\mathrm{T}}$ $Q = \frac{1}{T+1}$	$(L^2 : 0)$ $(0 : 1)$	L	L
4	$\mathbf{S}_{n+1}^{-1} = \mathbf{U} - \mathbf{F}Q$	$(L^2 : L^2)$	L	L

Algorithm 1 The deep pipelined background statistics (DPBS) target detection CEM algorithm

```
Input: Initialize the following parameters.
(1) HSI data size: $W \times H \times L = N \times L$;
(2) the value of $\beta$;
(3) the desired signature $\mathbf{d}$;
(4) the number of inverse matrices: $M = 4$;
(5) $bn$ indicates the index of number;
(6) $K$ indicates the number of pixel vectors collected before starting target detection;
Output: the final target detection results.
define an initial inverse matrix $S_0^{-1}$: $S_0^{-1} = \beta \cdot \mathbf{I}$
data segmentation:
for $i = 1 \,; i \le N + K \,; i + +$ do
    $bn = i \,\%\, M$
    calculate the inverse matrix:
    if $i \le N$ then
        $\left(\mathbf{S}^{-1}\right)^{\mathrm{bn}} = \left(\mathbf{S}^{\mathrm{bn}} + \mathbf{x}_i\mathbf{x}_i^{\mathrm{T}}\right)^{-1} = \left(\mathbf{S}^{-1}\right)^{\mathrm{bn}} - \frac{\left(\mathbf{S}^{-1}\right)^{\mathrm{bn}}\mathbf{x}_i\mathbf{x}_i^{\mathrm{T}}\left(\mathbf{S}^{-1}\right)^{\mathrm{bn}}}{\mathbf{x}_i^{\mathrm{T}}\left(\mathbf{S}^{-1}\right)^{\mathrm{bn}}\mathbf{x}_i+1}$
    endif
    calculate the target detection results:
    if $i \ge K$ then
        $DPBS - CEM\left(\mathbf{x}_{i-K}\right) = \frac{\mathbf{x}_{i-K}^{\mathrm{T}}\left(\mathbf{S}^{-1}\right)^{\mathrm{bn}}\mathbf{d}}{\mathbf{d}^{\mathrm{T}}\left(\mathbf{S}^{-1}\right)^{\mathrm{bn}}\mathbf{d}}$
    endif
endfor
```

3.2. Design Challenges

Feedback. There is a feedback problem in updating the inverse matrix. In fact, the inverse matrix updated in the fourth stage has to be transmitted back to the first stage as an input operand for the next updating. All of the stages are described by individual C/C++ functions. To substantially accelerate the process of updating the inverse matrix, we have to apply the data flow optimization directly to these functions so that the HLS tool can be guided to implement a task-level pipelining. Unfortunately, the HLS tool will not take place if it detects a feedback among the functions. As a result, the task-level pipelining cannot be achieved only using HLS directly.

Fanout. Due to the use of a large number of bands, there are some high fanout cases where some registers need to drive lots of loads like multipliers, which result in longer path delay and lower clock frequency. For example, in the fourth stage as described in Table 1, the scalar Q needs to be multiplied by L elements of a column in the matrix **F** simultaneously after parallel computation applied. It means that the element of the scalar Q has a high fanout to drive as much as L slave modules. It is simple to solve the high fanout problem by means of duplicating registers when designing with RTL, but it is not easy with HLS.

4. FPGA Implementation

In this section, an overall hardware structure of DPBS-CEM is given in Section 4.1. Section 4.2 describes the internal architecture of the inverse matrix updater in detail along with its workflow of deep pipeline. The difficulties in developing the hardware framework of DPBS-CEM using the HLS tool and their solutions are discussed in Section 4.3. Section 4.4 briefly introduces a few particular features of the proposed FPGA implementation of DPBS-CEM.

4.1. Overall Hardware Architecture of DPBS-CEM

As shown in Figure 1, the framework of DPBS-CEM mainly consists of three components including an off-chip memory, a processor core, and a scheduler. The off-chip memory (DDR3 SDRAM) is utilized to cache the hyperspectral image pixels. The processor core is responsible for the data processing of

DPBS-CEM, which involves three modules: the first module is an inverse matrix updater, dedicated to update the inverse matrix in five stages; the second module is a spectral pixel filter, applied to filter pixels in four stages; and the last module is a storage component, utilized to cache the inverse matrix. Finally, the third component is scheduler which is designed to schedule the two modules of inverse matrix updater and spectral pixel filter.

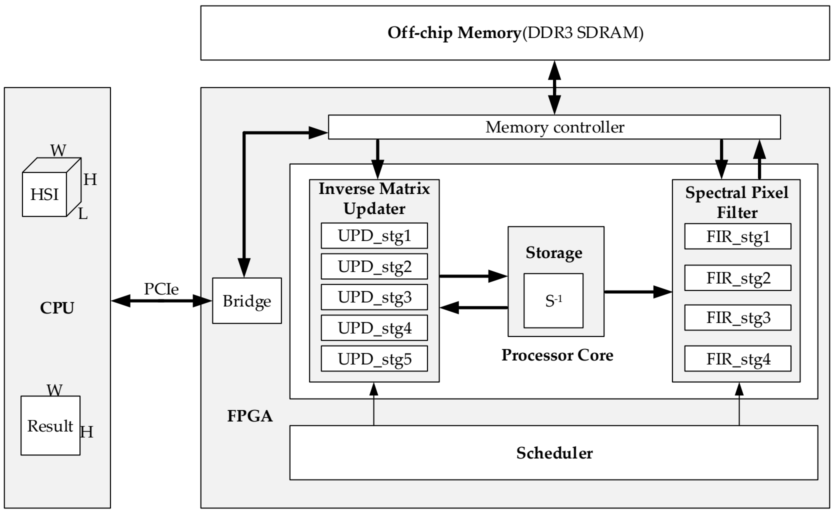


Figure 1. Overall hardware structure of DPBS-CEM.

4.2. *Update Process of Inverse Matrix*

4.2.1. Internal Architecture

As described in Figure 2, the inverse matrix updater contains five processing stages for updating the inverse matrices and four memory buffers for independently caching inverse matrices associated with four successive pixels. The four individual memories are allocated for solving the problem of data dependency described in Section 2.2.2. The specific calculations of each stage, the data flow, and the access mode of inverse matrices are clearly displayed in Figure 2. In addition, we arrange five blocks (Block A in Figure 3a, Block B in Figure 4a, Block C in Figure 5a, Block D in Figure 6a, and Block E in Figure 7a) to realize the last four processing stages in Figure 2, and we also provide pieces of C/C++ code written in HLS for these blocks on the right side of the Figures. In the Appendix A, the features of the #pragma used in these pieces of code are explained in Table A1.

