remote sensing

MDPI

Article

GPU-Based Lossless Compression of Aurora Spectral Data using Online DPCM

Jiaojiao Li [1], Jiaji Wu [1,*] and Gwanggil Jeon [1,2]

1 School of Electronic Engineering, Xidian University, Xi'an 710071, China
2 Department of Embedded Systems Engineering, Incheon National University, Incheon 22012, Korea
* Correspondence: wujj@mail.xidian.edu.cn

Received: 25 May 2019; Accepted: 7 July 2019; Published: 10 July 2019

Abstract: It is well known that aurorae have very high research value, but the data volume of aurora spectral data is very large, which brings great challenges to storage and transmission. To alleviate this problem, compression of aurora spectral data is indispensable. This paper presents a parallel Compute Unified Device Architecture (CUDA) implementation of the prediction-based online Differential Pulse Code Modulation (DPCM) method for the lossless compression of the aurora spectral data. Two improvements are proposed to improve the compression performance of the online DPCM method. One is on the computing of the prediction coefficients, and the other is on the encoding of the residual. In the CUDA implementation, we proposed a decomposition method for the matrix multiplication to avoid redundant data accesses and calculations. In addition, the CUDA implementation is optimized with a multi-stream technique and multi-graphics processing unit (GPU) technique, respectively. Finally, the average compression time of an aurora spectral image reaches about 0.06 s, which is much less than the 15 s aurora spectral data acquisition time interval and can save a lot of time for transmission and other subsequent tasks.

Keywords: aurora spectral data; online DPCM; CUDA; shared memory and registers; multi-stream technique; multi-GPU technique

1. Introduction

Aurorae are considered some of the most beautiful wonders in nature, which are colorful and constantly changing. When the high-energy charged particles in space are carried by the solar wind to the earth, they are attracted by geomagnetic fields to the poles of the earth, colliding with the molecules and atoms in the upper atmosphere, and finally resulting in an aurora [1]. Aurorae have very significant research value. They are helpful for the research of solar activities and their effects on the earth. Moreover, studying aurorae can provide a reference for studying other planets, which is because aurorae are a common phenomenon on planets with magnetic fields in the solar system.

Aurora spectral data is captured by the spectrometers mounted in the polar regions. As the exact occurrence time of an aurora is unknown, a spectrometer is required to capture aurora spectral data continuously at a very high frequency [2]. As a result, the amount of aurora spectral data is very large, which brings a huge challenge to storage and transmission. To address this problem, compression of aurora spectral data is indispensable, and lossless compression is preferred to lossy compression. There are two main reasons for this: one is that aurora spectral images have very high scientific value and their acquisition is quite expensive, so aurora spectral images have long-term preservation value; the other is that minor information loss may cause large errors in some applications [2], therefore, no information loss is allowed in aurora spectral image compression.

In this paper, the test dataset is the aurora spectral data collected at the Zhongshan Station in Antarctica in 2014. The temporal resolution of the camera is 15 s, i.e., an aurora spectral image is produced every 15 s. Each aurora spectral image is stored in Flexible Image Transport System (FITS) format [3] and is 1024 × 1024 in size. Each pixel is stored as a 16-bit unsigned integer. Figure 1a is an aurora spectral image captured at the Zhongshan Station on July 15, 2014, at 13:29:40, and is named as 20140715_132940. Figure 1b is an aurora spectral image captured on September 23, 2014, at 14:43:20, and is named as 20140923_144320. All the aurora spectral images referred to in the following contents will adopt this naming method. Since an aurora spectral image is produced every 15 s, the compression time must be within 15 s. Our scheme compresses the aurora spectral data using the online Differential Pulse Code Modulation (DPCM) method, and accelerate the algorithm on graphics processing units (GPUs) using Compute Unified Device Architecture (CUDA).

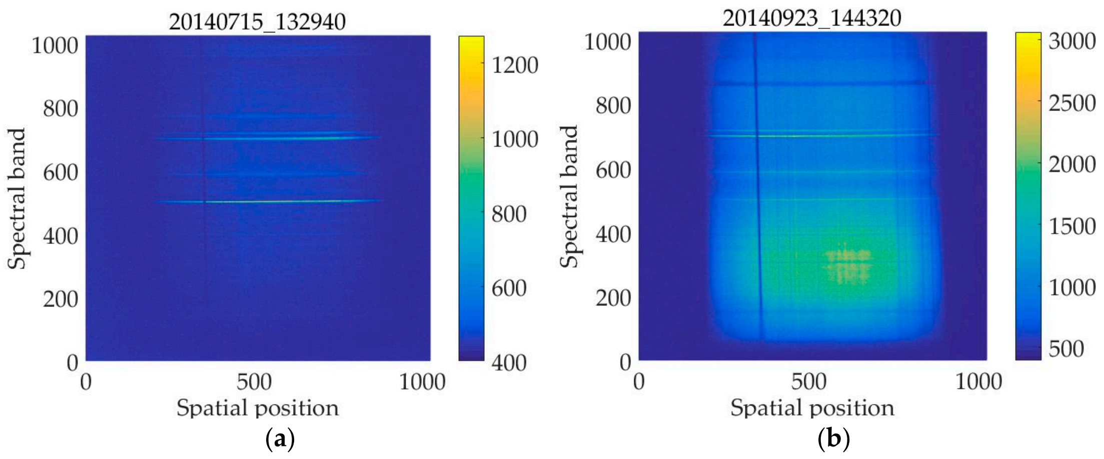


Figure 1. Two aurora spectral images captured at the Zhongshan Station. Each aurora spectral image has 1024 spectral bands and 1024 spatial sampling points. The color bar on the right is used to indicate the value of each color.

The idea of DPCM is to decorrelate data with linear regression prediction. Aiazzi et al. proposed two different linear regression prediction-based near-lossless and lossless still image compression algorithms. One is denoted as Relaxation-Labelled Prediction (RLP) [4], which partitions an image into small blocks and then calculates a linear predictor for each block. The other is named Fuzzy logic-based Matching Pursuit (FMP) [5], which computes the linear regression predictor through fuzzy-logic techniques. In addition, Aiazzi et al. proposed a context-based near-lossless or lossless coding method [6] for any causal DPCM scheme, which partitions prediction errors into homogeneous classes before arithmetic coding. Then, Mielikainen and Toivanen proposed a method named Clustered DPCM (C-DPCM) [7] for the lossless compression of hyperspectral images. The C-DPCM algorithm first clusters all of the spectral lines in a hyperspectral image into several classes, then computes the prediction coefficients of each spectral band of each class using the least square method (LSM), then calculates the residuals, and finally encodes the prediction coefficients and the residuals using the range coder [8]. The range coder is similar to the arithmetic coder, but is almost twice as fast as arithmetic coder.

Since both hyperspectral and aurora spectral images are spectral data, we apply the idea of DPCM to the lossless compression of aurora spectral images as well. However, an aurora spectral image has 1024 spectral bands, which is much more than the 220 or 224 spectral bands of the hyperspectral images introduced in a previous study [7], thus the total bitrate will be very large if the prediction coefficients of each band are encoded. To tackle this problem, the idea of online compression [9] is adopted in this paper. The main idea of online compression is to predict the current pixel using the already coded pixels, so that the decoder can calculate the prediction coefficients of the current pixel using the already decoded pixels, and there is no need to encode the prediction coefficients and transmit them to the decoder. Consequently, when using the online DPCM for the compression of the aurora spectral data, the only data that needs to be encoded and transmitted to the decoder is the residual. In addition, we proposed two improvements to improve the compression performance of the original online DPCM method [9]: one is on the establishment of the linear system of equations when computing the prediction coefficients, and the other is on the encoding of the residual.

In addition to DPCM, there are many other lossless compression algorithms, such as JPEG-LS (JPEG is the acronym for Joint Photographic Experts Group, and the JPEG-LS is the new standard for lossless and near-lossless compression of continuous-tone images), JPEG2000 (JPEG2000 is the successor of the JPEG with superior compression performance), and CALIC (Context-based, Adaptive, Lossless Image Codec). The core algorithm of JPEG-LS is LOw COmplexity LOssless COmpression for Images (LOCO-I) [10,11], in which a fixed predictor is used to detect vertical or horizontal edges, and the predicted value of a pixel can be seen as the median of the three candidate predicted values. JPEG2000 [12] is a wavelet-based still image compression standard, which can achieve both lossy and lossless compression within a common framework. The CALIC [13] uses a large number of modeling contexts to condition a nonlinear predictor, and the nonlinear predictor can adapt itself by learning from its past errors in a given context. However, JPEG-LS, JPEG2000, and CALIC were developed for common images, rather than spectral data.

Besides the two-dimensional (2-D) compression methods introduced above, there are also some three-dimensional (3-D) methods. Kim and Pearlman proposed the 3D-SPIHT (Three-Dimensional Set Partitioning In Hierarchical Trees) [14] for lossy video coding. Lucas et al. [15] proposed a method called 3D-MRP (Three-Dimensional Minimum Rate Predictors) for the lossless compression of medical images. Xiaolin and Memon extended the original CALIC to 3-D [16] for the lossless compression of multispectral images. Since hyperspectral images are usually very large and have high application value, many researchers have devoted their efforts to the compression of hyperspectral images using 3-D methods [17–20]. In fact, aurora spectral images can also be considered as 3-D images if the time dimension is added, and the 3-D methods can also be applied to the compression of aurora spectral images. However, the Antarctic Zhongshan Station is far from our country China and the transmission link is very limited [9], and larger files are more likely to cause transmission errors, therefore, aurora spectral images are usually compressed and transmitted frame by frame [2].

To speed up the online DPCM algorithm, this paper will take advantage of the powerful computing capability of the GPU. Traditionally, the GPU is only responsible for graphic rendering, and most of the processing is done by the central processing unit (CPU). Driven by the demand of the video game market and military scene simulation, GPU performance has rapidly improved, and now GPUs have developed into General Purpose GPUs (GPGPU). This also benefited from the development of CUDA, which is a general-purpose parallel computing platform and programming model, and allows programmers to program with the high-level programming language C. GPUs play a significant role in the fields of finance, petroleum, astronomy, signal processing, image processing, video compression, and others.

GPUs are specialized for high arithmetic intensity, high parallelism tasks. In our online DPCM aurora spectral data compression scheme, the computing of the predicted value of each pixel is independent, so that they can be calculated in parallel using GPUs. Furthermore, the prediction coefficients for each pixel are calculated using the least square method (LSM), and the main operations of LSM are matrix multiplication and matrix inversion, both of which are well-suited for acceleration with GPUs. Therefore, it is reasonable and suitable to accelerate the online DPCM algorithm in parallel with GPUs. Many researchers have utilized GPUs to speed up their algorithms. Simek and Asn [21] and Tenllado et al. [22] introduced parallel implementations of the 2-D Discrete Wavelet Transform (2D-DWT) on GPUs. The wavelet-based still image compression standard JPEG2000 was implemented on GPUs in a previous study [23]. Santos et al. [24] and Keymeulen et al. [25] used GPUs to speed up lossy and a lossless hyperspectral image compression algorithms, respectively. Wu et al. [26] implemented a GPU-based spatially adaptive hyperspectral image classification method.

The rest of the paper is structured as follows. Section 2 introduces the online DPCM method for aurora spectral data compression. Section 3 presents the detailed CUDA implementation of the online DPCM method. Section 4 analyzes the performance of the serial and the CUDA parallel implementations of the online DPCM algorithm, and optimizes the CUDA implementation using multi-stream and multi-GPU techniques, respectively. Finally, Section 5 summarizes this paper.

2. The Improved Online DPCM Method for Aurora Spectral Data Compression

2.1. Overview of the Online DPCM Method

The main steps of the online DPCM algorithm for aurora spectral image compression are as follows:

- Compute the prediction coefficients for each pixel;
- Calculate the prediction image and its difference from the original image, i.e., the residual;
- Encode the residual.

The prediction coefficients are calculated using the least square method (LSM). Figure 2 shows a pixel $x_{m,n}$ and the area used to predict it, and Equation (1) is the linear system of equations built to compute the prediction coefficients for $x_{m,n}$. N is the prediction order, which defines how many previous columns were used to compute the prediction coefficients. The $x_{i,j}$ $(i = 0, 1, \cdots, m-1, j = n-N, n-N+1, \cdots, n)$ is the pixel at the ith row and the jth column, and $\begin{bmatrix} \alpha_0 & \alpha_1 & \cdots & \alpha_N \end{bmatrix}^T$ is the prediction coefficient vector. It can be seen that the linear system of the equations involves all the pixels in area $A_1(x_{m,n})$, but does not involve the $x_{m,n}$ itself. Therefore, the decoder can also compute the prediction coefficients of $x_{m,n}$ using the pixels in $A_1(x_{m,n})$, and there is no need to transmit the prediction coefficients to the decoder. This is the idea of online compression.

$$\begin{bmatrix} x_{0,n-N} & x_{0,n-N+1} & \cdots & x_{0,n-1} \\ x_{1,n-N} & x_{1,n-N+1} & \cdots & x_{1,n-1} \\ \vdots & \vdots & \cdots & \vdots \\ x_{m-1,n-N} & x_{m-1,n-N+1} & \cdots & x_{m-1,n-1} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_N \end{bmatrix} = \begin{bmatrix} x_{0,n} \\ x_{1,n} \\ \vdots \\ x_{m-1,n} \end{bmatrix} \tag{1}$$

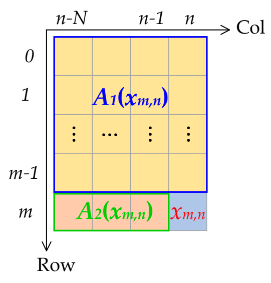


Figure 2. A small part of an aurora image that is used for the prediction of a certain pixel $x_{m,n}$.

