GPU Implementation of a Medical Imaging Data Compression Algorithm

Dino Šantl[1], Marko Đurasević[2], Josip Knezović[3]

[1] FER, dino.santl@fer.hr

[2] FER, marko.durasevic@fer.hr

[3] FER, josip.knezovic@fer.hr

***Abstract:** In this paper an algorithm for compressing medical images on GPUs is described and its energy consumption and computational performance analyzed. The objectives of this implementation are to reduce the demands of storing medical pictures, improve the execution time and reduce the consumption of electrical energy. The compression algorithm is based on a computationally complex prediction model of pixels and a contextual coding of the given prediction errors. The above implementation is compared with a serial implementation and with an implementation in a stream programming model in terms of execution time and electrical energy consumption. Experimental results show improvements in execution speed and savings in consumption of electrical energy using the proposed implementation.*

Keywords: *Medical Visual Data Compression, General Purpose GPU Programming, Lossless Coding Parallelization, Data-Intensive Computing.*

1. INTRODUCTION

Current information systems used in different human sectors, such as in medicine, are very complex in nature. The need for computational power and storing large amounts of data grows each day. Because processors are reaching their limits in terms of speed, multiprocessor systems are used more commonly. Apart from the need for high computational power, there is also a need for managing large amounts of data. Instead of storing data in raw format, different data compression algorithms are used in order to use storage space more rationally. Compression algorithms are divided in two different groups: losy and lossless. Losy algorithms have a much better compression rate, which comes at the cost that the original image cannot be restored without data loss from the compressed image. Lossless compression algorithms, on the other hand, have a lower compression rate, but allow for the image to be restored with no data loss. With the growth of computational power, energy consumption becomes more of a problem [1]. Therefore, beside raw performance, energy management for large systems is becoming more and more important. Hospital information systems are a typical example of such complex systems. One of their main characteristics is the need to store huge amounts of data.

In this paper we propose an implementation of a lossless compression algorithm *CBPC* for hospital information systems [2,3]. This algorithm consists of a very computational intensive pixel predictive part. As a solution to this problem we present a parallel implementation of this computational intensive part using commodity graphics processing units [4, 5]. The concept of using graphical processors as an accelerator for general purpose applications is a new concept that is becoming more popular. In the proposed implementation we chose the *CUDA* programming model and environment [6].

This paper is organized as follows. In section two we give a short overview of the *CBPC* algorithm and it's main characteristics. In section three we present the implementation details of the proposed solution. The experimental results for energy consumption and algorithm execution time between different implementations are given in section four. Finally, section five gives a short conclusion about this topic.

2. CBPC COMPRESSION ALGORITHM

Images and texts are very different types of information. Usual texts are compressed with classical methods like dictionary based or entropy coding, while images require a separate statistical redundancy removal step, most often in the form of a pixel predicting method. The basic idea for image compression is based on the prediction of image elements.

The prediction is based on a mathematical model. We know an image element and the prediction for this element. Prediction depends on the model and image elements which are relatively near to the observed image element. The prediction error is defined as the difference between the original image element value and the value of prediction. The error distribution acquired through such a method is much more convenient for compression then the original image elements. The mentioned compression algorithm is lossless. This is an important property for medical usage, because losing any information can be fatal for patients.

Steps of the CBPC algorithm are the following (shown on Figure 1)[7]:

1. Input - importing image elements from one image or from a medicinal study – image elements (pixels) are processed in raster order from the top left, down to the lower right corner.

2. PM - Predicting modeling - prediction of the current image element and prediction error computing.

3. CM - Contextual modeling - prediction error classification and generating a different distribution for every class of error.