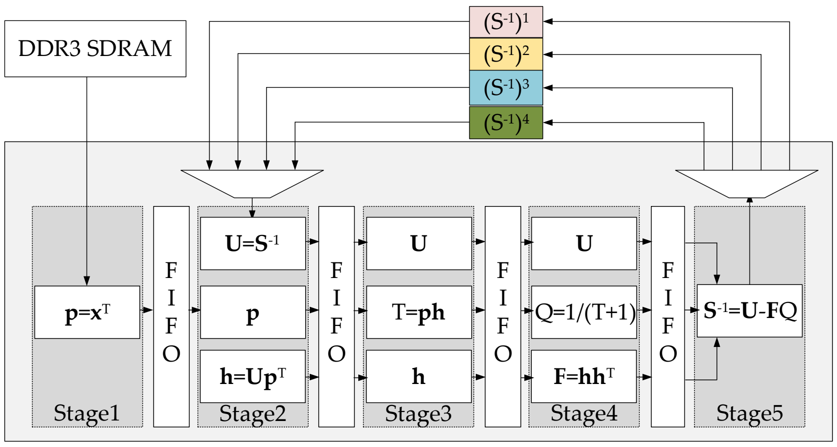
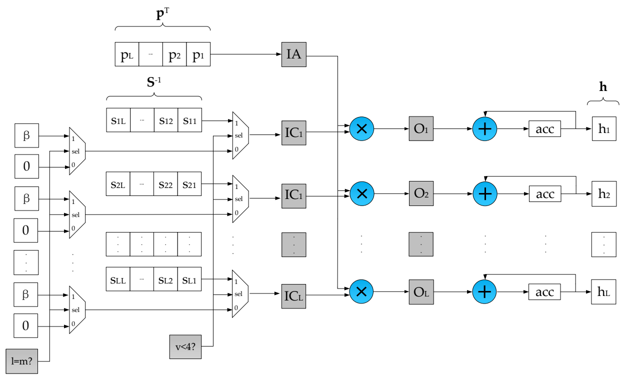


Figure 2. Block diagram of inverse matrix updater.

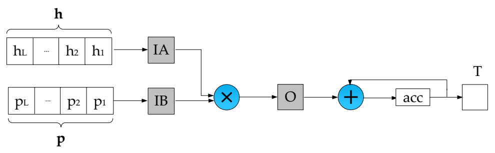


(**a**)

```
void Block_A(datatype_mid h[L],datatype_in pt[L],
          datatype_mid S_1_i[L][L][4],int v){
#pragma HLS INTERFACE ap_none port=h
#pragma HLS ARRAY_PARTITION variable=h complete dim=1
#pragma HLS INTERFACE ap_none port=pt
#pragma HLS ARRAY_PARTITION variable=pt complete dim=1
#pragma HLS INTERFACE ap_memory port=S_1_i
#pragma HLS ARRAY_PARTITION variable=S_1_i complete dim=1
     datatype_mid S_1_col[L]={0};
#pragma HLS ARRAY_PARTITION variable=S_1_col complete dim=1
     datatype_mid h_mid1[L]={0};
#pragma HLS ARRAY_PARTITION variable=h_mid1 complete dim=1
     datatype_mid h_mid2[L]={0};
#pragma HLS ARRAY_PARTITION variable=h_mid2 complete dim=1
     for(unsigned char l=0;l<L;l++)
#pragma HLS UNROLL
          h_mid1[l]=0;
     for(unsigned char m=0;m<L;m++){
#pragma HLS PIPELINE II=1
          for(unsigned char l=0;l<L;l++){
               S_1_col[l]=S_1_i[l][m][v%4];
               if (v<4){
                    S_1_col[l]=0;
                    if(l==m)
                         S_1_col[l]=P;
               }
               h_mid2[l]=pt[l]*S_1_col[l];
               h_mid1[l]+=h_mid2[l];
          }
     }
     for(unsigned char l=0;l<L;l++)
#pragma HLS UNROLL
          h[l]=h_mid1[l];
}
```

(**b**)

Figure 3. (**a**) Hardware structure and (**b**) C/C++ code in HLS of Block A. (v represents the pixel number, l represents the row number of the matrix $\mathbf{S}^{-1}$, and m represents the column number of the matrix $\mathbf{S}^{-1}$).

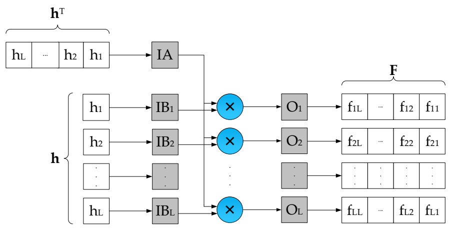


(**a**)

```
void Block_B(datatype_in h[L],datatype_mid p[L],
          datatype_mid T){
#pragma HLS INTERFACE ap_none port=h
#pragma HLS ARRAY_PARTITION variable=h complete dim=1
#pragma HLS INTERFACE ap_none port=p
#pragma HLS ARRAY_PARTITION variable=p complete dim=1
#pragma HLS INTERFACE ap_none port=T
     datatype_mid T_mid1=0;
     datatype_mid T_mid2=0;
     T_mid1=0;
     for(unsigned char m=0;m<L;m++){
#pragma HLS PIPELINE II=1
          T_mid2=h[m]*p[m];
          T_mid1+=T_mid2;
     }
     T=T_mid1;
}
```

(**b**)

Figure 4. (**a**) Hardware structure and (**b**) C/C++ code in HLS of Block B.

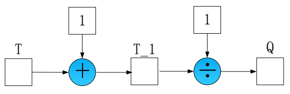


(**a**)

```
void Block_C(datatype_mid h[L],
          datatype_mid F[L][L]){
#pragma HLS INTERFACE ap_none port=h
#pragma HLS ARRAY_PARTITION variable=h complete dim=1
#pragma HLS INTERFACE ap_fifo port=F
#pragma HLS ARRAY_PARTITION variable=F complete dim=1
     for(unsigned char m=0;m<L;m++)
#pragma HLS PIPELINE II=1
          for(unsigned char l=0;l<L;l++)
               F[l][m]=h[m]*h[l];
}
```

(**b**)

Figure 5. (**a**) Hardware structure and (**b**) C/C++ code in HLS of Block C.

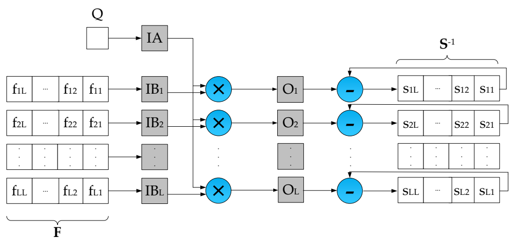


(**a**)

```
void Block_D(datatype_mid T,
          datatype_mid Q){
#pragma HLS INTERFACE ap_none port=T
#pragma HLS INTERFACE ap_none port=Q
     datatype_mid T_1=0;
     T_1=T+(datatype_mid)1;
     Q=(datatype_mid)1/T_1;
}
```

(**b**)

Figure 6. (**a**) Hardware structure and (**b**) C/C++ code in HLS of Block D.



(**a**)

```
void Block_E(datatype_mid F[L][L],
        datatype_mid U[L][L],datatype_mid Q,
        datatype_mid S_1_o[L][L][4],int v){
#pragma HLS INTERFACE ap_fifo port=F
#pragma HLS ARRAY_PARTITION variable=F complete dim=1
#pragma HLS INTERFACE ap_fifo port=U
#pragma HLS ARRAY_PARTITION variable=U complete dim=1
#pragma HLS INTERFACE ap_memory port=S_1_o
#pragma HLS ARRAY_PARTITION variable=S_1_o complete dim=1
    datatype_mid S_1_col_mid1[L]={0};
#pragma HLS ARRAY_PARTITION variable=S_1_col_mid1 complete dim=1
    datatype_mid S_1_col_mid2[L]={0};
#pragma HLS ARRAY_PARTITION variable=S_1_col_mid2 complete dim=1
    for(unsigned char m=0;m<L;m++){
#pragma HLS PIPELINE II=1
        for(unsigned char l=0;l<L;l++){
            S_1_col_mid1[l]=Q*F[l][m];
            S_1_col_mid2[l]=U[l][m]-S_1_col_mid1[l];
            S_1_o[l][m][v%4]=S_1_col_mid2[l];
        }
    }
}
```

(**b**)

Figure 7. (**a**) Hardware structure and (**b**) C/C++ code in HLS of Block E.

