The application of OpenCL to accelerate the lossless image compression algorithm based on cascading fragmentation and pixels sequence ordering

A. Khokhlachev¹, V. Smirnov¹, A. Korobeynikov¹

¹Kalashnikov Izhevsk State Technical University, Studentcheskaya 7, 426069, Izhevsk, Russia

Abstract

The previous papers of the authors offer approach to building the ordered sequence of image pixels at lossless compression, which comprises methods of cascading fragmentation and the use of bypasses code book. For fragment sized 6*6 the code book contains 22144 various bypasses, the cost of coding to be estimated for every one of them. The search of optimal bypass is an exhaustive search type. The present paper describes ways of increasing the image lossless compression rate by using parallel computation based on OpenCL. Algorithm functions with great runtime were changed in order to transfer calculations to OpenCL using GPU/CPU. The acceleration degree for different algorithm functions gained in experiments amounted to 3..32.

Keywords: lossless image compression; cascading fragmentation; pixels sequence ordering; optimal bypass; code book; computational acceleration; parallel computing; open computing language (OpenCL); graphics processing unit (GPU); central processing unit (CPU); Haar integral-valued wavelet transformation; interchannel decorrelation

1. Introduction

At the moment there exist both a large number of compression algorithms of particular data classes and universal compression algorithms. This work will address the lossless image compression algorithm based on optimization of bypass image being developed by the authors and described in [1...3]. When processing test images [7], the algorithm gives the average compression ratio of 1.54, which matches the analogues [5]. Let us consider test results by groups of images: 1) in group «2.1.*.tiff» by 1.426 2) in group «2.2.*.tiff» by 1.547 3) in group «4.1.*.tiff» by 1.622 4) in group «4.2.*.tiff» by 1.522 [5]. In addition, the algorithm has some other advantages [5].

To achieve a high compression ratio it is necessary to use a number of demanding algorithm functions, which leads to longer image processing program runtime. Presently, parallel computing is there. The aim of this work is to apply OpenCL to speed up the lossless compression algorithm. To achieve this goal it is necessary to: analyze duration of program execution; find the algorithm functions with time-consuming calculations; consider transfer of these functions to GPU. Image processing performed in the algorithm is based on handling particular fragments, therefore, in general case, such tasks can be carried out simultaneously. Furthermore, it is possible to perform the preprocessing functions for image fragments in parallel as well.

2. Basic algorithm

The basic algorithm inherently consists in cascading fragmentation of image [1], the search of the fragment optimal bypass (path) [2], and dynamic programming of pixels delta-code at fragment bypass [6]. After encoding, the obtained data is further compressed by Deflate algorithm using standard libraries. The compression ratio depends on the class of the image being compressed, and on average equals 1.54 for the array of test images [5].

The runtime of image compression program depends on the processed image size. Due to a number of algorithmic solutions such as cascading fragmentation, and the use of bypasses codebook instead of calculating the possible bypasses for each image fragment, the runtime was reduced. However, the image compression duration is still high enough [5]: 1) in group «2.1.*.tiff» - 101 seconds 2) in group «2.2.*.tiff» - 404 seconds 3) in group «4.1.*.tiff» - 24 seconds 4) in group «4.2.*.tiff» - 141 seconds.

In computational terms the most complex of the basic algorithm functions is to estimate the encoding cost for all possible bypasses. Meanwhile, this algorithm function is suitable for parallelization, since the optimal bypass choice uses exhaustive search of obtained cost estimates. For a fragment sized 6*6 the total bypasses number from the upper left corner is 22144.

To use all multi-core CPU resources it is necessary to effectively implement parallelization of functions between all cores. The basic program features parallel execution of optimal fragment bypass search cycle done with .Net Framework standard classes (SSE instructions). It is possible to use a more powerful CPU, but even in this case, the speed increase will not be significant.

In recent years the increasing number of programs with parallel data processing use GPU computing [7]. This is dictated by a growing gap in performance between CPU and GPU.