Then, if we denote the first matrix in Equation (1) as C, denote the vector $\begin{bmatrix} \alpha_0 & \alpha_1 & \cdots & \alpha_N \end{bmatrix}^T$ as $\vec{\alpha}$, and denote the vector on the right of the equal sign as $\vec{b}$, we get the condensed format of Equation (1), that is, $C\vec{\alpha} = \vec{b}$, and finally the coefficient vector $\vec{\alpha}$ can be computed using Equation (2) [2].

$$\vec{\alpha} = (C^T C)^{-1} C^T \vec{b} \tag{2}$$

From Equation (2), the calculation of the prediction coefficients requires one matrix multiplication, one matrix inversion, and two matrix-vector multiplications $((C^T C)^{-1}(C^T \vec{b}))$, and the calculation of the prediction coefficients is actually the most time-consuming part in DPCM. The computation of the predicted value of a pixel is much simpler, for example, the predicted value of the pixel $x_{m,n}$ in Figure 2 is the linear combination of the pixels in the area $A_2(x_{m,n})$. We use Equation (3) to describe this; $\hat{x}_{m,n}$ is the predicted value of $x_{m,n}$.

$$\hat{x}_{m,n} = \sum_{i=0}^{N-1} \alpha_{i+1} x_{m,n-N+i} \tag{3}$$

However, we cannot compute the predicted values for some edge pixels of an aurora image using Equation (3), such as the three regions A_1, A_2, and A_3 shown in Figure 3, which is because we cannot build a linear system of equations, such as Equation (1), to compute the prediction coefficients for these edge pixels. In Figure 3, the H is the height of an aurora image, and W is the width, A_1 is the row 0, and A_2 is the column 0. Since there is no row before row 0 and no column before column 0, A_1 and A_2 cannot be predicted using LSM. A_3 is column 1, since there is only one column before column 1; in other words, there is only one item on the left of the equal sign if we build a linear system of equations, and we will not predict A_3 using the LSM. Equation (4) shows the prediction method for each area. The $\hat{x}_{m,n}$ is the predicted value of $x_{m,n}$. In A_1 and A_3, the predicted value of each pixel is the value of the adjacent pixel in the column direction, except the pixel $x_{0,0}$. In A_2, the predicted value of each pixel is the value of the adjacent pixel in the row direction.

$$\begin{aligned} A_1 &: \hat{x}_{0,0} = x_{0,0}, \hat{x}_{0,n} = x_{0,n-1}, n \geq 1 \\ A_2 &: \hat{x}_{m,0} = x_{m-1,0}, m \geq 1 \\ A_3 &: \hat{x}_{m,1} = x_{m,0}, m \geq 1 \end{aligned} \tag{4}$$

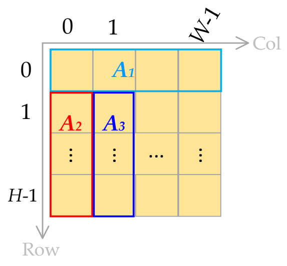


Figure 3. The three areas A_1, A_2, and A_3 cannot be predicted using the least square method.

The residual is the difference between the original value and the predicted value. The encode method is the range coder [8]. Since we are using the online DPCM method, the only data that needs to be encoded and transmitted to the decoder is the residual.

2.2. *Our Improvement to the Original Online DPCM Method*

Through experiments we found that the compression performance of the online DPCM algorithm can be improved by two improvements. One is on the establishment of the linear system of equations when computing the prediction coefficients, and the other is on the encoding of the residual.

2.2.1. The Improvement on the Establishment of the Linear System of Equations

From Equation (1), only one equation is built with each of the previous rows of pixel $x_{m,n}$, but through experiments we found that the compression performance can be improved when building multiple equations with each of the previous rows. We use the example shown in Figure 4 to illustrate how to build multiple equations with each row of an aurora image; $x_{m,n}$ is the pixel for which we are to compute its prediction coefficients. M is the number of equations built with each of the previous rows of $x_{m,n}$. Suppose $M = 3$, the prediction order $N = 4$, and the column number $n = 8$, then the M equations built with row i ($i = 0, 1, \ldots, m-1$) are (*), shown in Figure 4. It is worth noting that these equations can be obtained when sliding a window of length $N + 1$ on a row.

//M defines how many equations to build with each row of an aurora image
Conditions: $n = 8$, $N = 4$, $M = 3$

A part of row i ($i = 0, 1, \cdots, m - 1$) of an aurora image:

$x_{i,0}$	$x_{i,1}$	$x_{i,2}$	$x_{i,3}$	$x_{i,4}$	$x_{i,5}$	$x_{i,6}$	$x_{i,7}$	$x_{i,8}$

The M equations built with row i:

$$\begin{bmatrix} x_{i,2} & x_{i,3} & x_{i,4} & x_{i,5} \\ x_{i,3} & x_{i,4} & x_{i,5} & x_{i,6} \\ x_{i,4} & x_{i,5} & x_{i,6} & x_{i,7} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{bmatrix} = \begin{bmatrix} x_{i,6} \\ x_{i,7} \\ x_{i,8} \end{bmatrix} \quad (*)$$

Figure 4. An example illustrating how to create multiple equations with a row of an aurora image.

Combining the equations built with each of the rows before the pixel $x_{m,n}$, we get the complete linear system of equations used to compute the prediction coefficients of $x_{m,n}$, and the linear system of equations is shown in Equation (5). The M equations in the red dashed box are built with one row of the aurora image. It can be seen that each red dashed box contains $N + M$ different elements of a row of the aurora image, in other words, continuous $N + M$ elements in each of the rows before row m will be used to compute the prediction coefficients of the $x_{m,n}$. Figure 5 is used to explain this more clearly. Compared with Figure 2, the improved method requires more area, that is, $A_3(x_{m,n})$, whose number of columns is $M - 1$, and the total number of columns of $A_1(x_{m,n})$ and $A_3(x_{m,n})$ is $N + M$. Since Equation (5) is the improvement to Equation (1), we denote Equation (5) as $C\vec{\alpha} = \vec{b}$ as well, and C, $\vec{\alpha}$ and $\vec{b}$, which will be mentioned in the following contents, are the C, $\vec{\alpha}$, and $\vec{b}$ in Equation (5). Note that the number of columns of matrix C is the prediction order N, and the number of rows of matrix C is $M \times m$, which varies with the row number m.

$$\begin{bmatrix} x_{0,n-N-M+1} & x_{0,n-N-M+2} & \cdots & x_{0,n-M} \\ x_{0,n-N-M+2} & x_{0,n-N-M+3} & \cdots & x_{0,n-M+1} \\ \vdots \quad \text{Row 0} & \vdots & \cdots & \vdots \\ x_{0,n-N} & x_{0,n-N+1} & \cdots & x_{0,n-1} \\ \vdots & \vdots & \cdots & \vdots \\ x_{m-1,n-N-M+1} & x_{m-1,n-N-M+2} & \cdots & x_{m-1,n-M} \\ x_{m-1,n-N-M+2} & x_{m-1,n-N-M+3} & \cdots & x_{m-1,n-M+1} \\ \vdots \quad \text{Row m-1} & \vdots & \vdots & \vdots \\ x_{m-1,n-N} & x_{m-1,n-N+1} & \cdots & x_{m-1,n-1} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_N \end{bmatrix} = \begin{bmatrix} x_{0,n-M+1} \\ x_{0,n-M+2} \\ \vdots \\ x_{0,n} \\ \vdots \\ x_{m-1,n-M+1} \\ x_{m-1,n-M+2} \\ \vdots \\ x_{m-1,n} \end{bmatrix} \quad (5)$$

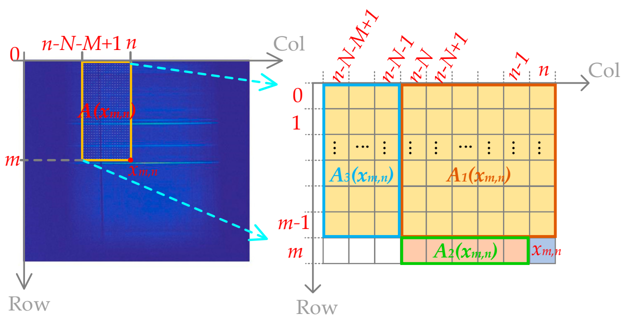


Figure 5. $A_1(x_{m,n})$ is the area that will be used to compute the prediction coefficients of $x_{m,n}$ when using the original online DPCM (Differential Pulse Code Modulation) method, and $A_1(x_{m,n})$ + $A_3(x_{m,n})$ is the area that will be used when using the improved online DPCM method.

2.2.2. The Improvement on the Encoding of the Residual

In this paper, the data to be compressed is the aurora spectral data captured at the Zhongshan Station in 2014, but we found that it had a shortcoming that would adversely affect the compression. Each aurora spectral image had some outliers that were much larger than other pixels, but only appear several times, as shown in Figure 6. These outliers are difficult to predict, consequently introducing very large residuals that have detrimental effects on compression. To handle this problem, we propose a dual threshold method that sets a threshold for positive and negative residuals, respectively. Only the residuals between the two thresholds will be encoded using the range coder, and other residuals will be stored intact in the compressed file.

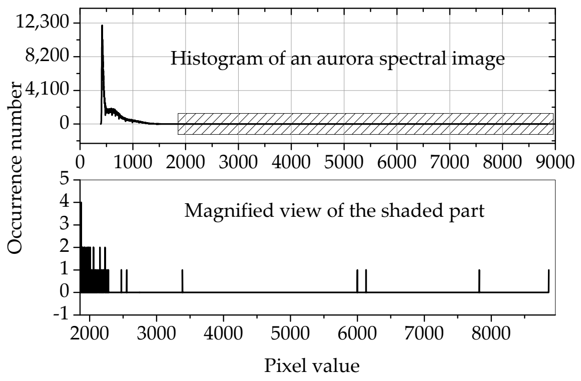


Figure 6. The upper part is the histogram of the aurora spectral image captured on October 17, 2014, at 17:41:20, and the lower part is the magnified view of the shaded part. It can be seen that the maximum pixel value is near 9000, but most of the pixels are within 500. The large majority of the pixel values do not appear in the shaded part, and the maximum number of occurrences is only 4.

Figure 7 shows the details of the dual threshold method. The T is a threshold for the occurrence number of the residuals. T_- is the first residual whose occurrence number is larger than or equal to T when searching from left to right, and T_+ is the first residual whose occurrence number is larger than or equal to T when searching from right to left. Only the residuals between T_- and T_+ will be encoded using the range coder, and the remaining residuals will be stored intact in the compressed file.

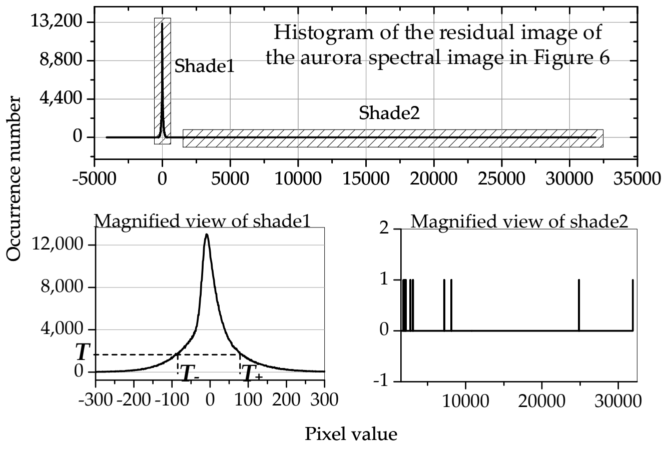


Figure 7. The upper part is the histogram of the residual image of the aurora spectral image in Figure 6; the lower left part is the magnified view of shade1. We can see that most of the residuals are around 0. The lower right part is the magnified view of shade2, where it can be seen that the largest residual is greater than 30,000, but most of the residual values do not appear in shade2, and the remaining residuals only appear once.

2.3. Optimization of the Parameters N, M, and T

The parameter N determines how many columns to use to predict the current pixel and M defines how many equations to build with each row of an aurora spectral image. Both of these parameters have important impacts on the accuracy of the prediction. The threshold T determines the two thresholds T_- and T_+, which will affect the encoding. In order to get the best compression performance, we must optimize these parameters N, M, and T through experiments. The experimental data set is the 2014 aurora spectral data set. Since aurora does not appear every day, the 2014 aurora spectral data set has more than 20 days of data, and each day has hundreds of images. In order to reduce the amount of calculations, we evenly take 100 samples from the images of each day for experiments.