In what follows, the complete updating process of the inverse matrices by using these blocks can be summarized in five stages as depicted in Figure 2.

Stage1 All elements of vector $\mathbf{x}_n^T$ read from the DDR3 SDRAM are loaded sequentially and passed on to the next stage.

Stage2 According to the index of the current pixel, we read a corresponding matrix $\mathbf{S}^{-1}$ from the storage module. When dealing with the first four pixels of an image, we need to overwrite the matrix $\mathbf{S}^{-1}$ with initialized matrix $\beta \cdot \mathbf{I}$. Then, we take matrix $\mathbf{U}$ and vector $\mathbf{p}^T$ as input operands into the Block A to calculate product $\mathbf{h}$. Subsequently, $\mathbf{p}$, $\mathbf{h}$, and $\mathbf{U}$ are passed to the next stage.

Stage3 We count T by applying the Block B, then transmit $\mathbf{U}$, T, and $\mathbf{h}$ to the next stage.

Stage4 The Block C is utilized to work out the product $\mathbf{F}$ of two vectors. We calculate Q by employing the Block D. Then $\mathbf{U}$, $\mathbf{F}$, and Q are delivered to the next stage.

Stage5 We figure out the new matrix $\mathbf{S}^{-1}$ through utilizing the Block E and write it to the corresponding location of the storage module according to the current pixel.

Besides, it is worth noting that the following design optimization strategies play an important role in improving the performance of the FPGA implementation.

(1) In the process of updating the inverse matrices, we allocate a single divider and execute it once for each inverse matrix updating. Thanks to such operation, a lot of logic resources and computation time consumed by the divider can be saved.

(2) There are three types of data that need to be cached between two stages, the scalar data, the vector data, and the matrix data. In order to attain the capability of parallel computation, the matrix is cached in L first in first out (FIFO) memories (In HLS, we use the STREAM directive to map these sorts of data into FIFOs). While the elements of a vector are realized as registers. In addition, L simple dual port RAMs (simple DPRAMs) are deployed to implement the storage module.

(3) The data type of input data is 16 bits signed fixed-point (15 bits fractional part), while the data type of intermediate data and detection results are not easy to assign. Due to the precision of intermediate data and detection results have a significant impact not only on the detection accuracy but also on the resource consumption, we performed some experiments to explore the relationship between the data precision and the detection accuracy. The experimental results demonstrate that the detection accuracy goes up with the increase of the bit-width of the fractional part. To better balance the trade-off between the detection accuracy and the resource consumption, we use different data types in different stages. As shown in Figure 2, the variable T and Q are defined as 38 bits signed fixed-point type (14 bits integer part, 23 bits fractional part). The elements of the matrix F are 32 bits signed fixed-point type (14 bits integer part, 17 bits fractional part).

All the other intermediate data are represented as 32 bits signed fixed-point type (7 bits integer part, 24 bits fractional part). T and Q have the significant impact on the detection accuracy. Therefore, they are assigned high data precision up to 38 bits. The elements of the matrix F are obtained by the accumulation operations, and more bits should be assigned to the integer part for avoiding data overflow. Though T and Q have larger bit-width up to 38 bits, it almost does not increase the logic resource consumption compared with the data type of 32 bits signed fixed-point. The reason is that only one single accumulation adder is allocated to compute T while one single adder and one single divider are placed to calculate Q. It is worthwhile to highlight that these data types can be defined and modified by HLS *ap_fixed* type easily.

4.2.2. Deep Pipeline

As shown in Figure 8, a full pipeline for updating inverse matrices is comprised of Task1, Task2, Task3, Task4, and Task5. These five tasks correspond to Stage1, Stage2, Stage3, Stage4, and Stage5 in Figure 2 respectively.

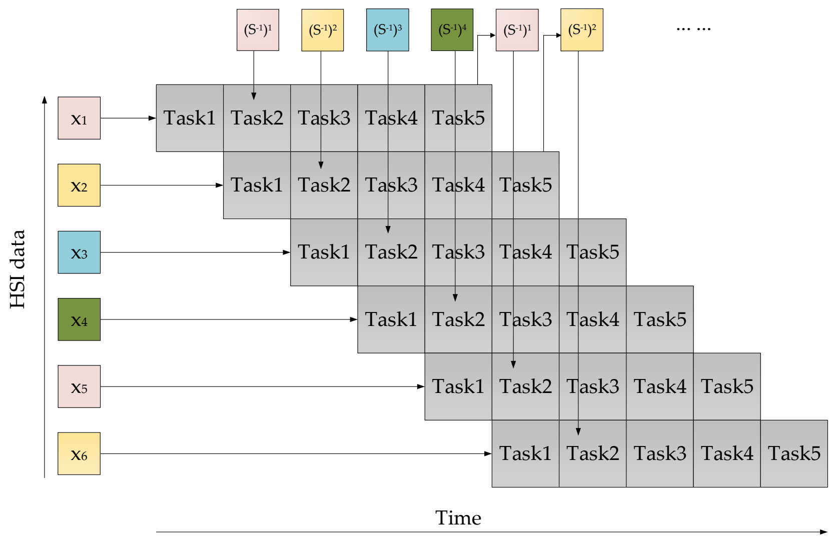


Figure 8. Timing diagram of the process of updating inverse matrix.

(1) For the purpose of reducing logic resources without compromising accuracy, we implement a high-precision division with the price of long latency. It takes near 30 clock cycles to output the division result. If the division operation is assigned to Task3, the running time of Task3 will increase a lot. As a result, Task3 will turn out to be a bottleneck in the pipeline. Therefore, we assign the division operation to Task4. Note that, the division and multiplication operations in Task4 are carried out simultaneously.

(2) For each task, it does not start until all input data are ready and all output FIFOs are not full. It can be simply realized in HLS by writing C/C++ code as shown in Figure 9. To make the pipeline run efficiently, these FIFOs, which are dedicated to bridging two adjacent tasks, are designed a little bit larger. In this work, the depth of FIFO for vector is 2, while the depth of FIFO for the matrix is $L \times 2$. Besides, the depth of simple DPRAM for the storage module is $L \times 2 \times 4$.

(3) With regard to the execution time of each task, it is consistent with $L + 12$ times of the system clock period. Among them, the input time of an L-dimensional vector is L clock cycles, the delay time of the multiplier is one clock cycle, and the remaining 11 clock cycles are used to control input/output of the task. Especially, because two extra clock cycles are required for overwriting the matrix $\mathbf{S}^{-1}$ with the initialized matrix $\beta \cdot \mathbf{I}$, the total execution time of Task2 is $L + 14$ clock cycles.