4. EN - Entropy modeling - using classical entropy coding for variable bit symbol generation of prediction error.

5. Output - save compressed image – saving the generated bit stream in compressed format.

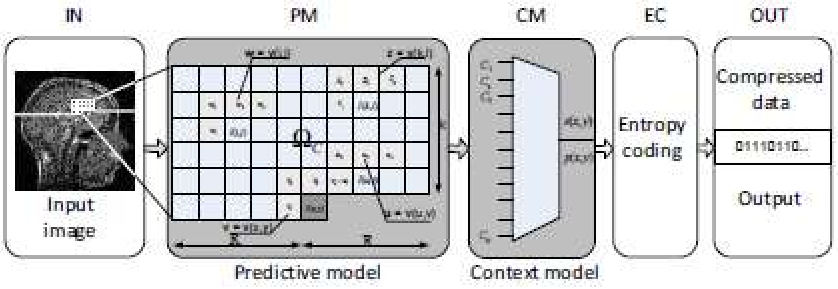

Figure 1: CBPC algorithm steps

Prediction is based on image elements neighbors. Neighbors of an image element are caled regions. In this algorithm a region is a set of image elements defined with relative coordinates from point (x,y) as shown in figure 2.

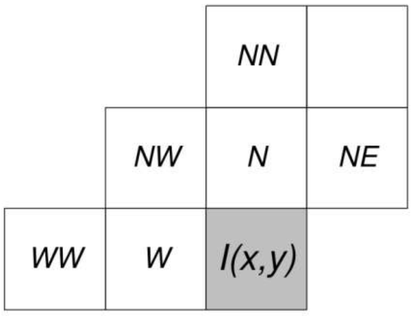

Figure 2: Pixel region

Prediction of current image element is computed using the equation (1)

$$\hat{I}(i,j) = \frac{\sum_{k=1}^{N}\frac{1}{G_k} * \hat{f}_k(i,j)}{\sum_{k=1}^{N}\frac{1}{G_k}} \tag{1}$$

Prediction error for current element with blending is computed using the equation (2)

$$e(x,y) = I(x,y) - \hat{I}(x,y) \tag{2}$$

This algorithm is based on blending prediction function (1). All functions have some influence in the equation and their part depends on their penalty.

$$G_k = \frac{1}{M}\sum_{I(i,j)\in\Omega_B}(\hat{I}_k(x,y) - I(x,y))^2 \tag{3}$$

Equation (3) represents a nice mathemathical model for implicit region type detection, where G_k represents the penalty and $\hat{I}$ the prediction value. We use k prediction functions. Each one of this functions f_k is good for predicting some types of the image region. For example some functions f_k are good on noisy regions of an image, and other functions are good for the edges. The main problem is to detect the type of region on the image. The role of the parameter G_k is to force down the functions which are not good on predicting the current image region. Cell is set of image elements which are the nearest to the current image element region. The distance between regions is computed with a certain metric. We use the Euclidean metric. Ω_c is the causal context used by the predictor (as can be seen on figure 1 in the predictive model). Image elements in this set are candidates for the cell. M is the number of image elements in the cell. We call this set the blending context $\boldsymbol{\Omega}_B$.

Parameters in the algorithm are:

1. R - size casual context (diameter)
2. M - number of image elements in the blending context
3. f_k - set of prediction functions

3. IMPLEMENTATION

Implementation was built based on a streaming programming model. Because image elements are independent, fine-grained data splitting was used. The implementation goal was to achieve a good algorithm design, so that the tradeoffs between good design and an optimized code could have been made. It was previously shown that the algorithm can be pipelined in five steps. We use the same steps in our implementation design. The implementation copies the whole image on from the main memory into the *GPU* global memory. The first step is to make regions of image elements. Then for every image element their region is computed. This information is then further used for creating cells. At this point the algorithm has all the information for prediction computing. When the prediction was computed the parallel part of algorithm is finished and data is ready for classical coding using an entropy coder of choice. It is important to note that only the prediction part of the algorithm is parallelized because it alone executes around 85% of the total algorithm execution time.

The implementation was made using the CUDA programming model [6]. Every thread handles one image element. After each algorithm step synchronization between all threads is performed because all the information computed in one step has to be available in the next step. Two types of optimizations were made. The first was memory optimizations. Access to global memory on the GPU is very expensive and slow. CUDA offers some possibility for memory optimization. When neighboring threads access neighboring memory locations it gives a possibility for the second memory access to be serialized. The second type of optimization is finding the optimal number of threads and blocks. Empirically it was shown that the best strategy is when only one block was used and the number of threads was equal to number of image elements in one row. Unfortunately, these results are dependent on the specific graphics card which is used.