Figure 8a presents the variation of the bitrate with the prediction order N. The bitrate is the average bitrate of 100 samples evenly selected from the data of each day and is measured in bpp (bits per pixel). The four values on the horizontal ordinate, 9, 10, 11, and 12, are determined by preliminary experiments. The five-pointed star symbol on each line indicates the optimal N that minimizes the average bitrate of each day, and the number near the five-pointed star symbol is the minimum bitrate. It can be seen that the optimal N for four days is 11, and for one day is 10. In order to determine the final N that minimizes the average bitrate of each day, we averaged the bitrate of the five days, and the result is the third line from top to bottom in Figure 8a. We can see that N = 11 minimizes the average bitrate of the five days, and therefore, we take N = 11 as the prediction order for the 2014 aurora spectral data compression.

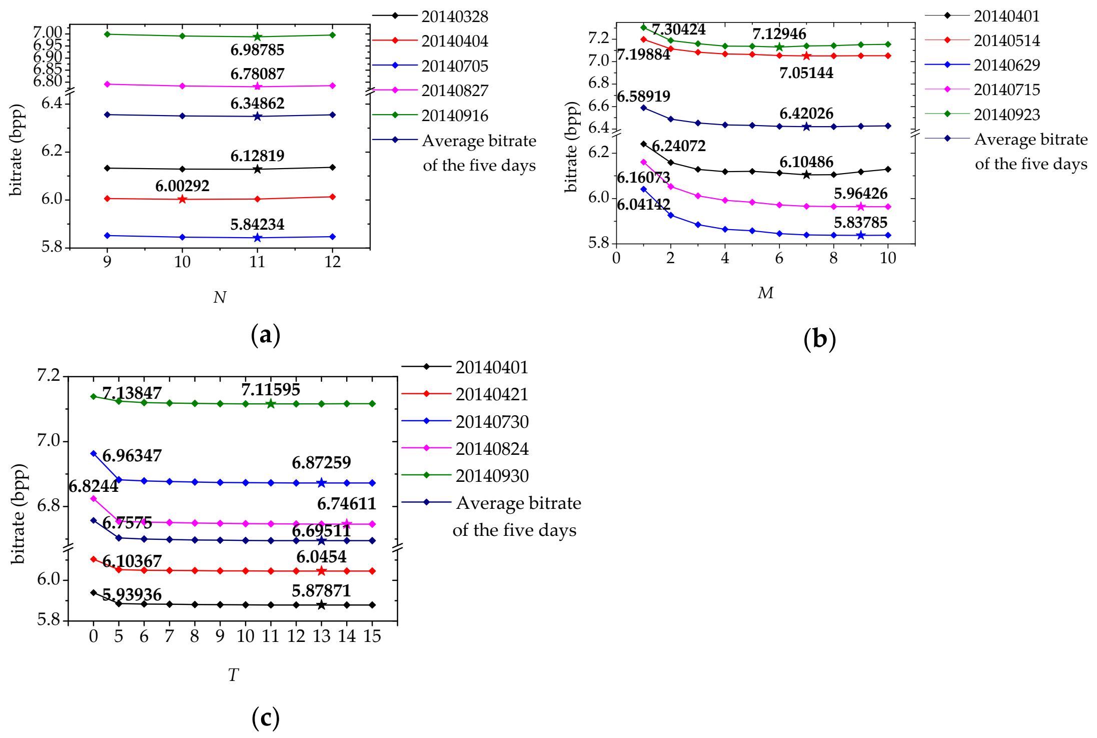


Figure 8. The variation of the bitrate with the three parameters N, M, and T. The bitrate is the average bitrate of each day, and the five days in each subfigure are randomly selected from the 2014 aurora spectral data set.

Figure 8b is similar to Figure 8a, but shows the variation of the bitrate with M, which defines how many equations to build with each row of an aurora spectral image. There are 10 values for M, 1, 2, . . . , 10. It can be seen that the optimal M that minimizes the bitrate of each day is different, but the M that minimizes the average bitrate of the five days is 7. Therefore, we take M = 7 for the compression of the 2014 aurora spectral data.

Note that the bitrate of each day when $M = 1$ is also presented near the first diamond node on each line in Figure 8b. The difference between the minimum bitrate and the bitrate when $M = 1$ ranges from more than 0.1 bpp to more than 0.2 bpp, and the average bitrate of the five days when $M = 1$ is about 0.17 bpp more than the minimum average bitrate of the five days. Consequently, we can conclude that the improvement to build multiple equations on each row of an aurora spectral image is beneficial to compression.

Figure 8c shows the variation of the bitrate with T, which determines the two encoding thresholds T_- and T_+. There are 12 values on the horizontal ordinate, 0, 5, 6, ... , 15. $T = 0$ represents that there are no thresholds and all the residuals will be encoded using the range coder. It can be seen that the optimal T that minimizes the bitrate of each day is different, but the minimum of the average bitrate of the five days is achieved when T is 13. Therefore, we will choose $T = 13$ for the compression of the 2014 aurora spectral data. In addition, the bitrate of each day at $T = 0$ is shown near the first diamond node on each line. We can see that the bitrate at $T = 0$ is larger than the minimum bitrate on each line, and the average bitrate of the five days when $T = 0$ is about 0.06 bpp more than the minimum average bitrate of the five days. Therefore, the dual threshold method for encoding is effective.

In addition, it can be seen that the average bitrate of each day can vary significantly, from about 5.8 bpp to about 7.2 bpp. This is because the aurora is constantly changing, including in its intensity. It can be seen that Figures 1a and 1b are very different.

3. The GPU Implementation of the Calculation of the Prediction Coefficients using the Improved Online DPCM Algorithm

The most time-consuming part of the online DPCM algorithm is the calculation of the prediction coefficients, which is because the prediction coefficients for each pixel are obtained by solving a linear system of equations. In addition, since this is an online method, the decoder also needs to compute the prediction coefficients for each pixel. Therefore, the calculation of the prediction coefficients must be accelerated, and this paper takes advantage of the powerful computing capability of GPUs.

From Equation (2), the main operations in the calculation of the prediction coefficients are matrix multiplication and matrix inversion. Hence, the rest of this section will focus on these two steps, but before that, we will introduce some basic concepts of GPU programming to facilitate the following descriptions.

3.1. Some Basic Concepts of GPU Programming

GPU is specialized for computation-intensive tasks. The programming model of GPU is CUDA, and the programming language is CUDA C. The kernel functions, which are much like the C functions but run on GPU, are labelled by hundreds of thousands of CUDA threads in parallel to handle the massive computations, and NVIDIA (a GPU corporation) denotes this architecture as Single Instruction Multiple Thread (SIMT).

All the threads assigned to a kernel function are organized into a two-hierarchical grid structure. A number of threads form a thread block, and a number of thread blocks form a thread grid. The size of all the thread blocks in a thread grid is the same. Both a thread block and a thread grid can be one-dimensional, two-dimensional, or three-dimensional. Figure 9 presents an example describing the thread structure. The size of the thread grid is 3×2, and the size of each thread block is 4×3. The ordered pair in parentheses is the index of a thread or a thread block.

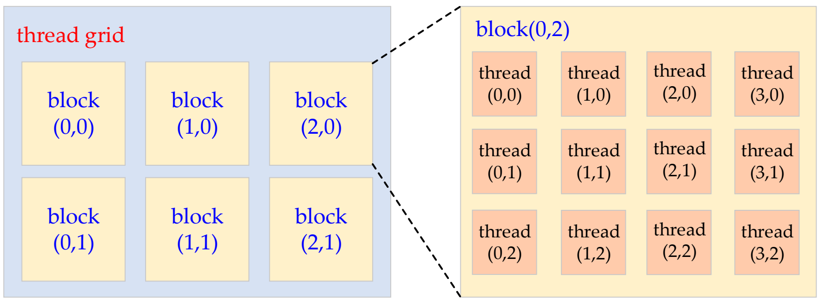


Figure 9. An example illustrating the thread structure of a kernel function.

The typical processing flow of a CUDA program is: (1) copy the data to be processed from host (the CPU memory) to device (the GPU memory); (2) launch the kernel function to process the data on the GPU; (3) copy the result from device to host. In addition, there is a type of memory that is very critical to the performance of a CUDA program—shared memory. Shared memory is on-chip memory, and its access latency is only a few clock cycles, which is much less than the access latency of the hundreds of clock cycles of global memory. Furthermore, shared memory provides a means of communication for all of the threads in a thread block.

3.2. CUDA Implementation of the Multiplication of Matrices C^T and C using a Decomposition Method

Matrix multiplication is a common operation in CUDA programming, and has been optimized very well. The Basic Linear Algebra Subprogram (cuBLAS) [27] is a linear algebra acceleration library for CUDA, which can process not only the matrix multiplication but also the matrix multiplication of a batch of matrices. In our case, the multiplication of matrices C^T and C in Equation (2) is required to predict each pixel, and there are 1024×1024 pixels in each aurora spectral image. However, the multiplication of matrices C^T and C cannot be batch processed using the cuBLAS library, which is because the size of matrix C is not fixed; it is determined by the row number of the pixel to be predicted, as shown in Equation (5), and the cuBLAS library can only process a batch of matrices with the same dimensions. In addition, if the multiplication of matrices C^T and C for each pixel is performed individually using the cuBLAS matrix multiplication function, the launch overhead is too heavy. Therefore, the cuBLAS library is not adopted for the multiplication of matrices C^T and C, and we must propose a scheme that is suitable for the situation in this paper.

From Equation (5), when predicting a pixel $x_{m,n}$ at the mth row and the nth column, all the rows before row m will be involved in its matrix C; let us label the matrix C as $C(x_{m,n})$ for the simplicity of description. Similarly, all of the rows before row $m + 1$ will be involved in matrix $C(x_{m+1,n})$ when predicting $x_{m+1,n}$. Furthermore, the first $m \times M$ row in matrix $C(x_{m+1,n})$ is exactly the matrix $C(x_{m,n})$, so if the multiplication of matrix $C^T(x_{m,n})$ (the transposed matrices of $C(x_{m,n})$) and $C(x_{m,n})$ and the multiplication of matrices $C^T(x_{m+1,n})$ and $C(x_{m+1,n})$ are performed directly and individually, many repeated data access instances and calculations will be introduced, which will seriously affect the computational efficiency. Therefore, we proposed a decomposition method for the multiplication of matrix C^T and C to handle this problem.

Figure 10 shows an example illustrating the decomposition method for the multiplication of matrices C^T and C. A small part of the upper left corner of an aurora spectral image is shown in the upper right corner of Figure 10. For simplicity, we assume that $N = 3$ and $M = 2$, and that we are to predict three pixels, $x_{1,8}$, $x_{2,8}$, and $x_{3,8}$. $C(x_{1,8})$, $C(x_{2,8})$, and $C(x_{3,8})$ represent matrix C for the three pixels $x_{1,8}$, $x_{2,8}$, and $x_{3,8}$, respectively, and $C^T(x_{1,8})$, $C^T(x_{2,8})$, and $C^T(x_{3,8})$ are the transposed matrices for $C(x_{1,8})$, $C(x_{2,8})$, and $C(x_{3,8})$, respectively. The two rows of matrix $C(x_{1,8})$ are grouped together and denoted as $C_{0,8}$. The subscript "0" of $C_{0,8}$ represents that the two rows are built with row 0 of the aurora image, and the subscript "8" represents that the pixel to be predicted is located in column 8. Similarly, $C(x_{2,8})$ is partitioned into two submatrices $C_{0,8}$ and $C_{1,8}$, and $C(x_{3,8})$ is partitioned into three submatrices $C_{0,8}$, $C_{1,8}$, and $C_{2,8}$. Therefore, the product of the matrices $C^T(x_{1,8})$ and $C(x_{1,8})$ is $C^T{}_{0,8}C_{0,8}$; the product of $C^T(x_{2,8})$ and $C(x_{2,8})$ is $C^T{}_{0,8}C_{0,8} + C^T{}_{1,8}C_{1,8}$, which can also be considered as $C^T(x_{1,8})C(x_{1,8}) + C^T{}_{1,8}C_{1,8}$; the product of $C^T(x_{3,8})$ and $C(x_{3,8})$ is $C^T{}_{0,8}C_{0,8} + C^T{}_{1,8}C_{1,8} + C^T{}_{2,8}C_{2,8}$, which can also be considered as $C^T(x_{2,8})C(x_{2,8}) + C^T{}_{2,8}C_{2,8}$. It can be seen that the multiplication of matrices $C^T(x_{m,n})$ and $C(x_{m,n})$ is transformed into the addition of many small matrices, and $C^T(x_{m,n})C(x_{m,n})$ is the sum of $C^T(x_{m-1,n})C(x_{m-1,n})$ and $C^T{}_{m-1,n}C_{m-1,n}$. In this way, all the matrices $C^T(x_{i,n})C(x_{i,n})$ ($i = 1, 2, \ldots, m$) can be obtained when the prefix sums an array of matrices, $C^T{}_{0,n}C_{0,n}$, $C^T{}_{1,n}C_{1,n}$, … , $C^T{}_{m-1,n}C_{m-1,n}$. The idea of this method is intuitively presented in Figure 11.