Taking into account the above said, it was decided to move part of the compression algorithm functions to GPU. Obviously, this will require some significant changes in the functions, but it will allow for significant decrease in the program runtime without changing the basic algorithm.

Currently there are several approaches to programs execution on GPU. OpenCL is an open standard [8], which can execute programs on both CPU and GPU of different manufacturers. Therefore, in this research, to speed the algorithm, OpenCL was chosen.
At the moment there exist quite a big number of various compression algorithms in general and algorithms for images in particular. Images compressed both as lossy and lossless are widely and effectively used. For example, lossless compression is used in PNG files where the actual compression is implemented with Deflate algorithm [9, 10], which is a combination of LZ and Huffman algorithms in its turn. There are no free turn-key programs available for lossless image compression making use of OpenCL. WinZip is an example of the lossless compression program based on universal algorithm and using OpenCL, which provides for performance increase of about 45% [11].

In addition to the basic algorithm, image preprocessing was implemented which was described in the authors’ previous works: interchannel decorrelation of image color layers [12] and the transformation of pixels matrix based on integer-valued Haar wavelets [13]. These functions can be easily threaded for the implementation on OpenCL.

3. Methods of acceleration

The general problem solved in present research is changing the compression software in order to transfer part of calculations to OpenCL. Image compression algorithm is shown in Fig. 1.

3.1. Preparation of fragments

The function receives separate color layers of an image. The function output is arrays of separate fragments of fixed size. Pixel values of the fragment nodes beyond the image borders are virtual pixels and the values of these pixels are set as constant (white pixels on Fig. 2). The top left pixels of each fragment on level 0 constitute the fragments on level 1 and so on, as long as the fragments number on a level is more than one. Data structure passed to the OpenCL kernel represents the matrix of image values, the output structure is the array of separate fragments [1].

3.2. Preprocessing

3.2.1. Interchannel decorrelation of color layers

This function is designed to calculate the interchannel decorrelation between the groups of color channels (layers) of the original image and to find the best variant to group them [12].

When function is started the arrays containing pixels values of all color channels of the fragment, and also the number of channels have to be conveyed (Fig. 3). In addition, data on the possible grouping of channels is needed.

Formula for calculating interchannel decorrelation for arbitrary channels number based on the mean and interchannel differences is applied [12]:

$$P_i = \text{Round} \left( \frac{\sum_{k=0}^{n} X_i^k}{n} \right)$$

$$P_i = X_i^i - X_i^i, i = 1, n$$

(1)
where Round is the rounding operation to the nearest integer; $X$ – pixels value on each of the channels; $k$ – channel index; $n$ – the number of processed channels.

The color channels can be independent from each other, therefore, the grouping variant with a minimum encoding costs estimate has to be selected. It is necessary to implement the decorrelation formulas for all dependent channels groups. The minimum channels number in the group is 2. If the image consists of 3 channels (24 bits per pixel), we get the following grouping variants:

$$
(X^{1}X^{2}X^{3})X^{1}(X^{2}X^{3})X^{2}(X^{1}X^{3})X^{3}(X^{1}X^{2})X^{1}X^{2}X^{3}
$$

(2)

where decorrelation formulas are to be applied to groups of channels in parenthesis.

The calculation of decorrelation is performed for all possible groups ($g=1..G$). The result is the index of $g$ grouping:

$$
g = \arg \min_g \left( \sum_{P_{ij}} \left( \sum_{k=1}^{n} \sum_{j=1}^{k} \text{Cost}(P_{ij}) \right) \right)
$$

where $P_{ij}$ – is the pixel value after the interchannel decorrelation for the grouping index $g$; $i$ – channel index; $j$ – pixel index; $n$ – number of channels; $k$– number of pixels number in the fragment.

$\text{Cost}$ is the a some estimation function of encoding costs estimate, for example, the length of the Fibonacci code which encoding the value $P_{ij}$ value, or the estimated length of binary coding:

$$
\log_2 |P_{ij} + 1| + 1
$$

(4)

3.2.2. Transformation based on Haar integer-valued wavelet: Haar

This function is designed intended for cascading processing of each image fragment by applying an integer-valued version of the Haar wavelet transform [13].