The main parts in the CUDA implementation are the following:

1. Finding the blending context for every image element
2. Calculating predictions for every image element
3. Computing the error and context

These three steps are running sequentially, but each step is run in parallel for every image element. In the first step, the algorithm computes the blending context or every image element. In step two, the algorithm computes the prediction for the current image element. In the end of the algorithm the error and context are computed for every image element.

During the execution of the algorithm, the image is stored into the global memory the on graphics device. The image is then separated into blocks of the same size, which are then further used during the execution of the algorithm.

4. EXPERIMENTAL RESULTS

The proposed solution has been compared against a serial implementation of the same algorithm (implemented in *C++*), and with an implementation of the algorithm using the streaming programming model and the programming language *StreamIt* [8]. The measurements were obtained on five different sets of medical images. All the images were in stored in the *PGM* format and had dimensions of 512x512 pixels. Basic image characteristics are shown in Table 1.

4.1. Compression execution time

One of the most important evaluation criteria is the execution time needed for compressing whole sets of medical images. Such sets can contain large amounts of data, equaling in hundreds of megabytes as shown in Table 1. Therefore, it is crucial that the compression of such large amounts of data can be completed as fast as possible.

Table 1: Medical sets used for measurements

Medical set name	Number of images	Total size of set [MB]
mr-breast-dm	2534	486,43
mr-breast	2241	559,53
mr-pelvis	383	95,76
ct-mix	3003	750,79
ot	1373	343,27

Figure 3 shows the execution times of the three different implementations for the five medical image sets. As expected, the implementation on the *GPU* delivers the best results for all medical sets. Overall, it was 2.47 times faster when compared to the serial implementation while compressing the medical test sets. On the other hand, the *StreamGate* implementation executes slower than expected, but nevertheless provides a certain speed up when compared to the serial implementation. The problem with the *StreamGate* implementation is that it repeatedly creates and destroys threads for each image it compresses, which proved to have a large influence on the performance of the implementation.

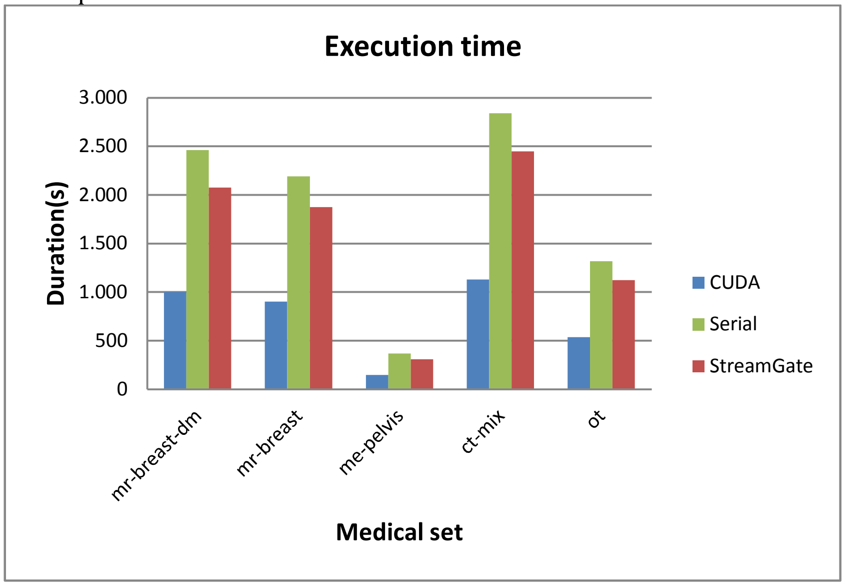

Figure 3: Execution time of all three implementations on the test sets

4.1. Energy consumption

Because parallel implementations of algorithms use more computational power and more resources, the question arises: "How does this affect the energy consumption during execution of such algorithms?" Figure 4 shows the energy consumption for compressing all medical sets with all implementations. Although the *GPU* unit generally does consume more power than the *CPU* unit, because of the high speed up, the energy consumption is overall lower when using the *GPU* implementation. For the smaller medical sets the consumption is mostly the same, because the speed up is not high enough to compensate for the higher consumption of the *GPU.* When compared to the serial implementation, the *CUDA* implementation achieves an overall reduction in energy consumption of 32%.

