Vector Quantization

A Many-Core Approach

Rita Silva, Telmo Marques, Jorge Désirat, Patrício Domingues
Informatics Engineering Department
School of Technology and Management, Polytechnic Institute of Leiria
Morro do Lena, Alto do Vieiro, 2411-901 Leiria, Portugal
{rita.lauz, telmofmm, jdesirat}@gmail.com, patricio@ipleiria.pt

Abstract— **Many-Core computing is an actual growing concept that allows the true parallelization of computational tasks. In the particular case of this paper, the vector quantization algorithm was adapted to the many-core concept with the objective of compressing images encoded in the PGM format. For that, a given sequential implementation of the algorithm was optimized and reengineered to support NVIDIA's CUDA framework and OpenCL open framework, resulting in two similar implementations of a much faster version of the vector quantization algorithm.**

Keywords: parallel computing, many-core computing, vector quantization, CUDA, OpenCL.

I. Introduction

Multimedia content encoding is crucial, as it allows reducing content sizes, and therefore, reducing costs in storage and transmission of content. However, content encoding can be an onerous task with significant computational cost, both for coding and decoding. In this context, it is usually more tolerable to use a better encoding algorithm that is computationally heavy, but whose decoding process is relatively lightweight, rather than the contrary, because it's more usual to find content that is to be encoded only once (when created) but is decoded numerous times.

This paper presents an adaptation of the Vector Quantization algorithm with a many-core paradigm computing approach, namely with CUDA [1] and OpenCL [2].

The group was initially given a sequential version of a VQ implementation (in C language) that could be used to encode pictures. A decoder and a LBG-VQ based application to create encoding dictionaries were also given. After some code re-engineering of this sequential version the work group moved on to the many-core approach, firstly relying on CUDA paradigm (provided by NVIDIA) and later adapting to OpenCL, in order to understand which one was faster and more efficient.

Throughout this paper it will be explained how this was accomplished and what were the achieved results.

II. Vector Quantization

Vector quantization is a quantization technique from signal processing, which consists in a lossy data compression method, based in the principle of block coding[3][4]. It is a system used for mapping a sequence of vectors into a digital sequence more suitable for communication (by compressing data). In other words, this system is capable of encoding values from a multidimensional vector space into a finite set of values from a discrete subspace [5].

In 1980 Linde, Buzo and Gray (LBG) proposed a vector quantizer design algorithm based in a training sequence, presenting a way to bypass the previous challenging problem created by the multidimensional-integration needed for vector quantization.

LBG is an iterative algorithm that processes a large set of vectors at each iteration (a training set). It requires an initial codebook, obtained by using a splitting method in which an initial codevector is set as the average of the entire training sequence. This codevector is then split in two, and the algorithm runs again with the two new codevectors compounding the initial codebook. The two codevectors are then split into four, and the process is repeated until the desired number of codevectors is obtained.

In this particular case, the vector quantization algorithm is used to replace a set of pixels (a block) with a unique value from a provided dictionary that is created resorting on the LBG VQ algorithm. This dictionary is also used later to decompress the image, resulting in a less-quality replica still holding the same amount of pixels as the original image, also in the PGM format.

III. Many-Core Implementation

As previously stated, it's objective of the described project to parallelize a given sequential implementation of the vector quantization algorithm, used to compress images encoded in the PGM format. Below we provide a quick overview of the given sequential implementation, and explain in detail the approach and options taken to the implementation of the parallelized version of the algorithm, done using both CUDA and OpenCL frameworks resulting in two very similar implementations.

A. Sequential Implementation Overview

A sequential implementation of the vector quantization algorithm was provided as the base of the developed project, and was composed by two main executables, written in C: an encoder and decoder, that could be executed in a Linux shell with the objective of compressing and decompressing images in the PGM format. The encoder, when executed, received as parameters the name of the PGM encoded image, to be compressed, and the name of the dictionary file, containing entries with which blocks of pixels were to be replaced. After execution an output file is generated, contained entries from the provided dictionary, and presenting a smaller size than the original image. The decoder program is then used together with the output file to recreate the original image, with the downside of losing quality in the process.

To give a basic technical understanding of the encoder underlying algorithm, the one adapted to the many-core concept, a quick description of its sequential actions is described below:

1. Both the image and dictionary files are loaded into memory, represented using two-dimensional arrays,
2. A luminance average of the image is then calculated by sequentially adding up all pixel values (raging from 0 to 255), and dividing the result by the image size, in pixels,
3. The previously calculated luminance average is subtracted from all the pixel values of the image, done by iterating all the values to perform the subtraction,
4. The image is iterated, block by block, comparing each block to all dictionary entries by calculating the quadratic error between them, and then choosing the dictionary entry that presented the smallest error. The iterated blocks were always a set of pixels of equal width and height, being 2x2 and 4x4 the supported sizes,
5. Finally, the chosen dictionary entries that best depict the original image are sequentially written to an output file.