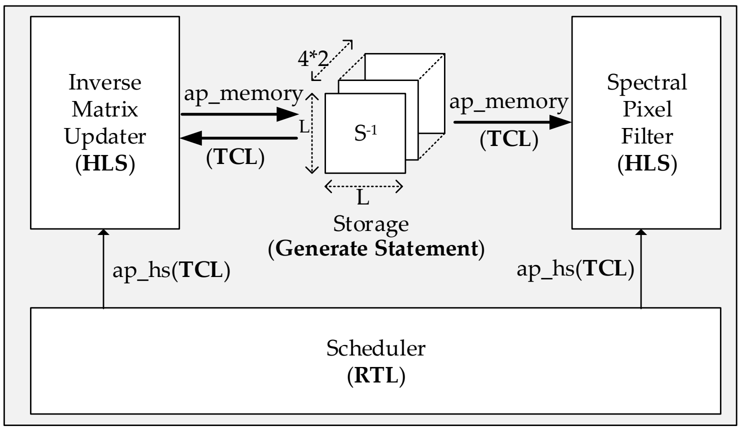
```
/****************************************************************************/
//parameter:task_cntrlfpre,task_cntrl2nxt
//implementation:The parameter task_cntrlfpre actually represents
//           a FIFO, and when the previous task is completed, the value
//           of end_flag is written to the FIFO. The value is 1 when
//           processing the last pixel of an image, 0 otherwise. The
//           parameter task_cntrl2nxt is similar to task_cntrlfpre.
/****************************************************************************/
void task(stream<bool>& task_cntrlfpre,stream<bool>& task_cntrl2nxt){
    bool end_flag;
    while(1){
        while(1)
    if((!task_cntrlfpre.empty())&&(!task_cntrl2nxt.full())) break;
        /*the calculations of this task,thus omitted*/
        ...
        task_cntrlfpre.read_nb(end_flag);
        task_cntrl2nxt.write_nb(end_flag);
        if(end_flag==1) break;
    }
}
```

Figure 9. Sample code used for implementing the data flow control of a task in HLS (The omitted lines of code are the specific computations of each stage described in Section 4.2.1).

4.3. Difficulties with Using HLS

4.3.1. Feedback

The function of task-level pipelining is available in HLS by applying DATAFLOW directive. However, one of the major difficulties with HLS is that HLS does not support to generate a task-level pipelined structure if data dependency (feedback) exists. Unfortunately, there is a feedback in the process of updating the inverse matrix as explained in Section 3.2. To solve this problem, we exploit a design method with a hybrid of RTL and HLS. As shown in Figure 10, HLS is applied to create the two complex modules, inverse matrix updater and spectral pixel filter. A small piece of RTL code is written to complete the scheduler whose function is quite simple. Verilog's generate statement is used to circularly instantiate all of the simple DPRAMs allocated in the storage module. Moreover, a TCL script for automatically connecting the above-mentioned modules is employed. When using HLS to realize the inverse matrix updater module, we define separate interface variables representing the input and output inverse matrices respectively. This separation strategy allows HLS to understand there is no data feedback in accessing the inverse matrix. In fact, the input and output inverse matrices are pointed to the same memory location in the storage module.



Inverse Matrix Updater (HLS)
ap_memory
(TCL)
4*2
L
S-1
L
Storage (Generate Statement)
ap_memory
(TCL)
Spectral Pixel Filter (HLS)
ap_hs(TCL)
ap_hs(TCL)
Scheduler (RTL)

Figure 10. Diagram of development with multiple tools.

4.3.2. High Fanout

Register duplication is one of the most common ways to solve high fanout violation. It can be applied to relieve the fanout challenge as described in Section 3.2. However, the difficulty is how to make HLS replicate registers automatically since there is no inherent support of such feature in HLS. To solve this problem, we modify part of C/C++ code in HLS to split the high fanout task into two or more identical subtasks, which allows HLS to generate duplicated circuits for reducing the fanout. With this optimization, our FPGA implementation is able to work at a rate of speed higher than 200 MHz.

4.4. Specific Features

4.4.1. Scalability and Portability

Parallel computation and memory units are placed in the stages of the core architecture of DPBS-CEM to accelerate the related operations of matrix multiplication. The number of the parallel units is equal to the value L. By modifying the value L, we can easily scale the core framework of DPBS-CEM with HLS to support different HSIs with different number of bands. Parameter customized design method with HLS greatly improves the scalability of the system. Simultaneously, the framework does not rely on any specific underlying physical devices of FPGA and vendor-provided IP cores. Thus it can be easily ported to other types of FPGAs.

4.4.2. Flexibility

The flexibility feature is referred to as multiple work modes supported by the proposed DPBS-CEM. The default work mode is high-speed, at which the pipeline is fully operating. Besides, DPBS-CEM is also allowed to be configured working at low-speed mode. Then it can produce global detection results with no need to split image data into four parts for applying deep pipeline. Furthermore, through altering the control of the pipeline between the two processes of inverse matrix updater and spectral pixel filter, DPBS-CEM can output detection results while part of the image pixels are obtained.

5. Experimental Results and Analysis

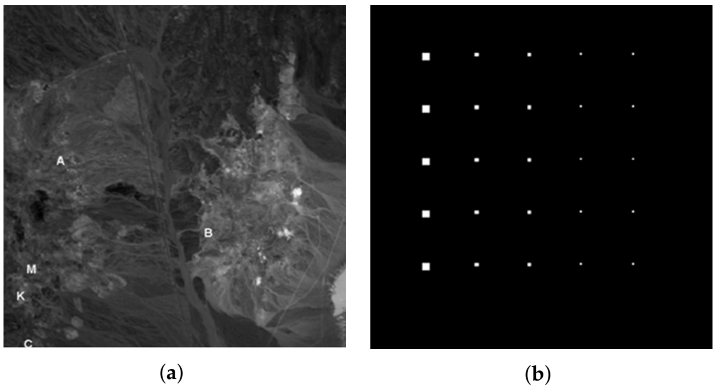
A Virtex7 FPGA board (Alpha-Data ADM-PCIE-7V3) is chosen as our development platform, which provides more logic resources than the Kintex-7 board used in [21]. Besides the FPGA implementation of DPBS-CEM, the simulation versions were also implemented using the MATLAB and C++ languages. The code of MATLAB and C++ are executed on Windows 7 operating system equipped with the Intel Core (TM) quad CPU @3.2 GHz and 4 GB main memory. We compare the performance of DPBS-CEM with SBS-CEM [21] under the same condition of FPGA implementation. The rest of this section is organized as follows. Section 5.1 describes two hyperspectral data sets used in the experiment. Section 5.2 shows the detection accuracy of the DPBS-CEM algorithm evaluated on both of the hyperspectral data sets. Section 5.3 gives a comparison of the processing time of the DPBS-CEM algorithm in MATLAB, C++ and FPGA. Finally, compared to the FPGA implementation of SBS-CEM [21], we analyze the advantages of the FPGA implementation of DPBS-CEM in terms of logic resources utilization and data processing speed.

5.1. Hyperspectral Image Data Set

5.1.1. TE1 Image

As shown in Figure 11a, 25 panels created with five United States Geological Survey (USGS, Reston, VA, USA) reflectance hyperspectral signatures: alunite (A), buddingtonite (B), calcite (C), kaolinite (K), and muscovite (M). Each row of the five panels in Figure 11b is simulated by the same mineral signature and each column of five panels has the same size [32,33]. Among 25 panels are: five 4×4-pure pixel panels, $px^i_{4\times 4}$ for $i = 1, \ldots, 5$ in the first column; five 2×2-pure pixel panels, $px^i_{2\times 2}$ for

$i = 1, \ldots, 5$ in the second column; five 2×2-mixed pixel panels, $\left\{px^i_{3,jk}\right\}^{2,2}_{j=1,k=1}$ for $i = 1, \ldots, 5$ in the third column; five subpixel panels, $px^i_{4,11}$ for $i = 1, \ldots, 5$ in the fourth column; and five subpixel panels, $px^i_{5,11}$ for $i = 1, \ldots, 5$ in the fifth column. Table 2 tabulates the mixing details of mineral composition in the 20 panels in the third column, while subpixel panels in the fourth and fifth columns are simulated with their abundance fractions tabulated in Table 3, where the background (BKG) is simulated by the sample mean of the real cuprite image scene in USGS [33]. The Synthetic image TE1 is 200×200 pixels, 189 bands from 0.4 um to 2.5 um.