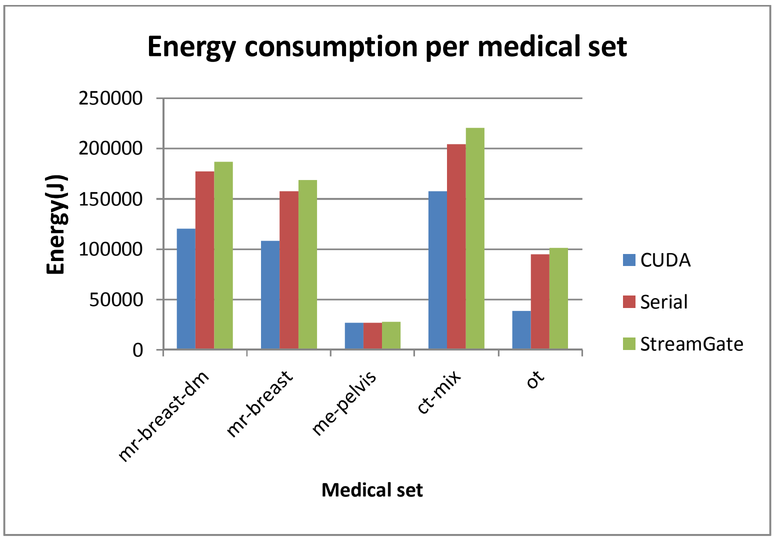

Figure 4: Energy consumption of all implementations for the test sets

5. CONCLUSION

In this paper we present a lossless compression algorithm implementation on the GPU which can significantly improve system performances. The compression factor is almost the same as in serial algorithm (the serial algorithm has some additional optimizations which were not included in the GPU implementation) but the GPU implementation needs less electrical energy for compressing the same size of image data. This property is very important for large systems where a huge amount of energy is consumed. Compression is also very important for systems where large amounts of data are stored. Finding methods and new algorithms which can make achieve a better compression factor and have other additional properties (like data parallelism) is important for real systems. The proposed algorithm implementation can further be improved and this is not the final step of our work on the *GPU* implementation, but rather a proof of concept to show that such an implementation has its benefits and advantages for usage in real systems.

LITERATURE

[1] Soomro T. R. and Sarwar M.: *Green Computing: From Current to Future Trends*, In *World Academy of Science, Engineering and Technology*, 2012.

[2] Knezović J., Kovač M., and Mlinarić H.: *Classification and blending prediction for lossless image compression,* In Electronic Proceedings 13th IEEE Mediterranean Electrotechnical Conference, (Malaga, Spain), 2006.

[3] Knezović J., Kovač M., Klapan I., Mlinarić H., Vranješ V., Lukinović J., and Rakić M., *Application of novel lossless compression of medical images using prediction and contextual error modeling,* Collegium antropologicum, vol. 31, no. 4, pp. 1143–1150, 2007

[4] Owens, D. John, Luebke, David, Govindaraju, Naga, Harris, Mark, Kruger, Jens, Lefohn, E. Aaron, Purcell, and J. Timothy, A Survey of General-Purpose Computation on Graphics Hardware, Computer Graphics Forum, pp. 80–113, March.

[5] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," IEEE Micro, vol. 31, pp. 7–17, Sept. 2011.

[6] Cuda Toolkit Documentation; http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html; July 2013.

[7] Knezović J., Mlinarić H., Žagar M.: Lossless I_mage Compression Exploiting Streaming Model for Efficient Execution on Multicores, *Automatika,* vol. 53-2012, pp. 272-283, 2012

[8] StreamIt Cookbook; http://groups.csail.mit.edu/cag/streamit/papers/streamit-cookbook.pdf; July 2013.