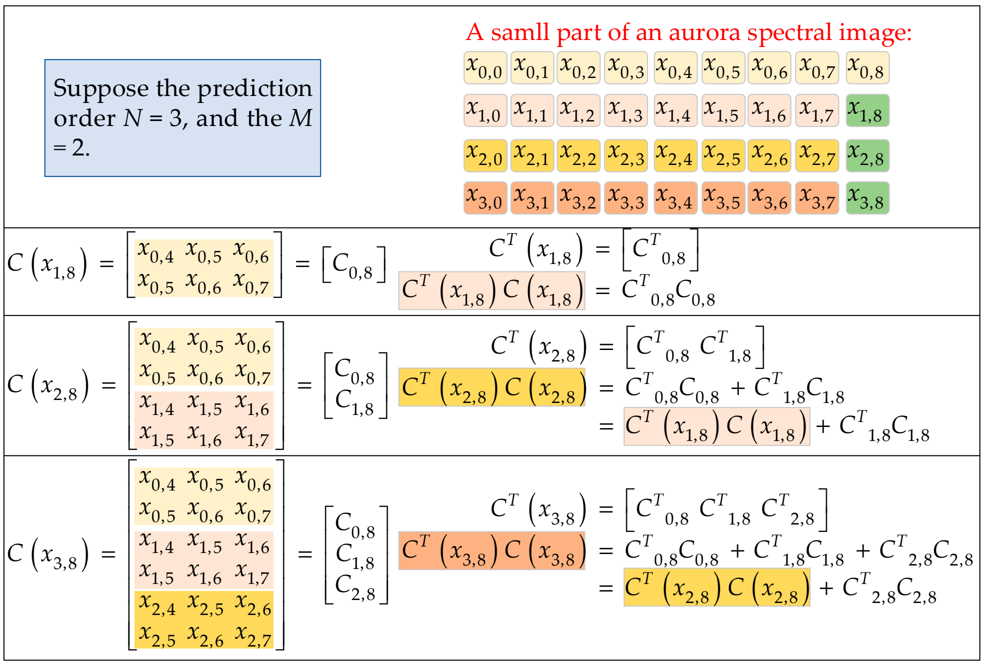

Suppose the prediction order $N = 3$, and the $M = 2$.
A samll part of an aurora spectral image:
$x_{0,0}$ $x_{0,1}$ $x_{0,2}$ $x_{0,3}$ $x_{0,4}$ $x_{0,5}$ $x_{0,6}$ $x_{0,7}$ $x_{0,8}$
$x_{1,0}$ $x_{1,1}$ $x_{1,2}$ $x_{1,3}$ $x_{1,4}$ $x_{1,5}$ $x_{1,6}$ $x_{1,7}$ $x_{1,8}$
$x_{2,0}$ $x_{2,1}$ $x_{2,2}$ $x_{2,3}$ $x_{2,4}$ $x_{2,5}$ $x_{2,6}$ $x_{2,7}$ $x_{2,8}$
$x_{3,0}$ $x_{3,1}$ $x_{3,2}$ $x_{3,3}$ $x_{3,4}$ $x_{3,5}$ $x_{3,6}$ $x_{3,7}$ $x_{3,8}$
$C(x_{1,8}) = \begin{bmatrix} x_{0,4} & x_{0,5} & x_{0,6} \\ x_{0,5} & x_{0,6} & x_{0,7} \end{bmatrix} = [C_{0,8}]$
$C^T(x_{1,8}) = [C^T{}_{0,8}]$
$C^T(x_{1,8})C(x_{1,8}) = C^T{}_{0,8}C_{0,8}$
$C(x_{2,8}) = \begin{bmatrix} x_{0,4} & x_{0,5} & x_{0,6} \\ x_{0,5} & x_{0,6} & x_{0,7} \\ x_{1,4} & x_{1,5} & x_{1,6} \\ x_{1,5} & x_{1,6} & x_{1,7} \end{bmatrix} = \begin{bmatrix} C_{0,8} \\ C_{1,8} \end{bmatrix}$
$C^T(x_{2,8}) = [C^T{}_{0,8}\ C^T{}_{1,8}]$
$C^T(x_{2,8})C(x_{2,8}) = C^T{}_{0,8}C_{0,8} + C^T{}_{1,8}C_{1,8} = C^T(x_{1,8})C(x_{1,8}) + C^T{}_{1,8}C_{1,8}$
$C(x_{3,8}) = \begin{bmatrix} x_{0,4} & x_{0,5} & x_{0,6} \\ x_{0,5} & x_{0,6} & x_{0,7} \\ x_{1,4} & x_{1,5} & x_{1,6} \\ x_{1,5} & x_{1,6} & x_{1,7} \\ x_{2,4} & x_{2,5} & x_{2,6} \\ x_{2,5} & x_{2,6} & x_{2,7} \end{bmatrix} = \begin{bmatrix} C_{0,8} \\ C_{1,8} \\ C_{2,8} \end{bmatrix}$
$C^T(x_{3,8}) = [C^T{}_{0,8}\ C^T{}_{1,8}\ C^T{}_{2,8}]$
$C^T(x_{3,8})C(x_{3,8}) = C^T{}_{0,8}C_{0,8} + C^T{}_{1,8}C_{1,8} + C^T{}_{2,8}C_{2,8} = C^T(x_{2,8})C(x_{2,8}) + C^T{}_{2,8}C_{2,8}$

Figure 10. An example illustrating the decomposition method for the multiplication of matrices C^T and C.

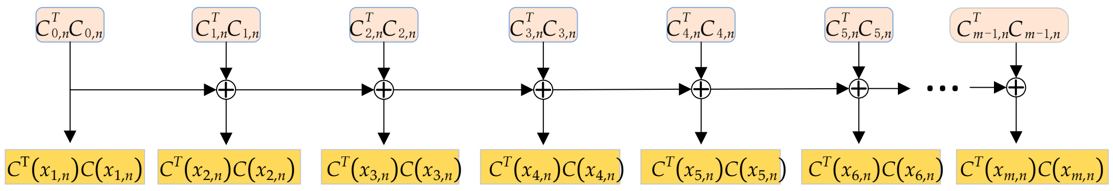
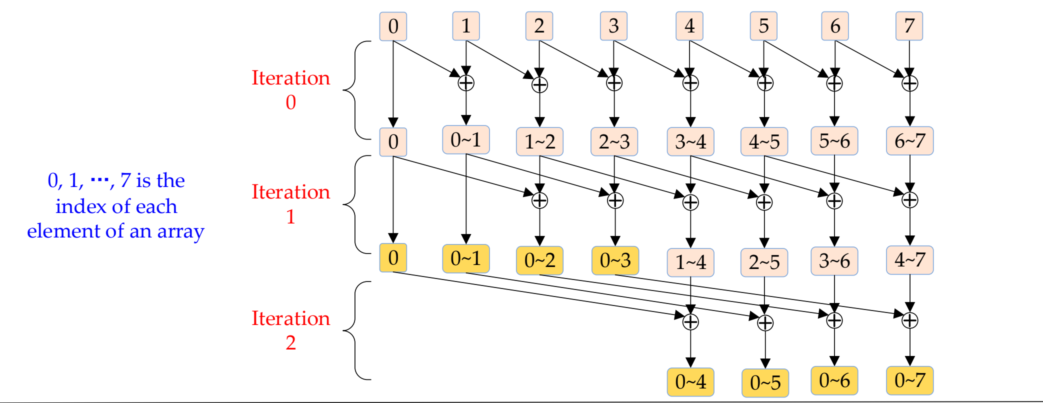


Figure 11. The method for computing a series of matrices, $C^T(x_{1,n})C(x_{1,n})$, $C^T(x_{2,n})C(x_{2,n})$, … , $C^T(x_{m,n})C(x_{m,n})$. $C_{i,n}$ ($i = 0, 1, \ldots, m-1$) is the matrix built with row i of an aurora image when predicting the pixels in column n. The symbol "$\oplus$" represents addition.

In order to accelerate the method introduced in Figure 11, a parallel scheme shown in Figure 12 is utilized in this paper. It is noteworthy that the purpose of the summation of two matrices is to add the elements in the same position. Similarly, the purpose of the prefix summing of a series of matrices is to prefix sum arrays composed of all the elements at the same position in the matrices. Therefore, the method introduced in Figure 12 is actually a parallel method for prefix summing of arrays. The upper part of Figure 12 is an example that depicts how to compute the prefix sum of an array with 8 elements. The i~j ($i < j$) represents the sum of the ith element to the jth element. It can be seen that the prefix summing of 8 elements is completed through 3 iterations.

The lower part of Figure 12 shows the CUDA codes for prefix summing of elements in a warp. The __shfl_up_sync() [28] is a CUDA shuffle instruction, which allows threads in the same warp to exchange data directly without consuming extra memory, and has lower latency than shared memory. This is why we use __shfl_up_sync(). The maximum range of action of __shfl_up_sync() is an entire warp, but in fact a warp can be evenly divided into a few thread groups, and __shfl_up_sync() is executed on each thread group individually. The *width* in the CUDA codes in Figure 12 is exactly the size of each thread group, and it is set to warpSize, which is 32. The function of __shfl_up_sync (*mask*, *value*, *i*, *width*) is to get the value of the *value* of the thread *lane_id* − *i*. It can be seen that the main body of the CUDA codes is a *for* loop. In the *i*th iteration, all the threads except the first *i* threads call __shfl_up_sync() to get the value of *value* held by the thread *lane_id* − *i*, and then add the obtained value to its own *value*. After five iterations, the prefix summing of a warp with 32 threads is completed. However, this only achieves the prefix summing in a warp; to compute the prefix sum of 1023 matrices (each aurora image has 1024 rows), we also need to compute the prefix sum inside each thread block, and finally the prefix sum across the entire thread grid.



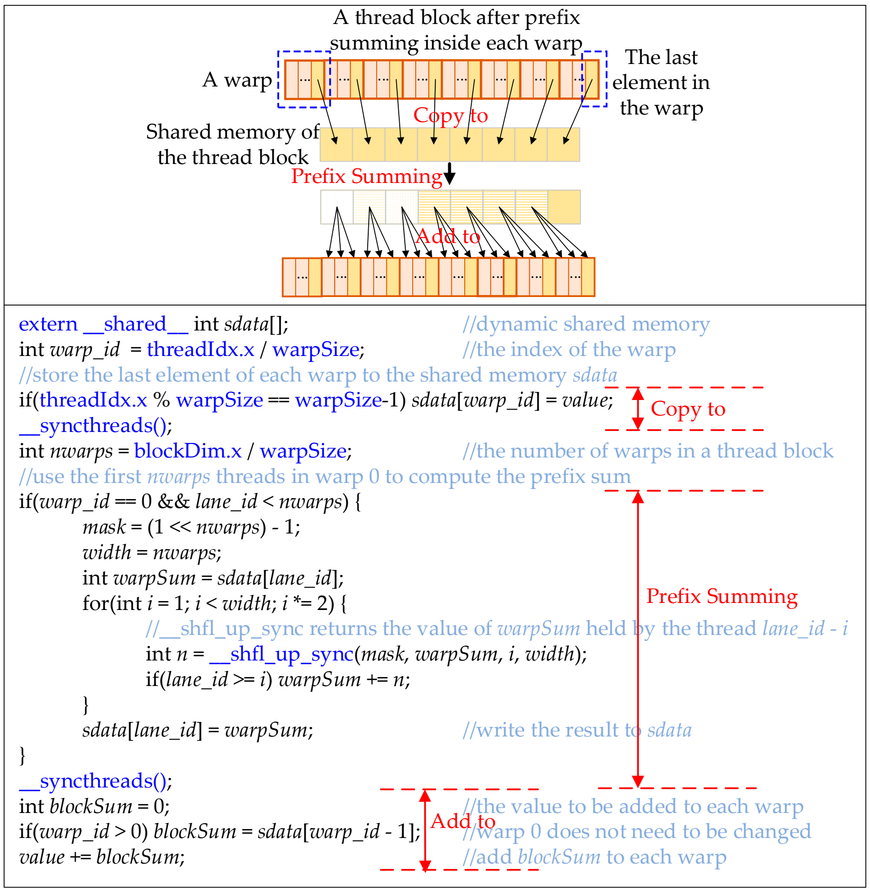
```
int idx = threadIdx.x + blockIdx.x * blockDim.x;  //the index of the thread in the thread grid
if ( idx < N) {                                   //N is the total number of elements
        int value = data[idx];                    //read data from the global memory data[idx]
        //width is the size of the thread group that the shuffle instruction is performed on
        int width = warpSize;                     //warpSize is a built-in variable, which defines the size of a warp
        int lane_id = threadIdx.x % width;        //the index of the thread in the thread group
        unsigned int mask = 0xffffffff;           //indicating the threads participating in the shuffle instruction
        for (int i = 1; i < width; i *= 2){
                //__shfl_up_sync() returns the value of the value held by the thread lane_id - i
                int n = __shfl_up_sync(mask, value, i, width);
                if (lane_id >= i) value += n;
        }
        data[idx] = value;                        //write the result to the global memory data[idx]
}
```

Figure 12. The upper part shows an example illustrating the idea of computing the prefix sum of an array in parallel, and the lower part presents the CUDA (Compute Unified Device Architecture) codes. The symbol "⊕" represents addition.