The function receives an array containing a single channel of image and the number of the processed fragments in by width and height which need to processed.

In the course of the algorithm execution it uses additional arrays for each executable OpenCL kernel with the size equal-too the one fragment each are used, for storing the intermediate results of the cascading conversion.
The function result is an array of the size equal to the original image. Image matrix should be divided into blocks of size $2^2$. Then calculated the values for $a, h, v, d$ are to be found by the formula [13]:

$$
\begin{align*}
    c_2 &= x_1 - x_2 \\
    c_3 &= x_1 - x_3 \\
    c_4 &= x_1 - x_4 \\
    c_t &= x_1 - \text{Round} \left( \sum_{l=2}^{4} c_l \right) = x_1 - x_l = \frac{1}{2} \sum_{i=1}^{4} x_l \\
    a &= c_1 \\
    h &= \text{Round}(d/2) + c_3 \\
    v &= \text{Round}(d/2) + c_2 \\
    d &= c_3 - c_4 + c_2
\end{align*}
$$

Where $\text{Round}$ is the rounding operation to the nearest integer, $x_l$ – original pixels of the block. The obtained values of $a, h, v, d$ should have to be recorded in positions of the matrix, as shown in Fig. 4. Calculation should be carried out as multiple-resolution in multiple scale, by repeating the transform on the blocks consisting of grouped values $a', h', v', d'$, each time and reducing their size in a timely half for in each coordinate every time, as long as it is possible to form a block from $d'$ values on a subsequent scale. Cascading transform will stop then the block $a', h', v', d'$ is absent.

It should be noted that when with the fragment sized $2^2$, it is possible to use preprocessing (interchannel decorrelation, and Haar transform) after the function of dividing on the fragments. In this case, the Haar transformation is possible only within the same fragment. With this approach, the fragment encoding is completely independently of the other fragments and therefore the decoding is possible for a single fragment.

3.3. Search for optimal bypass of fragment

The function receives the fragments obtained after preprocessing functions (integer-valued Haar transform, interchannel decorrelation of color layers).

3.3.1. Calculation of encoding cost for all possible bypasses of fragment

3.3.1.1. Possible ways to bypass of a fragment

Encoding and decoding algorithms have information about all the possible bypasses for a given fragment size, is known for encoding and decoding algorithms.

All bypasses (paths) have been calculated in advance and constitute stored in the bypasses codebook [2].

Therefore, in encoding and decoding only need to know the bypass index, but not the edges of bypass (path), is the only prerequisite for encoding and decoding, but not the edges of bypass (path).

3.3.1.2. Computing the delta-code of bypasses edges

This function is designed to calculate the difference of values for all pairs of nodes that make up the edge on a given fragment bypass [2]. In the course of the function the algorithm uses the previously prepared fragments. The result is a list of arrays consisting of the delta-code of all edges for each fragment.

$$
\Delta_e = x_{\text{stop}(e)} - x_{\text{start}(e)}
$$

Where $x_{\text{start}(e)}$, $x_{\text{stop}(e)}$ are the pixel values are connected by an edge $e$, $\text{start}(e)$ and $\text{stop}(e)$ are nodes indexes of edge $e$.

3.3.1.3. Calculation of encoding costs for bypass

For each fragment, estimates the encoding cost for each of the possible bypasses are calculated. The function receives the previously prepared fragments and details data on all the fragment bypasses in the codebook.
The result is the fragment-specific array containing estimated encoding costs for each bypass of the fragment (Fig. 5, Table 1).

Estimations of the encoding costs of a bypass through all edges (with its all delta-codes) of bypass edges it is possible to produce can be done in different ways. The cost of bypass for each fragment is needed to find the cost of the bypass:

\[ \Sigma_s = \sum_{e=1}^{E} \text{Cost}(\Delta_e) \cdot z^{es} \]  

(8)

where \( E \) is the length of bypass (the number of edges); \( e \) - edge index; \( S \) - the number of bypasses; \( s \) - bypass index; \( \Delta_e \) - delta-code of edge; \( z^{es} \) - presence of the same edge in bypass \( s \); Cost is the some estimation function of encoding costs, it is similar to the Cost in interchannel decorrelation function.