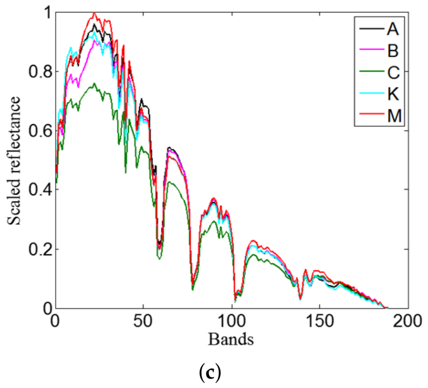


Figure 11. (**a**) Cuprite Airborne Visible/Infra Red Imaging Spectrometer (AVIRIS) image scene with spatial positions of five pure pixels corresponding to minerals: alunite (A), buddingtonite (B), calcite (C), kaolinite (K) and muscovite (M); (**b**) Synthetic image simulated by Scenario TE1; (**c**) Five reflectance USGS ground-truth mineral spectra.

Table 2. Simulated 20 mixed panel pixels in the third column.

Row1	$px^1_{3,11}$ = 0.5A + 0.5B	$px^1_{3,12}$ = 0.5A + 0.5C	$px^1_{3,21}$ = 0.5A + 0.5K	$px^1_{3,22}$ = 0.5A + 0.5M
Row2	$px^2_{3,11}$ = 0.5B + 0.5A	$px^2_{3,12}$ = 0.5B + 0.5C	$px^2_{3,21}$ = 0.5B + 0.5K	$px^2_{3,22}$ = 0.5B + 0.5M
Row3	$px^3_{3,11}$ = 0.5C + 0.5A	$px^3_{3,12}$ = 0.5B + 0.5C	$px^3_{3,21}$ = 0.5C + 0.5K	$px^3_{3,22}$ = 0.5C + 0.5M
Row4	$px^4_{3,11}$ = 0.5K + 0.5A	$px^4_{3,12}$ = 0.5K + 0.5B	$px^4_{3,21}$ = 0.5K + 0.5C	$px^4_{3,22}$ = 0.5K + 0.5M
Row5	$px^5_{3,11}$ = 0.5M + 0.5A	$px^5_{3,12}$ = 0.5M + 0.5B	$px^5_{3,21}$ = 0.5M + 0.5C	$px^5_{3,22}$ = 0.5M + 0.5K

Table 3. Abundance fractions of subpixel panels in the fourth and fifth columns.

Row	Fourth Column	Fifth Column
1	$px^1_{4,11}$ = 0.5A + 0.5BKG	$px^1_{5,11}$ = 0.25A + 0.75BKG
2	$px^2_{4,11}$ = 0.5B + 0.5BKG	$px^2_{5,11}$ = 0.25B + 0.75BKG
3	$px^3_{4,11}$ = 0.5C + 0.5BKG	$px^3_{5,11}$ = 0.25B + 0.75BKG
4	$px^4_{4,11}$ = 0.5K + 0.5BKG	$px^4_{5,11}$ = 0.25K + 0.75BKG
5	$px^5_{4,11}$ = 0.5M + 0.5BKG	$px^5_{5,11}$ = 0.25M + 0.75BKG

5.1.2. HyMap Reflectance Image

The hyperspectral data set is provided by the Digital Imaging and Remote Sensing Group, Center for Imaging Science, Rochester Institute of Technology [34]. Figure 12 shows the HyMap reflection map of Cook City, Montana, USA with a resolution of 280×800 and a total of 126 bands distributed between 0.4 and 2.4 um. There is a grass area and four real panels of fabric in the data set as shown in Table 4, where the area of interest is highlighted with a red circle.

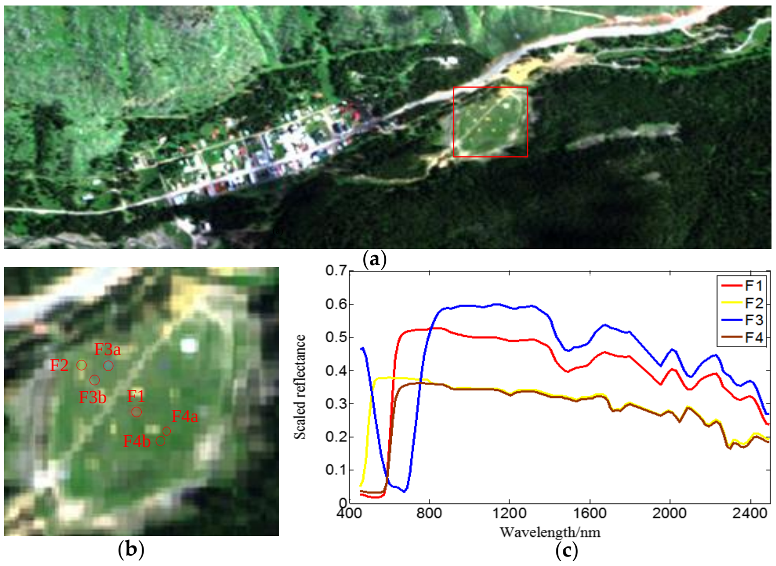


Figure 12. (**a**) HyMap reflectance image of Cook City in Montana, USA, and locations of the real targets; (**b**) Enlarged figure of red box area; (**c**) Spectral signatures of four targets.

Table 4. The characteristics of targets in the real scene of HyMap.

Name	**F1**	**F2**	**F3a**	**F3b**	**F4a**	**F4b**
Size (m)	3 × 3	3 × 3	2 × 2	1 × 1	2 × 2	1 × 1
Fabric type	Red cotton	Yellow nylon	Blue cotton	Blue cotton	Red nylon	Red nylon

5.2. Analysis of Target Detection Accuracy

In this part, we evaluate the detection accuracy of the FPGA implementation of DPBS-CEM by using the simulation/real HSI data sets described above. CEM and SBS-CEM are evaluated as well for comparison. The detection accuracy can be evaluated via Receiver Operating Characteristics (ROC) [35]. However, the ROC curves of different algorithms may be too close to determine which algorithm has better performance. Therefore, in this paper, we choose another way commonly used in medical diagnosis to calculate the area under a ROC curve, referred to as the area under the curve (AUC) [36]. The AUC values corresponding to the detection results can further quantify the differences in the accuracy of the algorithms. The higher the AUC, the better the detection accuracy.

5.2.1. Detection Accuracy of TE1

Figure 13 shows five detection maps produced by DPBS-CEM using the five-panel signatures A, B, C, K, and M in Figure 11c as the desired target signatures. The two-dimensional (2-D) results of real-time detection of target A illustrated in Figure 14. The experimental results show that all the AUC values of five desired targets detected by DPBS-CEM are one, indicating that the detection results are extremely satisfactory.

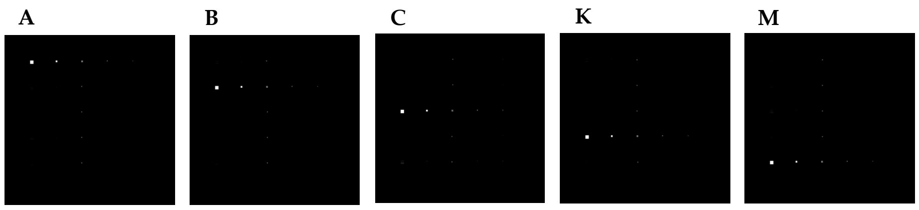


Figure 13. Detection maps of DPBS-CEM using A, B, C, K and M as desired target signature.