Figure 13 shows the second half of the thread block prefix summing, that is, prefix summing of the last element of all the warps in a thread block and adding each element of the resulting array to the corresponding warp. The upper part of Figure 13 depicts the general idea, and the lower part gives the CUDA codes. The codes can be divided into three parts. The first part is to copy the *value*, which stores the last element of each warp after the operations in Figure 12, to the shared memory *sdata*. The second part is to prefix sum the elements in *sdata*. Since there are *nwarps* warps in each thread block, and the last parameter *width* of __shfl_up_sync() is set to *nwarps*. The last part is to add each element of *sdata* to the corresponding warp. Since there is no warp before warp 0, 0 is added to warp 0.

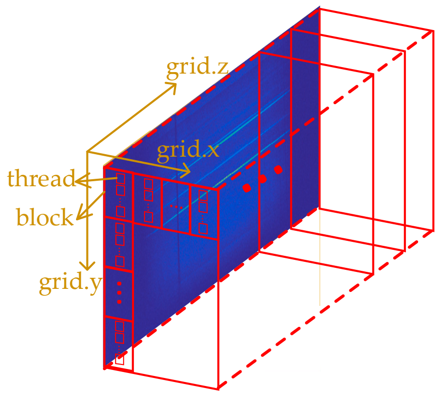
Combining the codes in Figures 12 and 13, we get the CUDA kernel function for computing the prefix sum within each thread block, and denote the kernel as C^T*CBlockPrefixKernel*. The thread structure of C^T*CBlockPrefixKernel* is presented in Figure 14. Each page of the thread grid processes a column of an aurora image, and there are WIDTH − 2 pages in total, which is because the first two columns are predicted in advance, as illustrated in Figure 3. Suppose each thread block has *nthreads* threads. Since the first row of an aurora image is also predicted in advance, there are HEIGHT − 1 C^TC matrices in each column of the aurora image, and the y dimension of the thread grid is (HEIGHT − 1 + block.x − 1)/block.x. Each C^TC matrix has $N \times N$ (N is the prediction order) elements, so there are $N \times N$ arrays to prefix summing, and the x dimension of the thread grid is $N \times N$.

The scheme for computing the prefix sum across a column of thread blocks of Figure 14 is similar to the method described in Figure 13. First compute the prefix sum of the last elements of each thread block, and then add each element of the resulting array to the corresponding thread block. After this procedure, we can get the matrices $C^T(x_{m,n})C(x_{m,n})$ ($m = 1, 2, \ldots, 1023$, $n = 2, 3, \ldots, 1023$) (the first row and the first two columns of an aurora image are predicted in advance, as illustrated in Figure 3).


A thread block after prefix summing inside each warp
A warp
The last element in the warp
Copy to
Shared memory of the thread block
Prefix Summing
Add to
Copy to
Prefix Summing
Add to

```
extern __shared__ int sdata[];                    //dynamic shared memory
int warp_id  = threadIdx.x / warpSize;            //the index of the warp
//store the last element of each warp to the shared memory sdata
if(threadIdx.x % warpSize == warpSize-1) sdata[warp_id] = value;
__syncthreads();
int nwarps = blockDim.x / warpSize;               //the number of warps in a thread block
//use the first nwarps threads in warp 0 to compute the prefix sum
if(warp_id == 0 && lane_id < nwarps) {
        mask = (1 << nwarps) - 1;
        width = nwarps;
        int warpSum = sdata[lane_id];
        for(int i = 1; i < width; i *= 2) {
                //__shfl_up_sync returns the value of warpSum held by the thread lane_id - i
                int n = __shfl_up_sync(mask, warpSum, i, width);
                if(lane_id >= i) warpSum += n;
        }
        sdata[lane_id] = warpSum;                 //write the result to sdata
}
__syncthreads();
int blockSum = 0;                                 //the value to be added to each warp
if(warp_id > 0) blockSum = sdata[warp_id - 1];    //warp 0 does not need to be changed
value += blockSum;                                //add blockSum to each warp
```

Figure 13. Parallel implementation of prefix summing of an array of the last element of all the warps in a thread block. The prefix sum of each warp has been calculated using the method in Figure 12. Combining Figure 12 with Figure 13, we get the prefix sum of each thread block in a thread grid.



```
//each block has nthreads threads
dim3 block(nthreads);
//N is the prediction order
//HEIGHT is the height of an aurora image
//WIDTH is the width of an aurora image
dim3 grid(NxN, (HEIGHT-1+block.x-1)/block.x, WIDTH-2);
```

Figure 14. The thread structure for C^T*CBlockPrefixKernel*.

The preceding contents have talked about how to decompose the $C^T(x_{m,n}) \times C(x_{m,n})$ into the addition of many small matrices $C^T{}_{i,n}C_{i,n}$ $(i = 0, 1, \ldots, m-1)$, but the multiplication of the matrices $C^T{}_{i,n}$ and $C_{i,n}$ still remains a problem. From Figure 10, the size of $C_{i,n}$ is $M \times N$, M determines how many equations to build with each row of an aurora image and N is the prediction order. Therefore, the dimension of the matrix $C^T{}_{i,n}C_{i,n}$ is $N \times N$. Furthermore, the optimal N is 11 for the 2014 aurora data set, as described in the Section 2.3, so there are $11 \times 11 = 121$ elements in each $C^T{}_{i,n}C_{i,n}$, which is much less than the maximum number of threads (1024) supported by each thread block of the recent GPU cards. Consequently, a thread block is enough for the multiplication of $C^T{}_{i,n}$ and $C_{i,n}$, and each thread is responsible for an element of the product matrix $C^T{}_{i,n}C_{i,n}$. Of course, if the dimension of the matrix $C^T{}_{i,n}C_{i,n}$ is larger than the maximum number of threads supported by each thread block, the product matrix $C^T{}_{i,n}C_{i,n}$ can be divided into small submatrices and each thread block processes one submatrix, as presented in the matrix multiplication sample in a previous study [28].

Figure 15 gives the CUDA implementation of the multiplication of matrices $C^T{}_{i,n}$ and $C_{i,n}$. The size of the thread block is $N \times N$, which is the same as the size of the product matrix $C^T{}_{i,n}C_{i,n}$. Each thread is responsible for one element of $C^T{}_{i,n}C_{i,n}$, in other words, each thread is responsible for the multiplication of a row of $C^T{}_{i,n}$ and a column of $C_{i,n}$. The size of the thread grid is (HEIGHT − 1) × (WIDTH − 2), which is because the first row and the first two columns of an aurora image are predicted in advance. Since the first several columns of an aurora image cannot establish a matrix $C_{i,n}$ with M rows and N columns, *cols* and *rows* are used to represent the actual width and height of $C_{i,n}$, respectively. If *col* (the index of the current column of the aurora image) is less than or equal to N, *cols* is *col*, and *rows* is 1, otherwise, *cols* is N, and *rows* is $col - N + 1$, which is the maximum number of equations can be built with each row of the aurora image, but if *rows* is larger than M, *rows* is set to M. Since all the elements of $C_{i,n}$ are from a consecutive segment of row i of the aurora image, as shown in Figure 10, the segment is copied to the shared memory *sdata* to reduce the accesses of the global memory *d_data* and take advantage of the low latency of the shared memory. It is noteworthy that in this way, we do not need to prepare the matrix $C_{i,n}$ in advance and copy it from host to device, and the only data that needs to be transferred from host to device is the aurora data, so a lot of time is saved.

```
dim3 block(N, N);              //N is the prediction order, that is, dimension of each matrix C^T_{i,n}C_{i,n}
dim3 grid(WIDTH - 2, HEIGHT - 1);//WIDTH is the width of an aurora image, and HEIGHT is the height
//d_data is the aurora image on the device, d_C^TC is the device memory to store the matrices C^T_{i,n}C_{i,n}
__global__ void matrixMulKernel(unsigned short *d_data, double *d_C^TC) {
        int col = blockIdx.x + 2;      //which column of the aurora image this thread block processes
        int row = blockIdx.y;          //by which row of the aurora image the matrix C_{i,n} is created
        int cols =  col;               //cols is the actual number of columns of C_{i,n}
        /* rows is the actual number of rows of C_{i,n}, that is, how many equations can be built with row row
        of the aurora image */
        int rows = 1;
        if(col > N) {
                cols = N;
                rows = col – N + 1;
                if(rows > M)           //the maximum rows is M
                        rows = M;
        }
        int idx = threadIdx.x + threadIdx.y * blockDim.x;        //the index of this thread in the thread block
        extern __shared__ unsigned short sdata[];
        /* all the elements of C_{i,n} are from a consecutive segment of row i of the aurora image. The
        segment starts from column col-rows+1-cols, ends in column col-1, and the length is cols+rows–1. */
        //copy this segment to the shared memory sdata
        if(idx < cols + rows - 1)    sdata[idx] = d_data[row * WIDTH + col – rows + 1 – cols + idx];
        __syncthreads();
        //the offset of the current C^T_{i,n}C_{i,n} in d_C^TC
        int offset = N * N * (HEIGHT - 1) * blockIdx.x + cols * cols * blockIdx.y;
        if(threadIdx.x < cols && threadIdx.y < cols) {            //the actual size of C^T_{i,n}C_{i,n} is cols*cols
                double sum = 0.0;
                /* each thread is responsible for one element of C^T_{i,n}C_{i,n}, that is, the multiplication of a row
                of C^T_{i,n} and a column of C_{i,n} */
                for(int i = 0; i < rows; i++)    sum += sdata[threadIdx.y + i] * sdata[threadIdx.x + i];
                d_C^TC[offset + threadIdx.y * cols + threadIdx.x] = sum;
        }
}
```

Figure 15. The CUDA kernel function for the multiplication of matrices $C^T{}_{i,n}$ and $C_{i,n}$.

So far, the introduction of the matrix multiplication is complete, but this decomposition idea can also be applied to the multiplication of C^T and $\vec{b}$ in Equation (2). This is because $C(x_{i-1,n})$ is the prefix of $C(x_{i,n})$, as shown in Figure 10, and $\vec{b}(x_{i-1,n})$ (the right-hand-side vector corresponding to $C(x_{i-1,n})$) is also the prefix of $\vec{b}(x_{i,n})$.

3.3. CUDA Implementation of the Inversion of Matrix C^TC using the Gaussian Jordan Elimination Method

As shown in Equation (2), the inversion of matrix C^TC is one of the most important procedures when computing the prediction coefficients of a pixel of an aurora image. The method for the inversion of C^TC used in this paper is the Gaussian Jordan Elimination method [29,30], and this subsection mainly talks about how to accelerate it in parallel using CUDA. The main step of the Gaussian Jordan Elimination method is to extend an identity matrix of the same dimension to the right of C^TC, and then sequentially change each column of C^TC into a unit vector through elimination. When matrix C^TC finally becomes an identity matrix, the right half matrix is the inverse matrix. That is, $[C^TC \vdots E] \overset{elimination}{\longrightarrow} [E \vdots (C^TC)^{-1}]$, E is an N-dimensional unit matrix. In addition, it is necessary to select the main element before elimination to control the rounding error and avoid division by zero, and for convenience, the range we adopted for selecting the main element is a column.

The reason why we choose the Gaussian Jordan Elimination method is that the dimension of the matrix C^TC is N, and the optimal N is 11 for the 2014 aurora spectral data set, so C^TC is a medium and small-sized matrix. If we use the adjoint matrix method to compute the inverse matrix of C^TC, we need to compute $N \times N$ algebraic cofactors and the determinant of C^TC, which is a large amount of calculations. If we use the decomposition methods, such as LU decomposition (which decomposes a matrix into a unit Lower triangular matrix and an Upper triangular matrix) or QR decomposition (which decomposes a matrix into an orthogonal matrix and an upper triangular matrix), we first need to decompose C^TC into several submatrices, then compute the inverse matrix of each submatrix, and finally compute the product of these inverse matrices. Each step of the decomposition method requires at least one call of a kernel function. Therefore, the decomposition method is very complicated and its processing time may be less than that of the Gaussian Jordan method for large matrices, but for the medium and small-sized matrix C^TC, the launch overhead of the kernel functions may offset the advantage of the decomposition method.