It should be noted that the estimate of bypasses encoding costs based on the table 1 is effective from the point of view of parallel computing. In this function there is a parallel processing of all possible bypasses downloaded from the codebook, for all fragments making up the processed image.

### 3.3.2. Search of the minimum among the possible bypasses

After the estimates of encoding costs of all paths is selected, and the save path of each fragment bypass with the smallest estimate for each fragment is picked and saved.

\[ s = \text{argmin}_s(\Sigma_s) \]  

(9)

![Fig. 5. Example of bypass of fragment 3*3 fragment.](image)

### Table 1. Calculation of encoding cost for bypass in fragment 3\(^2\) fragment

<table>
<thead>
<tr>
<th>e</th>
<th>( \Delta_e )</th>
<th>( \epsilon_1 )</th>
<th>( \epsilon_2 )</th>
<th>( \epsilon_3 )</th>
<th>( \epsilon_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \Delta_1 )</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>( \Delta_2 )</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>( \Delta_3 )</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>( \Delta_4 )</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>( \Delta_5 )</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>( \Delta_6 )</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>( \Delta_7 )</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>( \Delta_8 )</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>( \Delta_9 )</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>( \Delta_{10} )</td>
<td>0</td>
<td>1</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>( \Delta_{11} )</td>
<td>0</td>
<td>1</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>( \Delta_{12} )</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>0</td>
</tr>
</tbody>
</table>

### 3.4. Encoding of bypass

This function is designed to encode the previously found optimal bypass. That is, the obtained array of bypass nodes values with a minimum encoding costs must is to be handled by the processed with predictor and encoded by-with the encoder.

It should be noted that the encoding of bypass which previously was found as optimal bypass can be performed in various ways. In particular, there can be used certain known-generic common methods can be used: Huffman algorithm or arithmetic coding.

As for the considered suggested algorithm it is offered to perform the algorithm the bypass encoding of bypassis suggested which employs using a more sophisticated method [6]: 1) using a set of predictors and encoders for-to encode the bypass edge; 2) using dynamic programming for choice of choice predictors and encoders for edge in purpose to optimize (to minimize) of the total bypass encoding costs of bypass.
1.1.2.3.4.1. Encoding of bypass edge with different predictors and encoders

In the simplest most common case, the edge delta-code described above is used as the a predictor uses the edge delta-code described above, and Fibonacci codes are used as encoders uses the Fibonacci codes [3]. In this case, the application of dynamic programming to select the predictor and the encoder is not required. In more complex cases the number of choices of predictors and encoders variants may be more than one. For example, it is possible case with the predictors are possible not only on the basis of not only the finite difference of the first degree, but higher degrees, and Rice codes with different bases can be used as encoders with codes with use Rice codes with different bases. The use of a set of predictors and a set of encoders increases the resulting image compression ratio, but this raises the problem of choosing the best predictor and encoder for the current section of the bypass array.

1.1.2.3.4.2. Choice of predictor and encoder based on dynamic programming

In this embodiment, This variation of compression algorithm to encode with which each bypass edge is encoded uses the most optimal encoding parameters (predictor/encoder) based defined by dynamic programming [6]. In start of the function is started, it is loaded the table of encoding cost for every encoder for values of every predictors for all pixels on edges of the optimal bypass is downloaded and executed.

In the result The function creates a data file containing information with on the size of the encoded using encoders predictors values for edges and additional information - optimal switching of the predictors/encoders for edges of bypass encoding [6].

Due to the complexity of the dynamic-programming algorithm it was found possible to transfer into run on OpenCL, managed to transfer only a small part, responsible for the coding directly to OpenCL. This part contains branching, and is switching large sections of the algorithm takes place. These operations are an integral part of the algorithm, or changing the calculations flow in aim of parallel execution without the use of branches is not possible.