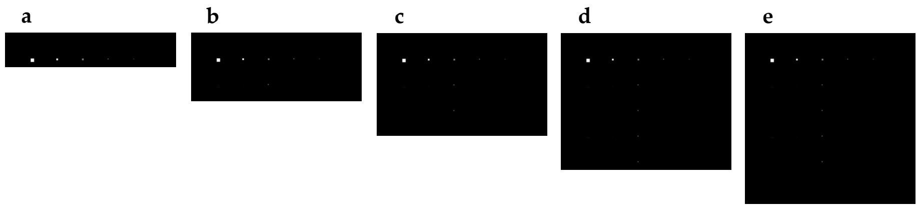


Figure 14. Real-time detection results with A used as desired target signature.

5.2.2. Detection Accuracy of HyMap

In order to further measure the performance of DPBS-CEM, we also focus on the detection results of HyMap data set. Figure 15 shows the results of the target F4 obtained by Global-CEM, SBS-CEM, and DPBS-CEM, respectively. For a more accurate representation of the detection results, we have an enlarged target region of interest, as shown in red boxes of target F4 and Figure 16 of target F1, F2, and F3. As we expected, in comparison to the target detection accuracy of SBS-CEM, DPBS-CEM has the same or even better performance. This conclusion is further verified by the AUC values in Table 5.

Table 5. AUC obtained by different algorithms for the targets.

	F1	**F2**	**F3**	**F4**
Global-CEM	0.9107	1	0.9067	0.9987
SBS-CEM [18]	0.9783	1	0.9862	0.9972
DPBS-CEM	0.9997	0.9999	0.9992	0.9994

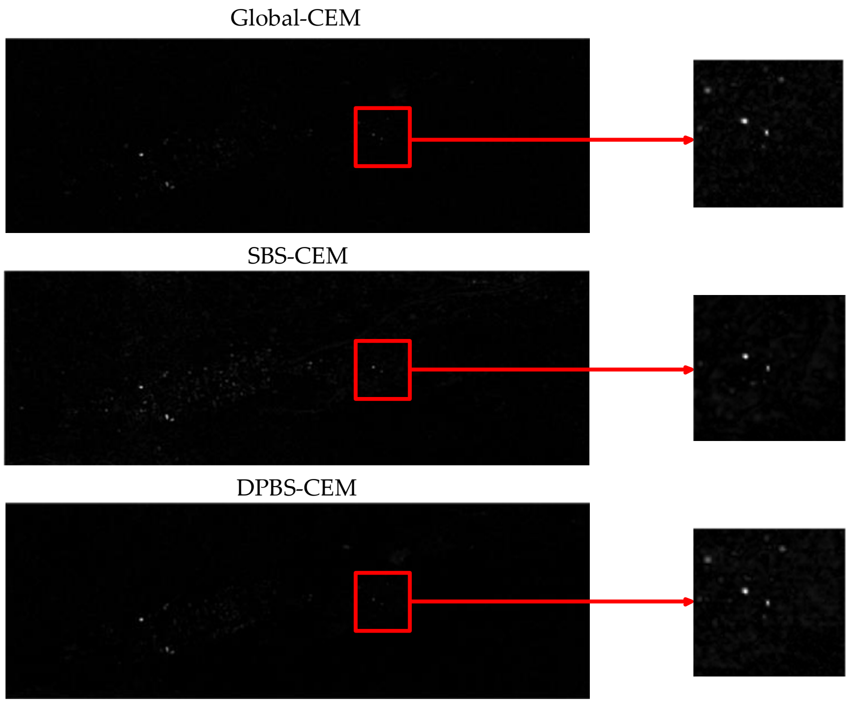


Figure 15. Detection results for target F4 obtained by different algorithms.

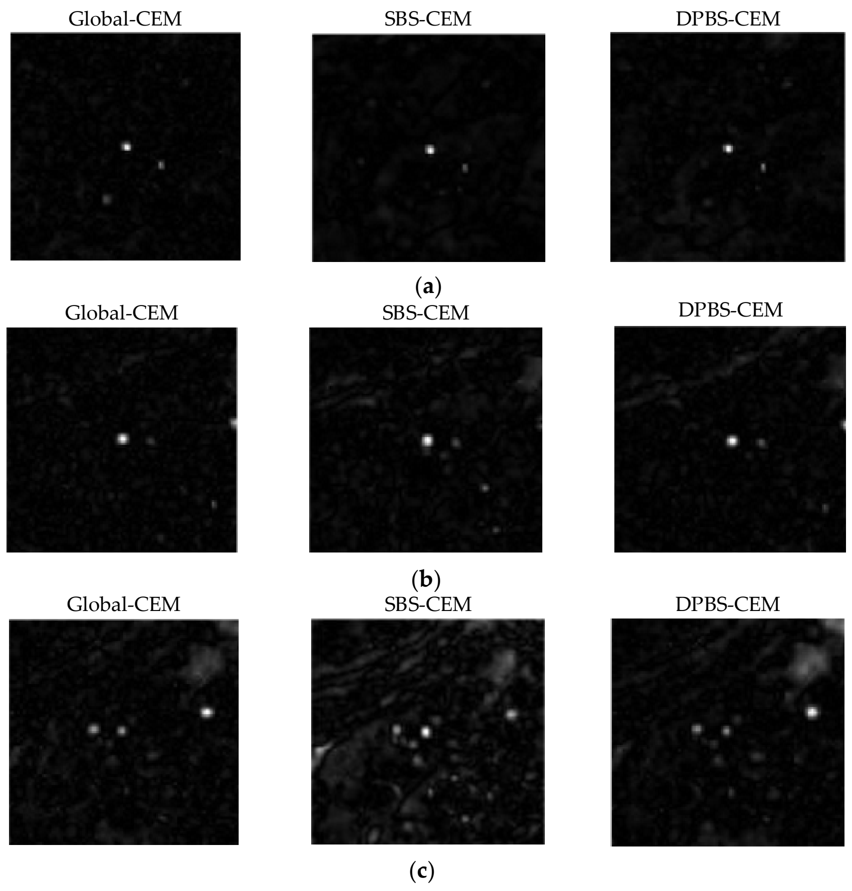


Figure 16. Detection results obtained by different algorithms for targets: (**a**) F1; (**b**) F2; and (**c**) F3.

5.3. Cross-Platform Performance Comparison

From the previous section, we can see that the proposed DPBS-CEM is very close to SBS-CEM [18] in detection accuracy, some detection results of DPBS-CEM are even superior to the latter one. Table 6 shows the processing time comparison of the proposed DPBS-CEM on different platforms (such as MATLAB, C++, and FPGA). The version of MATLAB used here is R2014a. The C++ environment directly uses the software simulation environment of Vivado HLS 2017.3. The clock frequency of FPGA is set at 200 MHz. As shown in Table 6, the processing time of DPBS-CEM implemented on FPGA has achieved significant improvements compared to MATLAB and C++ implementations. On the other hand, the processing time of our software versions is also superior to that of SBS-CEM software implementations [21] since the proposed DPBS-CEM algorithm is less computationally expensive than the SBS-CEM algorithm.

Table 6. Processing time measured for DPBS-CEM methods in MATLAB, C++, and FPGA implementations.