The CUDA implementation of the Gaussian Jordan method for the calculation of the inverse matrix of C^TC is presented in Figure 16. Each thread block calculates the inverse matrix of a matrix C^TC, and the size of the thread block is $(N \times 2) \times N$, which is the same as the size of the augmented matrix $invC^TC$; *cols* is the actual dimension of the matrix C^TC. First, the matrix C^TC is copied to the left half of $invC^TC$, and the right half of $invC^TC$ is set to a unit matrix. Then, each iteration of the *for* loop turns one column of $invC^TC$ into a unit vector through Gaussian Jordan Elimination. Since the actual number of columns of C^TC is *cols*, there are *cols* iterations in the *for* loop. The first step in each iteration is to search the main element of a column. As there are only $N = 11$ elements in a column, and the search range of the main element in the kth iteration is from row k to the last row, which means that the search range of the main element is decreasing, thread 0 is assigned to search the main element in a column serially. Note that if the main element equals zero, there will be a division by zero error, and the matrix C^TC is considered to be singular. The shared memory *kk* is used to store the main element, and the *pivot* is the row index of the main element. Now that the column main element has been calculated, we need to normalize the row *pivot* and exchange it with row k. Since the row pivot will be used several times during elimination, our scheme is to copy the row *pivot* to the shared memory *srow* to reduce the accesses of the global memory. In this way, the normalization is performed on the *srow*, and the row *pivot* and row k can be exchanged with the help of *srow*. The last step is elimination. Each row of the matrix $invC^TC$, except row k, minus a certain ratio of the row *pivot* to turn the kth element of this row into zero. Actually, the ratio is the ratio of the kth element of each row to the kth element of row *pivot*. Since the row *pivot* has been normalized and the kth element has turned into 1, the ratio becomes the kth element of each row. It can be seen that the column k will be accessed many times during the elimination, therefore, it is copied to the shared memory *scol* to avoid the severe access latency of the global memory. After *cols* iterations, the left half of $invC^TC$ is turned into a unit matrix, and the right half is exactly the inverse matrix of C^TC.

```
//N is the dimension of matrix C^TC, N*2 is the number of columns of the augmented matrix
dim3 block(N * 2, N);
//WIDTH is the width of the aurora image, and HEIGHT is the height
dim3 grid(WIDTH - 2, HEIGHT - 1);
//d_invC^TC is used to store the augmented matrix of d_invC^TC, d_info records the singularity of each C^TC
__global__ void gaussianJordanKernel(double *d_C^TC, double *d_invC^TC, int *d_info) {
        int col = blockIdx.x + 2;                    //the index of the current column of the aurora image
        int cols = col <= N ? col : N;               //the actual dimension of the current C^TC
        //invC^TCPtr is the pointer to the current matrix invC^TC
        double *invC^TCPtr = d_invC^TC + N * N * 2 * (HEIGHT - 1) * blockIdx.x + cols * cols * 2 * blockIdx.y;
        //copy C^TC to the left half of invC^TC, and set the right half of invC^TC to a unit matrix
        /* ··· code ··· */
        extern __shared__ double sdata[];
        //scol has N elements, and is used to store one column of invC^TC
        double *scol = sdata;
        //srow has N * 2 elements, and is used to store one row of invC^TC
        double *srow = sdata + N;
        int idx = threadIdx.y * blockDim.x + threadIdx.x;        //the index of the thread in the thread block
        __shared__ double kk;                                    //the value of the main element
        __shared__ int pivot;                                    //the row index of the main element
        //each iteration turns one column of C^TC into a unit vector
        for(int k = 0; k < cols; k++) {
                //copy column k of C^TC into the shared memory scol
                if(idx >= 0 && idx < cols) scol[idx] = invC^TCPtr[idx * cols * 2 + k];
                __syncthreads();
                if(idx == 0) {
                        //search the main element
                        double max = scol[k]; pivot = k;
                        for(int i = k + 1; i < cols; i++) {
                                double tmp = scol[i];
                                if(fabs(tmp) > fabs(max)){ max = tmp; pivot = i; }
                        }
                        //if the main element equals to zero, the matrix is considered to be singular
                        if(fabs(max) < 1e-6) {
                                d_info[(HEIGHT - 1) * blockIdx.x + blockIdx.y] = 1;     //1 indicates singularity
                                break;
                        }
                        kk = invC^TCPtr[pivot * cols * 2 + k];
                }
                __syncthreads();
                //normalize row pivot and exchange row pivot and row k
                if(idx >= k && idx < cols * 2) {
                        srow[idx] = invC^TCPtr[pivot * cols * 2 + idx];                 //copy row pivot to srow
                        srow[idx] /= kk;                                                 //normalize row pivot
                        invC^TCPtr[pivot * cols * 2 + idx] = invC^TCPtr[k * cols * 2 + idx];
                        invC^TCPtr[k * cols * 2 + idx] = srow[idx];
                }
                //exchange scol[pivot] and scol[k]
                if(idx == 0){ double tmp = scol[pivot]; scol[pivot] = scol[k]; scol[k] = tmp; }
                __syncthreads();
                //all rows in invC^TC except row k perform the elimination
                if(threadIdx.y < cols && threadIdx.y != k && threadIdx.x > k && threadIdx.x < cols * 2)
                        invC^TCPtr[threadIdx.y * cols * 2 + threadIdx.x] -= scol[threadIdx.y] * srow[threadIdx.x];
                __syncthreads();
        }
}
```

Figure 16. The CUDA kernel function for the computation of the inverse matrix of C^TC.

4. Experimental Results

The previous two sections have detailed the online DPCM algorithm and its CUDA implementation, and this section will focus on the verification of the performance of the CUDA implementation through experiments. The test data set is the aurora spectral data set captured at the Zhongshan Station in 2014, which is introduced in Section 1. The CPU that runs the serial program is the Intel Xeon E3-1240 v3 CPU, whose dominant frequency is 3.4 GHz. The GPU is the NVIDIA TITAN X GPU, which has 3584 CUDA cores and 12 GB GDDR5X memory, and the memory bandwidth is 480 GB/s.

4.1. The Compression Performance of the Improved online DPCM Algorithm Compared with Several Other Lossless Compression Algorithms

The details of the improved online DPCM algorithm are introduced in Section 2, and the three parameters N, M, and T are set to their respective optimal values, that is, $N = 11$, $M = 7$, and $T = 13$. Figure 17 presents the compression performance of the online DPCM algorithm and a few other lossless compression algorithms. The Bzip2 and PPMD (the Prediction by Partial Matching algorithm implemented by Dmitry)are implemented in the 7-Zip software [31]. The Bzip2 is based on the Burrows–Wheeler transform (BWT) [32] lossless data compression algorithm. The PPMD is the implementation of the PPM (Prediction by Partial Matching) with Information Inheritance (PPMII) algorithm [33] with small changes, and the PPMII is a modification of the PPM [34] algorithm to reduce its high computational complexity. The JPEG-LS [11] and the CALIC [13] are briefly introduced in Section 1. The bitrate for each day is the average bitrate of 100 aurora spectral images evenly selected from the data of each day, and the smaller the bitrate, the better the compression performance. It can be seen that the bitrate of the online DPCM is the lowest for 6 days of the 9 days, and the average bitrate of the online DPCM on these 9 days is also lower than that of all the other algorithms. Therefore, we can conclude that the online DPCM outperforms all the other algorithms shown in Figure 17.

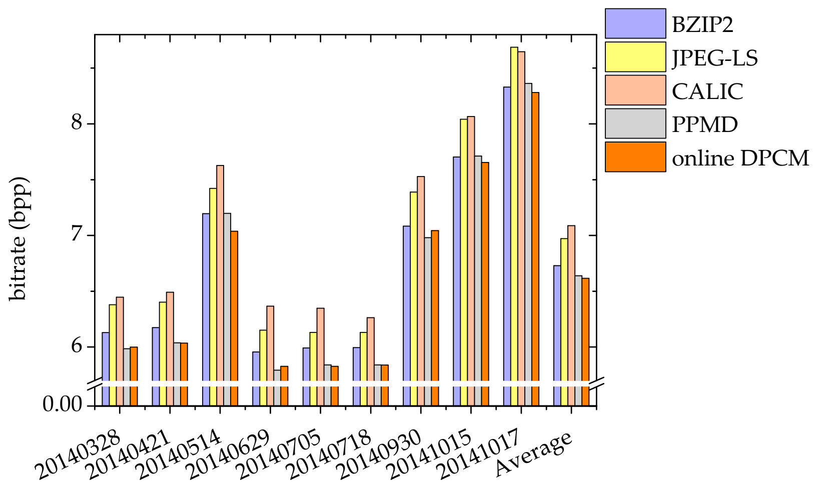


Figure 17. Average bitrate of the online DPCM (Differential Pulse Code Modulation) algorithm and several other lossless compression algorithms on the aurora spectral data of 9 days. The average represents the average bitrate of each algorithm on these 9 days.

4.2. The Performance of the Parallel Implementation of the Online DPCM Algorithm

For a CUDA program, what we are most concerned about is the speed comparison between it and its corresponding serial program. Table 1 shows the execution time for the parallel implementation and two serial implementations of the online DPCM algorithm. The time for each day is actually the average time for 100 aurora spectral images evenly selected from the data of each day. The purpose of the "serial decomposed" implementation is to apply the idea of decomposition introduced in Section 3 to the matrix multiplication $C^T \times C$ and the matrix-vector multiplication $C^T \times \vec{b}$ (as shown in Equation (2), and the purpose of the "serial direct" implementation is to directly calculate the matrix multiplication and the matrix-vector multiplication without considering the redundant computations. Both of these serial implementations are tested on the Intel Xeon E3-1240 v3 CPU. It can be seen that the execution time of the serial direct implementation is about 870 s, which is much larger than the execution time of

3.74 s of the serial decomposed implementation. Therefore, the decomposition method introduced in Section 3 is very effective, which can avoid redundant calculations and save a lot of time. The parallel implementation is the method described in detail in Section 3 and is tested on the TITAN X GPU. From Table 1, the execution time of the parallel implementation is about 0.21 s, which is much less than the 15 s aurora data acquisition time interval, and therefore can save a lot of time for other operations on each aurora spectral image.

Table 1. The execution time for three different implementations of the online DPCM algorithm. The speedup is the ratio of the serial decomposed time to the parallel time.

Implementation / Data	Serial Direct Time (s)	Serial Decomposed Time (s)	Parallel Time (s)	Speedup
20140328	869.16	3.75	0.19	19.7
20140404	872.56	3.74	0.21	17.8
20140425	872.28	3.74	0.21	17.8
20140629	872.24	3.74	0.21	17.8
20140705	872.12	3.74	0.21	17.8
20140718	869.67	3.74	0.21	17.8
20140730	870.48	3.73	0.21	17.8
20140827	872.68	3.74	0.21	17.8

The parallel time presented in Table 1 is the total execution time of the parallel application, including the execution time of all the kernel functions and the time of the data transfers between host and device. Furthermore, Table 2 shows the time and ratio of three CUDA kernel functions and two data transfers. C^T*CMulKernel* is the kernel function shown in Figure 15, C^T*CBlockPrefixKernel* (shown in Figures 12 and 13) is the kernel function for computing the prefix sum of an array in a thread block, and *matrixInverseKernel* (shown in Figure 16) is the kernel function for computing the inverse matrix of matrix C^TC; *memcpyHtoD* is the data transfer from host to device, and the only data that needs to be transferred from host to device is the aurora spectral data. Additionally, *memcpyDtoH* is the data transfer from device to host, that is, the transfer of the residual from device to host. It can be seen that the *matrixInverseKernel* takes the most time, the total time spent by these three kernel functions is more than 84%, and the data transfer between host and device only takes about 0.2% of the total time. Therefore, we can say that the execution of the kernel functions takes almost all the time of the application.

Table 2. The time and ratio of three CUDA kernel functions and two data transfers on two aurora images 20140514_133241 and 20140923_155500.

	20140514_133241		20140923_155500	
	Time	**Ratio (%)**	**Time**	**Ratio (%)**
C^T*CMulKernel*	10.820 ms	5.71	10.759 ms	5.71
C^T*CBlockPrefixKernel*	11.920 ms	6.29	11.857 ms	6.29
matrixInverseKernel	137.29 ms	72.41	136.42 ms	72.36
memcpyHtoD	191.24 us	0.10	194.25 us	0.10
memcpyDtoH	161.32 us	0.09	161.32 us	0.09

In addition to speed, there are also some other metrics to measure the performance of a CUDA program, such as the *achieved_occupancy* and *gld_throughput*, both of which are provided by the CUDA command line profiler, called *nvprof*. The *achieved_occupancy* is used to measure the occupancy of Streaming Multiprocessors (SMs), and the *gld_throughput* is the global memory load throughput. Table 3 shows the performance of the three kernel functions C^T*CMulKernel*, C^T*CBlockPrefixKernel*, and *matrixInverseKernel* on these two metrics. It can be seen that the *achieved_occupancy* of these three kernel functions is very high, and the *achieved_occupancy* of *matrixInverseKernel* is 99.63%. As for the *gld_throughput*, the throughput of the C^T*CBlockPrefixKernel* is more than 300 GB/s, the throughput of *matrixInverseKernel* is more than 180 GB/s, and the throughput of C^T*CMulKernel* is about 5 GB/s. This is because in the C^T*CMulKernel* (shown in Figure 15), each thread block is responsible for the multiplication of $C^T{}_{i,n}$ and $C_{i,n}$, and the consecutive $N + M - 1$ elements (shown in Figure 5) that make up the $C_{i,n}$ are copied to shared memory. As a result, the access of the global memory is greatly reduced.