2.4. Results and Discussion

Screen form The interface of the developed software is shown in Fig. 6. The program displays the following information: used hardware processor device (processing unit) and software platform being used, the number of files to be processed; the current processed file; the execution duration time of the compression program particular functions; the execution duration of overall compression time; the compression ratio.

To use Hardware requirements for OpenCL acceleration requires the presence of either GPU or CPU with support of OpenCL 1.2 [8]. You must install the appropriate OpenCL support software which distributed with equipment is to be installed.

To use hardware processor device (processing unit) and software platform being used, the number of files to be processed; the current processed file; the execution duration time of the compression program particular functions; the execution duration of overall compression time; the compression ratio.

To use Hardware requirements for OpenCL acceleration requires the presence of either GPU or CPU with support of OpenCL 1.2 [8]. You must install the appropriate OpenCL support software which distributed with equipment is to be installed.

To use Hardware requirements for OpenCL acceleration requires the presence of either GPU or CPU with support of OpenCL 1.2 [8]. You must install the appropriate OpenCL support software which distributed with equipment is to be installed.

The basic program required about 250 MB of RAM. When algorithm was adapted the algorithm for accelerating on OpenCL was added using large-volume arrays were added of large volume and hence memory required demand increased to 900 MB.
Test batch-sample of images designed is intended to assess acceleration of all core functions of modified program in comparing with relative to basic one. To estimate the dependence of the program speed - runtime on the size - images size, the batch have the images of different sizes are sampled. To check used For test purposes 4 images from the standard set provided by the Institute of signal processing and images processing were used: 4.1.06.tif, 4.2.05.tif, 4.2.06.tif, 4.2.07.tif. The color depth of the images are is 24 bit. Image sizes are: 256*256, 512*512, 1024*1024, 2048*2048.

At testing was performed using image compression was performed with both - the basic program on the CPU AMD Phenom II X4 955 platform and a program using OpenCL. For testing OpenCL parallel processing was used different 4 different devices were used: 1) GPU AMD Radeon HD6850; 2) GPU Nvidia GTX 960; 3) CPU AMD Phenom II X4 955; 4) CPU AMD FX-4300. The time spent on the particular functions of the algorithm, and the total processing time for each image are given in Table 2 and Fig. 7, where F1 - integer-valued Haar transform, F2 - interchannel decorrelation of color layers, F3 - search of the optimal bypass, F4 - encoding bypass using dynamic programming.

When testing for each fragment had was fixed size: 6*6 pixels, and the number of bypasses: 22144 at testing. It should be noted, that when using the with GPU Nvidia GTX 960 configurations, according to Profiler, the load does not exceed 60% while despite numerous the high number of processing devices and high work frequency, according to Profiler the load does not exceed 6 Compression. The image of size 2048*2048 pixels size could not be compressed on the GPU AMD Radeon HD6850 failed to produce due to the lack of sufficient graphics memory. In the future, to avoid this situation such failures, the necessary modification of the program needs to be modified: to run the performed calculations flow should be divided into several groups threads and processed sequentially.

In the basic program were not implemented the functions of the integer-valued Haar transform and the interchannel decorrelation were not implemented, and therefore, testing of these functions was not carried out. Testing showed yielded approximately the same reduction in compression total overall compression time when using with both CPU and GPU application. The larger the size of the processed image, the greater the acceleration obtained as long as there is memory available for OpenCL.

GPU showed the best results performed better in the for searching of the optimal bypass task. CPU well with the function of handles dynamic programming well, due to because of presence of a large number of branches in the function, despite the small number of processor of processor cores. Calculations of interchannel decorrelation and integer-valued Haar transform is performed using OpenCL for a short times insignificant compared to total compression time.