This given sequential implementation was, additionally, optimized with the objective of getting the best results possible from this implementation, and later compared with the CUDA and OpenCL versions of this project. Such optimizations primary constituted some reengineering of the code, eliminating global variables and adapting existing functions to the changes made on variable's scope, and also the translation of two-dimensional array's, used to represent the image pixels and dictionary entries, to a one-dimensional array. This last change was done to simplify CUDA and OpenCL implementations, as described in more detail below, but also to optimize the allocation and access to the elements of the array by eliminating the need of having pointers-to-pointers, resulting in faster memory allocation and simpler access. Finally, this given sequential version of the algorithm was also compiled using the -O3 GCC [6] compiler flag, and tested for accuracy, guaranteeing the proper functioning of the program.

Given this, below is described the approach taken to design a parallel algorithm with the same results.

B. *Many-Core Approach*

The objective of the designed parallel implementation was primarily to perform the previously described fourth step concurrently to several blocks of pixels, meaning that each block of 2x2 or 4x4 pixels were to be processed at the same time, each inside an independent thread. The sum and subtraction operations described in the second and third steps, respectively, were also to be parallelized.

One of the key implementations of the sequential implementation that was changed in this parallel approach was to represent the PGM image pixels and dictionary entries using a one-dimensional array, instead of the two-dimensional array used on the sequential version of the algorithm. The flattening of the arrays to one dimension was done by using row-major ordering, with the objective of simplifying both CUDA and OpenCL implementations, by eliminating the need to allocate two-dimensional arrays in the device memory, and meaning that thread's unique identifiers would become directly mapped to the array elements through the use of a function later described. This also optimizes array element access as arrays are contiguously represented in memory.

Three tasks were then to be parallelized: the sum of all pixel values, the subtraction of the calculated luminance average from all the pixel values and, finally, the replacement of blocks of pixels with the dictionary elements that best represented said blocks. For this, three device kernels were designed with the purpose to run on a GPU device, following the below described actions:

1. Both the image and dictionary files are loaded into memory and represented using one-dimensional arrays;
2. A luminance average of the image is then calculated by performing a reduction on the GPU, and dividing the result by the image size, in pixels;
3. The previously calculated luminance average is subtracted from all the pixel values of the image, done in a parallel fashion using the GPU;
4. The pixels blocks of the image are iterated in parallel by the GPU, comparing each block to all dictionary entries by calculating the quadratic error between them, and then choosing the dictionary entry that presented the smallest error.
5. Finally, the chosen dictionary entries that best depict the original image are sequentially written to an output file.

This approach optimizes three tasks previously done in a sequential manner, most importantly the task where blocks of pixels are replaced by dictionary entries. This approach, while still keeping the dictionary iteration and quadratic error calculation sequential inside each thread, satisfies the initial objective for processing several blocks of pixels at the same time.

1) *CUDA Implementation*

The CUDA implementation of the many-core approach to the given algorithm was written in C and followed the aforementioned objectives by implementing three CUDA kernels.

The first kernel performs a sum of all the values of the image's pixels. This kernel performs a reduction on the one-dimensional array that represents the pixels of the image in memory and, as a result, a smaller set of values is returned by this kernel, one value by each CUDA block, which is finally added up sequentially on the CPU. This value is later used to calculate the luminance average of the image.

The second kernel subtracts the luminance average of the image from each pixel value. This is done by delegating the subtraction of the average from a single pixel value to each CUDA thread. If there are more pixels than CUDA threads available, the threads jumps ahead to perform another subtraction, until all pixels have been processed.

Finally, the third developed kernel has the objective of obtaining, from a given dictionary, a single value that best represents a block of pixels from the image, thus performing the vector quantization algorithm on the image. The kernel begins by keeping a copy of the one-dimensional arrays that represent both the image and the dictionary on the device's global memory, and also providing memory space, also on the global memory, on which CUDA kernels can write the obtained dictionary values. Given this, each CUDA thread receives a unique identifier that directly maps that thread to a block of pixels of the image, being this unique thread identifier equal to the array index that represents the first pixel of that block. This was a particularly difficult task to implement, as these unique thread identifiers had to be carefully calculated in a way that the processed pixel blocks wouldn't overlap.