Platform	MATLAB (s)	C++ (s)	FPGA (s)
HyMap	60.7378	58.135	0.1568

5.4. Performance Comparison between DPBS-CEM and SBS-CEM

The FPGA design of DPBS-CEM is implemented on a Virtex7 XC7VX690T FPGA. This FPGA contains 108,300 slices, 433,200 six-input LUTs, 1470 BRAMs, and 3600 DSPs. To facilitate the performance comparison between DPBS-CEM and SBS-CEM, we selected HyMap, the same hyperspectral data source used by SBS-CEM, as our input HSI. Next, we compare the FPGA implementations of SBS-CEM and DPBS-CEM from two aspects of logic resources utilization and data processing speed.

Table 7 shows the resource utilization corresponding to SBS-CEM and DPBS-CEM. The right-hand side lists the unit's ratios and average ratios of DPBS-CEM and SBS-CEM. As Table 7 illustrates, the average resource utilization of DPBS-CEM is 5.21 times more than that of the SBS-CEM algorithm, which is caused by the deep pipelined structure. As aforementioned in Section 4.2.1, the intermediate data precision has a dramatic impact on the detection accuracy. Table 8 shows the relationship between the detection accuracy represented by AUC and the intermediate data precision. In Table 8, we set the data precision as fixed-point type with total of 32, 34, 36, 38, 40, and 42 bits, and identical 14 bits integer part. The experimental results demonstrate that the detection accuracy goes up sharply with the increase of data precision from 32 to 38 while keeps the same from 38 to 42. Due to the same bit-width of the integer part, it can be concluded that the bit-width of the fractional part mainly determines the detection accuracy. According to the experimental results, the bit-width of the fractional part should be more than 23.

The performance of DPBS-CEM has been greatly improved compared with SBS-CEM. Table 9 shows the number of clock cycles occupied by SBS-CEM and DPBS-CEM and the ratio between them. At the same clock frequency of 200 MHz, the number of clock cycles consumed by SBS-CEM is nearly 7.3 times more than that of DPBS-CEM. In other words, when processing the same image, the data processing speed of DPBS-CEM is 7.3 times faster than that of SBS-CEM. It is worthwhile to mention that our work is conducted by mainly using HLS.

Table 7. Comparison of resource utilization for the FPGA implementations of SBS-CEM and DPBS-CEM.

	SBS-CEM Units (G)	DPBS-CEM Units (Z)	Ratio $\left(\frac{Z}{G}\right)$
Number of DSP48Es	265	1396	5.27
Number of Block RAM	120	379	3.16
Number of Slices	12,088	58,167	4.81
Number of Flip Flops	28,245	217,958	7.72
Number of LUTs	21,730	111,073	5.11
Average Ratio	–	–	5.21

Table 8. Corresponding AUC values with different intermediate data accuracy of algorithm (we set F1 in HyMap image as the desired target).

Precision (Bit)	32	34	36	38	40	48
AUC	0.2530	0.4779	0.5463	0.9997	0.9997	0.9997

Table 9. Comparison of data processing speed for the FPGA implementations of SBS-CEM and DPBS-CEM.

	SBS-CEM	DPBS-CEM	Speedup
Frequency (MHz)	200	200	7.3×
Number of clock periods	229,607,996	31,360,557	

6. Discussion

CEM is an effective algorithm for subpixel target detection in hyperspectral imagery. The classical CEM needs to solve a large matrix inversion problem. SBS-CEM takes the Sherman-Morrison formula to update the inverse matrix for each pixel, which can avoid the complex calculation of large matrix inversion. However, SBS-CEM still uses sliding windows and has data dependency problems, which prevents its further performance improvement on target detection in terms of processing speed.

To solve these problems, we proposed an optimized algorithm called DPBS-CEM. It follows the same way that is used to update the inverse matrix gradually according to the Sherman-Morrison formula [28] but uses cumulative windows instead of sliding windows to reduce the number of calculations. Pixel data splitting and separating inverse matrix memories are utilized to remove the data dependency existing in the process of updating the inverse matrix. Moreover, we provide an FPGA implementation of the proposed DPBS-CEM whose deep pipelined architecture can be realized by using HLS.

According to the experimental results presented in this paper, the target detection accuracy of the proposed DPBS-CEM algorithm on two data sets are nearly the same. Compared to SBS-CEM, it has the same or even better detection accuracy. Regarding the processing speed performance, DPBS-CEM gained about 7.3 times speedup than that of SBS-CEM. It is worth noting that the proposed architecture of DPBS-CEM can also gain benefits in terms of scalability, portability, and flexibility with the help of HLS. This is particularly suitable for the real-time hyperspectral target detection applications on satellite.

7. Conclusions

In this paper, an optimized algorithm , referred to as DPBS-CEM for hyperspectral target detection, is proposed. A deep pipelined architecture of DPBS-CEM on FPGA is developed by using HLS as well. The experimental results show that the proposed FPGA implementation of DPBS-CEM has an extraordinary performance improvement in terms of data throughput without compromising for detection accuracy. Under the same test conditions, the detection speed of our proposed DPBS-CEM is about 7.3 times faster than that of SBS-CEM.

Acknowledgments: This work was partially supported by the National Natural Science Foundation of China (Nos. 61571345, 91538101, 61501346, 61502367, and 61701360) and the 111 project (B08038).

Author Contributions: Jie Lei and Yunsong Li conceived and designed the experiments; Dongsheng Zhao and Jing Xie performed the experiments; Jie Lei and Jing Xie analyzed the result data; Chein-I Chang provided suggestions on algorithm optimization and paper revision; Lingyun Wu, Xuepeng Li, Jintao Zhang, and Wenguang Li contributed reagents/materials/analysis tools; Jie Lei wrote the paper.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

HSI	Hyperspectral image
FPGA	Field programmable gate array
CEM	Constrained energy minimization
HLS	High-level synthesis

Appendix A

Table A1. Vivado HLS optimization pragmas.

Directive	Description
#pragma HLS INTERFACE	Specifies how RTL ports are created from the function description.
#pragma HLS PIPELINE	Reduces the initiation interval by allowing the concurrent execution of operations within a loop or function.
#pragma HLS ARRAY_PARTITION	Partitions large arrays into multiple smaller arrays or into individual registers, to improve access to data and remove block RAM bottlenecks.
#pragma HLS UNROLL	Unroll for-loops to create multiple independent operations rather than a single collection of operations.
#pragma HLS DATAFLOW	Enable task level pipelining, allowing functions and loops to execute concurrently. Used to minimize interval.