Table 2. Results of processing of test images:

<table>
<thead>
<tr>
<th>Program / device</th>
<th>Image size, pixels</th>
<th>Execution time of compression particular functions, milliseconds</th>
<th>Acceleration, Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Radeon HD 6850</td>
<td>256*256</td>
<td>71 65 677 528 2581 2.2 9.0 2.8</td>
<td>F1 F2 F3 F4 Total</td>
</tr>
<tr>
<td></td>
<td>512*512</td>
<td>43 63 913 1129 4493 6.1 13.4 5.2</td>
<td>F3 F4 Total</td>
</tr>
<tr>
<td></td>
<td>1024*1024</td>
<td>66 187 2017 4492 15217 12.9 13.5 6.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2048*2048</td>
<td>151 438 — — — — — —</td>
<td></td>
</tr>
<tr>
<td>Nvidia GeForce GTX 960</td>
<td>256*256</td>
<td>23 20 160 360 2057 9.4 13.2 3.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>512*512</td>
<td>11 36 411 1153 4402 13.6 13.1 3.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1024*1024</td>
<td>33 148 1474 4226 13416 17.7 14.4 7.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2048*2048</td>
<td>122 675 533 14530 50031 31.8 15.9 8.6</td>
<td></td>
</tr>
<tr>
<td>AMD CPU FX-4300</td>
<td>256*256</td>
<td>54 30 150 499 2581 4.3 9.5 2.8</td>
<td>F1 F2 F3 F4 Total</td>
</tr>
<tr>
<td></td>
<td>512*512</td>
<td>18 48 913 1146 4207 6.1 13.2 5.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1024*1024</td>
<td>45 206 1324 4093 15514 7.9 14.8 6.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2048*2048</td>
<td>172 920 13164 13629 58567 12.9 17.0 7.4</td>
<td></td>
</tr>
<tr>
<td>AMD Phenom II X4 955</td>
<td>256*256</td>
<td>31 32 455 664 2407 3.3 7.2 3.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>512*512</td>
<td>21 56 1378 1222 5981 4.0 12.4 4.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1024*1024</td>
<td>67 239 4731 4477 16777 7.7 13.6 5.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2048*2048</td>
<td>227 992 18432 12804 61356 9.2 18.1 7.0</td>
<td></td>
</tr>
<tr>
<td>Basic program / AMD Phenom II X4 955</td>
<td>256*256</td>
<td>— — 1504 4751 7275 1.0 1.0 1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>512*512</td>
<td>— — 5570 15155 24183 1.0 1.0 1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1024*1024</td>
<td>— — 26107 60998 97297 1.0 1.0 1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2048*2048</td>
<td>— — 170161 231398 431555 1.0 1.0 1.0</td>
<td></td>
</tr>
</tbody>
</table>
3.5. Conclusion

In the course of this work the basic program for lossless image compression was modified, with the aim of increasing its runtime speed. The parallel processing based on OpenCL was used for program acceleration. This solution significantly affected the processing speed, enabling making it possible to reduce computational time. This modification will allow more efficient use of the program in the future, will facilitate further research aimed at improving the compression ratio.

The changing of optimal bypass search function allowed for obtaining the acceleration up to 32-fold on the large images. This acceleration has been achieved because of executing OpenCL functions on OpenCL are almost linear, and branching, even where they are when it is the case, is limited to only a few simple operations. Furthermore, for future program modification the acceleration of this function is important for future program modification because it is possible to use fragments of larger size which was previously impossible because earlier due to too much great execution time. Among other things, Moreover, fragments with the size of $2^n \times 2^n$ will effectively allow applying the integer-valued transformation to the fragments, and will allow to compressing every each fragment separately.

Somewhat worse is the situation with dynamic programming. However, the prospects for the future are not as bright during encoding parallel execution of some operations, shutdown of operations which need only be used solely for debugging purposes and the use of the packet data read operations. The part that runs on OpenCL gives the increase in performance is only about 30% compared with ordinary parallel computing. On the other hand, even this result is relatively good enough, given the fast provided that OpenCL function has relatively large enough branching. It should be noted that the bypass encoding can be performed in various ways, for example with Huffman algorithm or arithmetic coding.

References