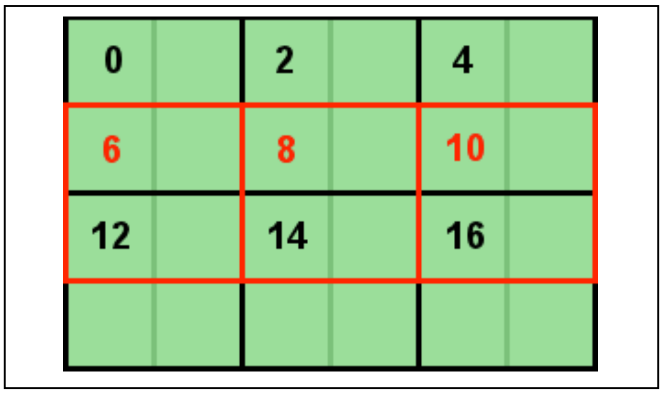


Figure 1. Overlapping, or intermediate, blocks problem. The numbers represent the block's identifier pixel index, and In red are represented the non useful, "overlapping", blocks.

The above image best describes the problem in hands. The 2x2 pixel blocks represented in black are the ones we intend to process, while in red we have the blocks that are just intermediate, whose result is not useful. Processing these blocks is a waste of computational resources. To solve this problem a function that is capable of calculating the unique CUDA thread identifiers, in a way that only useful block are mapped by those threads, was deduced, being as follows:

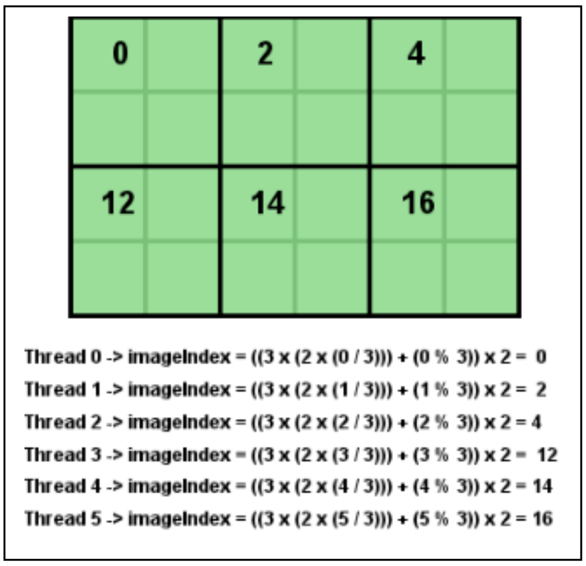
$$\text{nBlocks} = \text{imageSize\_x} \div \text{dictionarySize\_x}$$

$$\text{threadIndex} = \text{threadIdx.x} + (\text{blockIdx.x} * \text{blockDim.x})$$

$$\text{threadImageIndex} = ((\text{nBlocks} \times (\text{dictionarySize\_x} \times (\text{threadIndex} \,/\, \text{nBlocks}))) + (\text{threadIndex} \,\%\, \text{nBlocks})) \times \text{dictionarySize\_x}$$

Where ÷ is integer division and % is modulo operation.

By using this function the available CUDA threads are directly mapped only to useful blocks, while also allowing them to jump ahead in case there are more blocks than threads. The following image shows an example of the usage of this function on a 6x4 pixel image using 2x2 pixel blocks.



Thread 0 -> imageIndex = ((3 x (2 x (0 / 3))) + (0 % 3)) x 2 = 0
Thread 1 -> imageIndex = ((3 x (2 x (1 / 3))) + (1 % 3)) x 2 = 2
Thread 2 -> imageIndex = ((3 x (2 x (2 / 3))) + (2 % 3)) x 2 = 4
Thread 3 -> imageIndex = ((3 x (2 x (3 / 3))) + (3 % 3)) x 2 = 12
Thread 4 -> imageIndex = ((3 x (2 x (4 / 3))) + (4 % 3)) x 2 = 14
Thread 5 -> imageIndex = ((3 x (2 x (5 / 3))) + (5 % 3)) x 2 = 16

Figure 2. Demonstration of deduced function.

After all pixel blocks are processed, and respective best representative elements from the dictionary obtain, these last ones are copied back from the GPU's memory and sequentially written to a file by the CPU, as it is required to write a file to a computer's storage device, such as a hard drive.

2) *OpenCL Implementation*

The OpenCL implementation, also developed in C, being done after the CUDA implementation, followed very closely the one of CUDA, the two being very much algorithmically identical.

Regarding the OpenCL implementation, syntax and function calls were adapted from the developed CUDA code to support the OpenCL framework and, being more dynamic in a way it was designed from the ground up to run on devices other than GPU's, it required a little more initial effort to make the appropriate function calls to prepare the machine to run OpenCL code. As the OpenCL kernels follow exactly the same implementation as CUDA's (almost all kernel code was directly compatible, changing only framework specific terminology), reading the previous chapter, Cuda Implementation, will provide all the insight needed to understand the OpenCL implementation of this project.