Table 3. The *achieved_occupancy* and *gld_throughput* of the three kernel functions on two aurora images 20140514_133241 and 20140923_155500.

	20140514_133241		**20140923_155500**	
	achieved_occupancy	*gld_throughput*	*achieved_occupancy*	*gld_throughput*
C^T*CMulKernel*	87.06%	5.43 GB/s	87.24%	4.61 GB/s
C^T*CBlockPrefixKernel*	94.88%	321.91 GB/s	94.88%	310.63 GB/s
matrixInverseKernel	99.63%	183.53 GB/s	99.63%	183.21 GB/s

4.3. The Parallel Implementation of the Online DPCM Algorithm using the Multi-Stream Technique

A CUDA stream is a series of CUDA operations that run on the device in the order in which they are launched by the host code. By default, there is only one CUDA stream, and all the data is processed in this stream. If we want to use multiple streams, we must explicitly create multiple streams, divide all the data to be processed into several small parts, and distribute each part to a stream. The advantage of using multiple streams is that the execution order of CUDA operations in different streams is arbitrary, which makes it possible to overlap the execution of CUDA operations in different streams, and therefore, reduce the execution time of the entire application. Figure 18 is an example that more clearly describes the overlap of CUDA operations in different streams. There are 4 streams, and each stream has only one kernel function. The HtoD represents the data transfer from host to device, and DtoH represents the data transfer from device to host. It can be seen that there are three types of overlaps: the overlapping of data transfer and kernel execution, the overlapping of kernel executions, and the overlapping of data transfers in different directions.

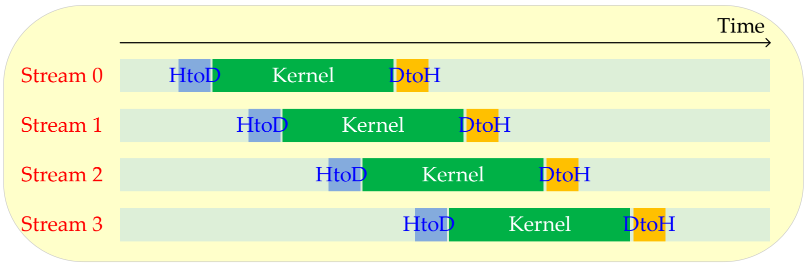


Figure 18. An example timeline of a CUDA application with 4 streams.

For the optimization of the online DPCM algorithm using the multi-stream technique, the purpose of our scheme is to evenly divide an aurora image by columns and assign each part to a stream, which is because the prediction of a pixel in an aurora image involves all the rows in front of it, as shown in Figure 5. The method for dividing an aurora image into 4 streams is shown in Figure 19, and if there are 2 streams, the division method is similar. Figure 20 shows the CUDA codes of the multi-stream technique. All of the CUDA kernels, including the C^T*CMulKernel*, the C^T*CBlockPrefixKernel*, the *matrixInverseKernel*, and so on, are equivalent to one kernel, whose input is the aurora data and output is the residual. Note that the aurora data is copied to the device using the synchronous function cudaMemcpy(), which will block the host until the data transfer is finished, rather than the asynchronous function cudaMemcpyAsync(), which allows each stream to copy the data assigned to it from host to device asynchronously. This is because the aurora data is stored by rows but assigned to each stream by columns, so each stream cannot copy the aurora data assigned to the device by calling cudaMemcpyAsync() once, unless the aurora data assigned to each stream is prepared in advance on the host, but this requires extra time. The reason for copying the residual using cudaMemcpy() is similar.

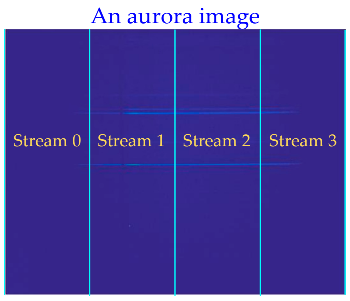


Figure 19. The method for dividing an aurora image to four streams.

```
//NSTREAMS is the number of CUDA streams
cudaStream_t *streams = (cudaStream_t*)malloc(sizeof(cudaStream_t)*NSTREAMS);
//create multiple streams
for (int i = 0; i < NSTREAMS; i++)
        cudaStreamCreate(&streams[i]);
unsigned short *d_auroraData;              //DATASIZE is the number of pixels in an aurora image
cudaMalloc((void**)&d_auroraData, sizeof(unsigned short)*DATASIZE);
short *d_residual;                         //the device memory to store the residuals
cudaMalloc((void**)&d_residual, sizeof(short)*DATASIZE);
//auroraData is the host memory that stores the aurora data
cudaMemcpy(d_auroraData, auroraData, sizeof(unsigned short)*DATASIZE, cudaMemcpyHostToDevice);
for (int i = 0; i < NSTREAMS; i++){
        //the equivalent kernel of all the kernel functions to compute the residuals
        kernel <<<grid, block, size of dynamic shared memory in bytes, streams[i] >>>(d_auroraData, d_residual);
}
//copy the residuals from device to host
cudaMemcpy(residual, d_residual, sizeof(short)*DATASIZE, cudaMemcpyDeviceToHost);
```

Figure 20. The CUDA codes of the online DPCM algorithm using multi-stream technique.

Figure 21 shows the execution time of the parallel implementation of the online DPCM algorithm using two streams and four streams. The time for each day is the average time for 100 aurora spectral images evenly selected from the data of each day. The speed of using the default stream has been shown in Table 1, and here the aim is to make a comparison with the speed using multiple streams. It can be seen that the speed of these three implementations is basically the same, which is because the occupancy of the SM of the three kernels C^T*CMulKernel*, C^T*CBlockPrefixKernel*, and *matrixInverseKernel*, which accounts for more than 84% of the total time (Table 2), is very high (Table 3), and the occupancy of the SM of other kernels is actually also very high. In a multi-stream program, if a kernel function in one stream has been started, the kernel function in another stream can be started only when there are enough available SMs, otherwise, there will be no overlap of kernel functions, and thus cannot contribute to reducing the total execution time.

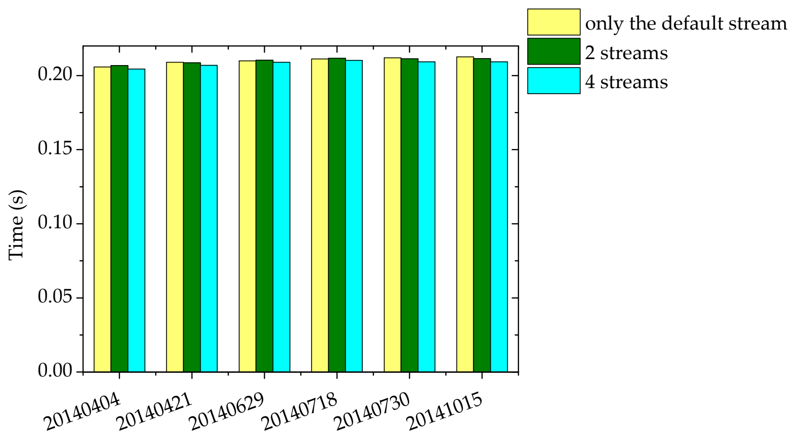


Figure 21. The comparison in speed between the parallel implementation of the online DPCM algorithm using two streams, four streams, and the default stream.

4.4. The Parallel Implementation of the Online DPCM Algorithm using Multi-GPU Technique

To calculate with multiple GPUs, a programmer needs to subdivide a large calculation problem into several small subproblems and distribute each subproblem to a GPU. All of the GPUs run concurrently, and as a result, the total execution time is reduced. When implementing the online DPCM algorithm using multiple GPUs, the method for subdividing the calculation task is the same as that when using multiple streams, that is, evenly divide the aurora image by columns and distribute each part to a GPU. However, each GPU has its own memory, unlike in multi-stream programming, where all the streams use the same GPU's memory. Therefore, the aurora data cannot be transmitted to the device by calling the cudaMemcpy() once, as in Figure 20; we must transmit the aurora data assigned to each GPU to the memory of each GPU using the cudaMemcpyAsync(), and the purpose of the preparation before the transmission is to prepare the data for each GPU in advance on the host, though this requires extra time. In addition, the residual computed by each GPU is stored on the memory of each GPU, so we must transfer it to the host using cudaMemcpyAsync() as well, rather than cudaMemcpy(). Some core CUDA codes of the multi-GPU implementation are presented in Figure 22.

The first part in Figure 22 is to prepare the data for each GPU. The *nColsPerGPU* is the number of columns that are assigned to each GPU. The *dataCols* is the number of columns that each GPU needs to predict the *nColsPerGPU* columns of pixels, and this is $nColsPerGPU + N + M - 1$, except for GPU 0. This is because each pixel requires $N + M - 1$ previous columns to predict it, as shown in Figure 5, but there is no data before the data assigned to GPU 0, and the initial several columns of data of GPU 0 are predicted using the actual available data. The cudaMallocHost() is a CUDA function to allocate pinned host memory, and the reason for allocating pinned host memory for *h_auroraData*[*i*] is that cudaMemcpyAsync() requires the host memory to be pinned memory. The second part is to allocate device memory to store the aurora data and residual for each GPU, and create a stream for each GPU. In the third part, first, the aurora data required by each GPU is asynchronously transmitted from the pinned host memory *h_auroraData*[*i*] to each device. Then, the kernel function is launched to calculate the residual, which is actually the equivalent of all the kernels, including the $C^TCMulKernel$, the $C^TCBlockPrefixKernel$, the *matrixInverseKernel*, and so on. Finally, the residual is copied asynchronously from each device to the pinned host memory *h_residual*.

The speed of the multi-GPU implementation using 2 GPUs and 4 GPUs is shown in Table 4. The time for each day is the average time for 100 aurora spectral images evenly selected from the data of each day. It can be seen that the execution time is about 0.12 s when using 2 GPUs, and when the number of GPUs increases to 4, the execution time is between 0.06 s and 0.07 s, which is much less than the 15 s aurora data acquisition time interval. The speedup is the ratio of the serial time to the 2 GPUs time and the 4 GPUs time, respectively.

```
// 1. prepare the aurora data needed by each GPU on the host
//NGPUS is the number of available GPUs
unsigned short**h_auroraData = (unsigned short**)malloc(sizeof(unsigned short*)*NGPUS);
//nColsPerGPU is the number of columns that assigned to each GPU to compute the residual
int nColsPerGPU = WIDTH / NGPUS;      //WIDTH is the number of columns of an aurora image
for (int i = 0; i < NGPUS; i++){
        //all the GPUs except GPU 0 need extra N+M-1 columns of data to process the first column assigned to it
        int dataCols = (i == 0) ? nColsPerGPU : nColsPerGPU + N + M - 1;
        //allocate pinned host memory to store the data for each GPU
        //HEIGHT is the number of rows of an aurora image
        cudaMallocHost((void**)&h_auroraData[i], sizeof(unsigned short)*dataCols*HEIGHT);
        //copy the corresponding aurora data to h_auroraData[i]
        /*......*/
}
// 2. allocate device memory and create CUDA streams
unsigned short**d_auroraData = (unsigned short**)malloc(sizeof(unsigned short*)*NGPUS);
short**d_residual = (short**)malloc(sizeof(short*)*NGPUS);
cudaStream_t *streams = (cudaStream_t*)malloc(sizeof(cudaStream_t)*NGPUS);
for (int i = 0; i < NGPUS; i++){
        cudaSetDevice(i);                    //set current device
        int dataCols = (i == 0) ? nColsPerGPU : nColsPerGPU + N + M - 1;
        cudaMalloc((void**)&d_auroraData[i], sizeof(unsigned short)*dataCols*HEIGHT);
        cudaMalloc((void**)&d_residual[i], sizeof(short)*nColsPerGPU*HEIGHT);
        cudaStreamCreate(&streams[i]);       //create multiple streams
}
short *h_residual;                           //allocate a pinned memory for asynchronous transfer of residual
cudaMallocHost((void**)&h_residual, sizeof(short)*WIDTH*HEIGHT);
// 3. launch the kernels and asynchronous data transfer
for (int i = 0; i < NGPUS; i++){
        cudaSetDevice(i);
        int dataCols = (i == 0) ? nColsPerGPU : nColsPerGPU + N + M - 1;
        //asynchronous transfer of aurora data from the pinned host memory to each device
        cudaMemcpyAsync(d_auroraData[i], h_auroraData[i], sizeof(unsigned short)*dataCols*HEIGHT,
cudaMemcpyHostToDevice, streams[i]);
        //all the kernel functions are equivalent to this one kernel function
        kernel <<<grid, block, size of dynamic shared memory in bytes, streams[i] >>>(d_auroraData[i], d_residual[i]);
        //asynchronous transfer of residual from each device to the pinned host memory
        cudaMemcpyAsync(h_residual + i*nColsPerGPU*HEIGHT, d_residual[i], sizeof(short)*nColsPerGPU*HEIGHT,
cudaMemcpyDeviceToHost, streams[i]);
}
// 4. synchronize streams
for (int i = 0; i < NGPUS; i++){
        cudaSetDevice(i);
        cudaStreamSynchronize(streams[i]);
}
```

Figure 22. The core CUDA codes for the multi-GPU implementation of the online DPCM algorithm.