References

1. Chang, C.I. *Hyperspectral Imaging: Spectral Techniques for Detection and Classification*; Kluwer Academic Publishers: Norwell, MA, USA, 2003.
2. Ryan, J.P.; Davis, C.O.; Tufillaro, N.B.; Kudela, R.M.; Gao, B.C. Application of the hyperspectral imager for the coastal ocean to phytoplankton ecology studies in Monterey Bay, CA, USA. *Remote Sens.* **2014**, *6*, 1007–1025.
3. Dale, L.M.; Thewis, A.; Boudry, C.; Rotar, I.; Dardenne, P.; Baeten, V.; Pierna, J.A.F. Hyperspectral imaging applications in agriculture and agro-food product quality and safety control: A review. *Appl. Spectrosc. Rev.* **2013**, *48*, 142–159.
4. Zhang, B.; Wu, D.; Zhang, L.; Jiao, Q.; Li, Q. Application of hyperspectral remote sensing for environment monitoring in mining areas. *Environ. Earth Sci.* **2012**, *65*, 649–658.
5. Cloutis, E.A. Review article hyperspectral geological remote sensing: Evaluation of analytical techniques. *Int. J. Remote Sens.* **1996**, *17*, 2215–2242.
6. Chang, C.I. Real-Time recursive hyperspectral sample and band processing: Algorithm architecture and implementation. In *Real-Time Recursive Hyperspectral Sample and Band Processing,* 1st ed.; Springer: Berlin, Germany, 2017; pp. 123–156, ISBN 978-3-319-45170-1.
7. Wang, Y.; Huang, S.; Liu, D.; Wang, H. A target detection method for hyperspectral imagery based on two-time detection. *J. Indian Soc. Remote Sens.* **2017**, *45*, 239–246.
8. Zou, Z.; Shi, Z. Hierarchical suppression method for hyperspectral target detection. *IEEE Trans. Geosci. Remote Sens.* **2016**, *54*, 330–342.
9. He, C.; Zhao, Y.; Tian, J.; Shi, P.; Huang, Q. Improving change vector analysis by cross-correlogram spectral matching for accurate detection of land-cover conversion. *Int. J. Remote Sens.* **2013**, *34*, 1127–1145.
10. Chaudhuri, S.; Chatterjee, S.; Katz, N.; Nelson, M.; Goldbaum, M. Detection of blood vessels in retinal images using two-dimensional matched filters. *IEEE Trans. Med. Imaging* **1989**, *8*, 263–269.
11. Yuhas, R.H.; Goetz, A.F.; Boardman, J.W. Discrimination among semi-arid landscape endmembers using the spectral angle mapper (SAM) algorithm. In *JPL, Summaries of the Third Annual JPL Airborne Geoscience Workshop*; NASA: Washington, DC, USA, 1992; pp.147–149.
12. Du, Q.; Ren, H.; Chang, C.I. A comparative study for orthogonal subspace projection and constrained energy minimization. *IEEE Trans. Geosci. Remote Sens.* **2003**, *41*, 1525–1529.
13. Ren, H.; Chang, C.I. A target-constrained interference-minimized filter for subpixel target detection in hyperspectral imagery. In Proceedings of the Geoscience and Remote Sensing Symposium, Honolulu, HI, USA, 24–28 July 2000; pp. 1545–1547.
14. Manolakis, D.; Marden, D.; Shaw, G.A. Hyperspectral image processing for automatic target detection applications. *J. Lincoln Lab.* **2003**, *14*, 79–116.
15. Scharf, L.L.; Friedlander, B. Matched subspace detectors. *IEEE Trans. Signal Process.* **1994**, *42*, 2146–2157.
16. Harsanyi, J.C.; Chang, C.I. Hyperspectral image classification and dimensionality reduction: An orthogonal subspace projection approach. *IEEE Trans. Geosci. Remote Sens.* **1994**, *32*, 779–785.
17. Chen, Y.; Nasrabadi, N.M.; Tran, T.D. Sparse representation for target detection in hyperspectral imagery. *IEEE J. Sel. Top. Signal Process.* **2011**, *5*, 629–640.
18. Mittal, S.; Vetter, J.S. A survey of CPU-GPU heterogeneous computing techniques. *ACM Comput. Surv.* **2015**, *47*, 69.
19. Plaza, A.; Chang, C.I. Clusters Versus FPGA for parallel processing of hyperspectral imagery. *Int. J. High Perform. Comput. Appl.* **2008**, *22*, 366–385.
20. Wang, J.; Chang, C.; Cao, M. FPGA design for constrained energy minimization. *Proc. SPIE* **2004**, 262–273, doi:10.1117/12.518559.
21. Yang, B.; Yang, M.; Plaza, A.; Gao, L.; Zhang, B. Dual-mode FPGA implementation of target and anomaly detection algorithms for real-time hyperspectral imaging. *IEEE J.-STARS* **2015**, *8*, 2950–2961.
22. Gonzalez, C.; Bernabe, S.; Mozos, D.; Plaza, A. FPGA implementation of an algorithm for automatically detecting targets in remotely sensed hyperspectral images. *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.* **2016**, *9*, 4334–4343.
23. Santos, L.; López, J.F.; Sarmiento, R.; Vitulli, R. FPGA implementation of a lossy compression algorithm for hyperspectral images with a high-level synthesis tool. In Proceedings of the 2013 NASA/ESA Conference on Adaptive Hardware and Systems, Torino, Italy, 24–27 June 2013; pp. 107–114.

24. García, A.; Santos, L.; López, S.; Callicó, G.M.; Lopez, J.F.; Sarmiento, R. Efficient lossy compression implementations of hyperspectral images: Tools, hardware platforms, and comparisons. In *Satellite Data Compression, Communications, and Processing X*; International Society for Optics and Photonics: Bellingham, WA, USA, 2014.
25. Domingo, R.; Salvador, R.; Fabelo, H. High-level design using Intel FPGA OpenCL: A hyperspectral imaging spatial-spectral classifier. In Proceedings of the 2017 IEEE 12th International Symposium on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC), Madrid, Spain, 12–14 July 2017; pp. 1–8.
26. Del Sozzo, E.; Solazzo, A.; Miele, A.; Santambrogio, M.D. On the automation of high level synthesis of convolutional neural networks. In Proceedings of the 2016 IEEE International Symposium on Parallel and Distributed Processing, Chicago, IL, USA, 23–27 May 2016; pp. 217–224.
27. Guan, Y.; Liang, H.; Xu, N.; Wang, W.; Shi, S.; Chen, X. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In Proceedings of the 2017 IEEE International Symposium on Field-Programmable Custom Computing Machines, Napa, CA, USA, 30 April–2 May 2017; pp. 152–159.
28. Hager, W.W. Updating the inverse of a matrix. *Siam Rev.* **1989**, *31*, 221–239.
29. Chang, C.I.; Li, H.C.; Song, M.; Liu, C.; Zhang, L. Real-time constrained energy minimization for sub pixel detection. *IEEE J.-STARS* **2015**, *8*, 2545–2559.
30. Chang, C.I. Real-Time Recursive Hyperspectral Sample Processing for Active Target Detection: Constrained Energy Minimization. In *Real-Time Recursive Hyperspectral Sample and Band Processing,* 1st ed.; Springer: Berlin, Germany, 2017; pp. 123–156, ISBN 978-3-319-45170-1.
31. Nasrabadi, N.M. Regularized spectral matched filter for target recognition in hyperspectral imagery. *IEEE Signal. Proc. Lett.* **2008**, *15*, 317–320.
32. Wang, J.; Chang, C.I. Applications of independent component analysis in endmember extraction and abundance quantification for hyperspectral imagery. *IEEE Trans. Geosci. Remote Sens.* **2006**, *44*, 2601–2616.
33. Chang, C.I. Design of Synthetic Image Experiments. In *Hyperspectral Data Processing: Algorithm Design and Analysis,* 1st ed.; John Wiley & Sons: Hoboken, NJ, USA, 2013; pp. 103–113, ISBN 978-0-471-69056-6.
34. Snyder, D.; Kerekes, J.; Fairweather, I.; Crabtree, R.; Shive, J.; Hager, S. Development of a web-based application to evaluate target finding algorithms. In Proceedings of the Geoscience and Remote Sensing Symposium, Boston, MA, USA, 6–11 July 2008; pp. II-915–II-918.
35. Parker, D.R.; Gustafson, S.C.; Ross, T.D. Receiver operating characteristic and confidence error metrics for assessing the performance of automatic target recognition systems. *Opt. Eng.* **2005**, *44*, 097202.
36. Metz, C.E. Basic principles of ROC analysis. *Semin. Nucl. Med.* **1978**, *8*, 283–298.