IV. Evaluation

In this section we present the method of execution and results from the performed tests.

A. Execution

The method used to obtain the results we present below was to run the sequential, CUDA and OpenCL implementations ten times on the same machine, and picking out the best results.

The test machine specifications were as follows:

- Intel(R) Core(TM) 2 Quad CPU Q9400 @ 2.66GHz
- NVIDIA Corporation GeForce GTX 480 GPU
- 4 GB DDR3 of RAM
- Ubuntu Server v11.04 64-bit operating system
- CUDA framework v2.0

As for the number of CUDA/OpenCL blocks/work-groups and threads/work-items used when testing, 256 CUDA threads times 32 CUDA blocks were used to perform the sum (reduction) and subtraction, while 1024 CUDA threads times 8 CUDA blocks were used to perform the replacement of blocks of pixels. Regarding OpenCL, a total of 128 work-items distributed along 4 work-groups (resulting in 32 work-items per work-group) were used to perform all of the kernels. These values were hardcoded into each of the specified implementations and represent the greatest values from which better results were obtained, given the machine specifications previously described. Additionally, the sequential version results provided below were also taken from the most optimized version of the sequential implementation we could get (using optimizations described in the Sequential Implementation Overview section of this paper), compiled using GCC together with the -O3 flag.

B. Test Results

After implementing both CUDA and OpenCL versions of the project, some tests were made in order to assess which implementation performed better to compress PGM encoded images. The obtained values are presented below in a tabular format.

TABLE I. Test Results (ms)

Image (image size)	**GPU Time (ms)**		**Total Time (ms)**		
	OpenCL	***CUDA***	***OpenCL***	***CUDA***	***Seq.***
ILikeEI.pgm (2560x1920)	629,52	202,53	990	700	17830
Informatica_e_no_IPL---cartaz.pgm (4069x2896)	1519,69	489,71	2380	1330	42970

a. Seq. sequential

TABLE II. Speedup

Image (image size)	**Speedup**	
	Sequential vs OpenCL	***OpenCL vs CUDA***
ILikeEI.pgm (2560x1920)	18.01×	1.41×
Informatica_e_no_IPL---cartaz.pgm (4069x2896)	18.05×	1.79×

These results show that, under the testing conditions provided, the OpenCL implementation was up to 18.05 times faster than the optimized sequential version, and the CUDA implementation up to 1.79 times faster than that of OpenCL's. The results also confirm that our OpenCL implementation was faster than sequential, but CUDA was even faster than OpenCL.

V. Conclusion and Future Work

Regarding future work, there are two main objectives. The first objective targets the way CUDA and OpenCL blocks/work-groups and threads/work-items are calculated to run each of the implemented kernels. As described, this task is currently executed given hardcoded static values, but optimally these values are to be calculated using an algorithm capable of choosing the best pair of values to any given machine running the code, by analyzing the machine's specification. The second, and final, objective consist of building a package containing all three implementations: sequential, CUDA and OpenCL; thus supporting a greater hardware base with a single package, while allowing the user to choose the preferred implementation, or, as an alternative, automatically analyzing the machine's specifications and executing the best implementation, based on the test results presented above.

As for CUDA and OpenCL, they both revealed themselves as capable frameworks to achieve the project's objectives, with no troubles compiling code either for ATI or NVIDIA GPU's, unlike those referred on "A Performance Comparison of CUDA and OpenCL" [7], as all the code was designed to be directly compatible with both frameworks.

Acknowledgments

We'd like to thank our teacher, Patrício Domingues, for the knowledge and support given in application debugging.

References

[1] Parallel Programming and Computing Platform | CUDA | NVIDIA. [online] Accessed Januray 2012: http://www.nvidia.com/object/cuda_home_new.html

[2] OpenCL - The open standard for parallel programming of heterogeneous systems. [online] Accessed January 2012: http://www.khronos.org/opencl/

[3] Vector Quantization. [online] Accessed January 2012: http://www.data-compression.com/vq.html

[4] Theory of Data Compression. [online] Accessed January 2012: http://www.data-compression.com/theory.shtml#theory

[5] Gray, Robert M. (April 1984). Vector Quantization. IEEE ASSP Magazine.

[6] GCC, the GNU compiler collection – GNU Project – Free Software Foundtation (FSF). [online] Accessed January 2011: http://gcc.gnu.org

[7] K. Karimi, N. G. Dickson, F. Hamze (May 2010). A Performance Comparison of CUDA and OpenCL. [online] Accessed January 2012: http://arxiv.org/pdf/1005.2581v3.pdf