Table 4. The execution time for the serial and two multi-GPU (Graphics Processing Unit) implementations of the online DPCM algorithm.

		20140514	**20140705**	**20140827**	**20140916**	**20140930**	**20141017**
Serial time(s)		3.74	3.74	3.74	3.74	3.74	3.74
2 GPUs	Time(s)	0.12	0.12	0.12	0.12	0.12	0.12
	Speedup	31.2	31.2	31.2	31.2	31.2	31.2
4 GPUs	Time(s)	0.06	0.06	0.07	0.07	0.07	0.07
	Speedup	62.3	62.3	53.4	53.4	53.4	53.4

Furthermore, the speed comparison between using one GPU, 2 GPUs, and 4 GPUs is depicted in Figure 23. The performance of using one GPU has been shown in Section 4.2 and here it is compared with using multiple GPUs. It can be seen that when the number of GPUs is doubled, the time is reduced by about half. This is because when the number of GPUs is doubled, the amount of data processed by each GPU is reduced by half. Note that the time is not exactly cut in half, but always slightly more than a half. This is because as the number of GPUs increases, the overheads increase too, such as the device startup overhead and the kernel launch overhead.

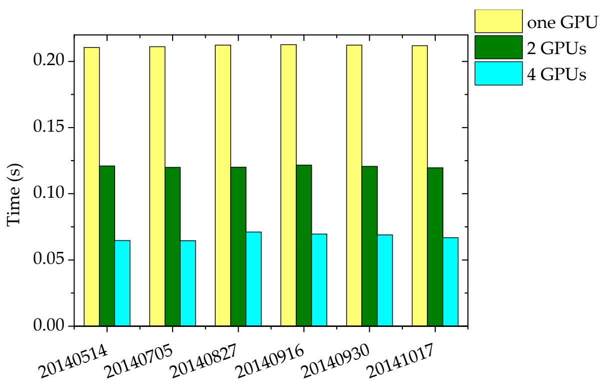


Figure 23. The comparison in speed between the parallel implementation of the online DPCM algorithm using 2 GPUs, 4 GPUs, and only one GPU.

5. Conclusions

This paper introduces the parallel CUDA implementation of the prediction-based online DPCM algorithm for the lossless compression of the aurora spectral data. The first step of the online DPCM method is to compute the prediction coefficients using the least square method, then use the prediction coefficients to compute the predicted value of each pixel, subtract the predicted value from the original value to get the residual, and finally, encode the residual. In order to improve the compression performance of the online DPCM algorithm, we proposed two improvements. The first is to build multiple equations using each of the rows before the current pixel when computing the prediction coefficients, and the other is to apply a dual-threshold method to the encoding of the residual. In the CUDA implementation of the online DPCM, we focus on the multiplication of matrices C^T and C and the inversion of the matrix C^TC. A decomposition method, which decompose the multiplication of C^T and C into the addition of many small matrices, is proposed to remove redundant data access and calculations. In addition, the multi-stream technique and the multi-GPU technique are used to optimize the CUDA implementation. Finally, the average compression time for an aurora spectral image is about 0.06 s when using 4 GPUs.

Since aurorae have the characteristic of instantaneous change and are captured continuously, several consecutive aurora spectral images in time direction should have some correlation. Therefore, in the future work, we plan to apply 3-D methods to aurora spectral image compression to utilize the correlations in time direction, which is impossible for 2-D methods. Although the 3-D methods are more likely to cause transmission errors than 2-D methods, as illustrated in Section 1, we can decorrelate these by grouping only a few aurora spectral images, such as 3–6 frames, to minimize transmission errors. We believe that the improved compression gains justify the increased transmission errors.

Author Contributions: Conceptualization, J.W. and J.L.; Methodology, J.L.; Software, J.L.; Validation, J.L., J.W. and G.J.; Formal Analysis, J.L.; Investigation, J.L.; Resources, J.W.; Data Curation, J.W.; Writing-Original Draft Preparation, J.L.; Writing-Review & Editing, J.W.; Visualization, G.J.; Supervision, J.W.; Project Administration, J.W.; Funding Acquisition, J.W.

Funding: This research was funded by National Natural Science Foundation of China, numbers 61775175 and 61771378.

Acknowledgments: We thank all the editors and reviewers for their valuable comments that greatly improved the presentation of this paper. We thank the Polar Research Institute of China for providing the aurora spectral data.

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

References

1. Kong, W.; Wu, J. Fast DPCM scheme for lossless compression of aurora spectral images. In Proceedings of the High-Performance Computing in Geoscience and Remote Sensing VI, Edinburgh, UK, 26 September 2016; p. 100070M. [CrossRef]
2. Kong, W.; Wu, J.; Hu, Z.; Anisetti, M.; Damiani, E.; Jeon, G. Lossless compression for aurora spectral images using fast online bi-dimensional decorrelation method. *Inf. Sci.* **2017**, *381*, 33–45. [CrossRef]
3. Wells, D.C.; Greisen, E.W.; Harten, R.H. FITS—A flexible image transport system. *Astron. Astrophys. Suppl.* **1981**, *44*, 363.
4. Aiazzi, B.; Alparone, L.; Baronti, S. Near-lossless image compression by relaxation-labelled prediction. *Signal Process.* **2002**, *82*, 1619–1631. [CrossRef]
5. Aiazzi, B.; Alparone, L.; Baronti, S. Fuzzy logic-based matching pursuits for lossless predictive coding of still images. *IEEE Trans. Fuzzy Syst.* **2002**, *10*, 473–483. [CrossRef]
6. Aiazzi, B.; Alparone, L.; Baronti, S. Context modeling for near-lossless image coding. *IEEE Signal Process. Lett.* **2002**, *9*, 77–80. [CrossRef]
7. Mielikainen, J.; Toivanen, P. Clustered dpcm for the lossless compression of hyperspectral images. *IEEE Trans. Geosci. Remote Sens.* **2003**, *41*, 2943–2946. [CrossRef]
8. Martin, G. Range encoding: An algorithm for removing redundancy from a digitised message. In Proceedings of the Video and Data Recording Conference, Southampton, UK, 24–27 July 1979.
9. Kong, W.; Wu, J. A lossless compression algorithm for aurora spectral data using online regression prediction. In Proceedings of the High-Performance Computing in Remote Sensing V, Toulouse, France, 21 September 2015; p. 964611. [CrossRef]
10. Weinberger, M.J.; Seroussi, G.; Sapiro, G. LOCO-I: A low complexity, context-based, lossless image compression algorithm. In Proceedings of the Data Compression Conference-DCC 96, Snowbird, UT, USA, 31 March–3 April 1996; pp. 140–149. [CrossRef]
11. Weinberger, M.J.; Seroussi, G.; Sapiro, G. The LOCO-I lossless image compression algorithm: Principles and standardization into JPEG-LS. *IEEE Trans. Image Process.* **2000**, *9*, 1309–1324. [CrossRef]
12. Taubman, D.S.; Marcellin, M.W. JPEG2000: Standard for interactive imaging. *Proc. IEEE* **2002**, *90*, 1336–1357. [CrossRef]
13. Wu, X.; Memon, N. Context-based, adaptive, lossless image coding. *IEEE Trans. Commun.* **1997**, *45*, 437–444. [CrossRef]
14. Kim, B.J.; Pearlman, W.A. An Embedded Wavelet Video Coder Using Three-Dimensional Set Partitioning in Hierarchical Trees (SPIHT). In Proceedings of the Conference on Data Compression, Snowbird, UT, USA, 25–27 March 1997; p. 251. [CrossRef]
15. Lucas, L.F.R.; Rodrigues, N.M.M.; Cruz, L.A.d.S.; Faria, S.M.M.d. Lossless Compression of Medical Images Using 3-D Predictors. *IEEE Trans. Med Imaging* **2017**, *36*, 2250–2260. [CrossRef]
16. Xiaolin, W.; Memon, N. Context-based lossless interband compression-extending CALIC. *IEEE Trans. Image Process.* **2000**, *9*, 994–1001. [CrossRef]
17. Tang, X.; Pearlman, W.A. Three-dimensional wavelet-based compression of hyperspectral images. In *Hyperspectral Data Compression*; Springer: Boston, MA, USA, 2006; pp. 273–308. [CrossRef]
18. Zhang, J.; Fowler, J.E.; Liu, G. Lossy-to-Lossless Compression of Hyperspectral Imagery Using Three-Dimensional TCE and an Integer KLT. *IEEE Geosci. Remote Sens. Lett.* **2008**, *5*, 814–818. [CrossRef]
19. Karami, A.; Beheshti, S.; Yazdi, M. Hyperspectral image compression using 3D discrete cosine transform and support vector machine learning. In Proceedings of the 2012 11th International Conference on Information Science, Signal Processing and their Applications (ISSPA), Montreal, QC, Canada, 2–5 July 2012; pp. 809–812. [CrossRef]
20. Aiazzi, B.; Alparone, L.; Baronti, S.; Lastri, C. Crisp and Fuzzy Adaptive Spectral Predictions for Lossless and Near-Lossless Compression of Hyperspectral Imagery. *IEEE Geosci. Remote Sens. Lett.* **2007**, *4*, 532–536. [CrossRef]
21. Simek, V.; Asn, R.R. GPU Acceleration of 2D-DWT Image Compression in MATLAB with CUDA. In Proceedings of the 2008 Second UKSIM European Symposium on Computer Modeling and Simulation, Liverpool, UK, 8–10 September 2008; pp. 274–277. [CrossRef]

22. Tenllado, C.; Setoain, J.; Prieto, M.; Piñuel, L.; Tirado, F. Parallel Implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter Bank versus Lifting. *IEEE Trans. Parallel Distrib. Syst.* **2008**, *19*, 299–310. [CrossRef]
23. Jozef, Z.; Matus, C.; Michal, H. Graphics processing unit implementation of JPEG2000 for hyperspectral image compression. *J. Appl. Remote Sens.* **2012**, *6*, 011507. [CrossRef]
24. Santos, L.; Magli, E.; Vitulli, R.; López, J.F.; Sarmiento, R. Highly-Parallel GPU Architecture for Lossy Hyperspectral Image Compression. *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.* **2013**, *6*, 670–681. [CrossRef]
25. Keymeulen, D.; Aranki, N.; Hopson, B.; Kiely, A.; Klimesh, M.; Benkrid, K. GPU lossless hyperspectral data compression system for space applications. In Proceedings of the 2012 IEEE Aerospace Conference, Big Sky, MT, USA, 3–10 March 2012; pp. 1–9. [CrossRef]
26. Wu, Z.; Shi, L.; Li, J.; Wang, Q.; Sun, L.; Wei, Z.; Plaza, J.; Plaza, A. GPU Parallel Implementation of Spatially Adaptive Hyperspectral Image Classification. *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.* **2018**, *11*, 1131–1143. [CrossRef]
27. CUBLAS_Library. Available online: https://docs.nvidia.com/cuda/cublas/#axzz4h7431Tsi (accessed on 22 May 2019).
28. CUDA_C_Programming_Guide. Available online: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4dQa7XYMd (accessed on 22 May 2019).
29. Juanjuan, B.; Ling, G.; Lidong, H. Constructing windows+gcc+mpi+omp and performance testing with Gauss-Jordan elimination method in finding the inverse of a matrix. In Proceedings of the 2010 International Conference On Computer Design and Applications, Qinhuangdao, China, 25–27 June 2010. [CrossRef]
30. Nyokabi, G.J.; Salleh, M.; Mohamad, I. NTRU inverse polynomial algorithm based on circulant matrices using gauss-jordan elimination. In Proceedings of the 2017 6th ICT International Student Project Conference (ICT-ISPC), Skudai, Malaysia, 23–24 May 2017; pp. 1–5. [CrossRef]
31. 7-Zip. Available online: https://www.7-zip.org/ (accessed on 22 May 2019).
32. Burrows, M.; Wheeler, D.J. A Block-Sorting Lossless Data Compression Algorithm. *SRC Res. Rep.* **1994**.
33. Shkarin, D. PPM: One step to practicality. In Proceedings of the DCC 2002, Data Compression Conference, Snowbird, UT, USA, 2–4 April 2002; pp. 202–211. [CrossRef]
34. Cleary, J.; Witten, I. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Trans. Commun.* **1984**, *32*, 396–402. [CrossRef]