in Proceedings of PDP 2015

Heterogeneous Acceleration of Volumetric JPEG 2000

Jan G. Cornelis[1,2], Jan Lemeire[1,2], Tim Bruylants[1,2], Peter Schelkens[1,2]
[1]Vrije Universiteit Brussel (VUB), Electronics and Informatics (ETRO) Dept.,
Pleinlaan 3, B-1050 Brussels, Belgium
jgcornel@vub.ac.be
[2]iMinds, Multimedia Technologies Dept.
Gaston Crommenlaan 8 (box 102), B-9050 Ghent, Belgium

***Abstract*—We present the implementation of a volumetric JPEG 2000 codec as a real-world use case of software acceleration with GPUs and multi-core CPUs. We present a generic methodology to accelerate existing code written in C with OpenCL. Furthermore, we account for the volumetric nature of the processed data and formulate associated optimization guidelines. The resulting software can exploit different accelerator types - GPUs and multi-core CPUs - and delivers a decent speedup on a variety of hardware platforms for a relatively small effort.**

***Index Terms*—hybrid, acceleration, opencl, volumetric JPEG 2000**

I. Introduction

In the last decade parallel systems and parallel programming have become increasingly important. The data sets that need to be processed have grown dramatically and the algorithms used are becoming more and more complex. Nevertheless, due to the end of frequency scaling it is no longer possible for traditional sequential programs to obtain a free performance gain with every new processor generation. Processors have now become multi-core. To exploit their computing power it is necessary to write parallel code. Furthermore, it is also possible to use the graphical card of a computer for general purpose computing. These many-core GPUs need even more parallelism to be exploited efficiently.

JPEG 2000 Part 10 or JP3D is an extension to the JPEG 2000 image compression standard to compress volumetric images. The size of such images and the complexity of JPEG 2000 compression and decompression make it a perfect example of a program in need of lots of computational power.

OpenCL is a standard developed in 2008 for parallel programming a variety of hardware accelerators. Contrary to NVIDIA's CUDA it is an open standard that is not limited to NVIDIA GPUs but that can be used to write code that runs on any hardware accelerator that provides the necessary support. For example, it is possible to run OpenCL code on GPUs, multi-core CPUs, FPGAs and other accelerators like Intel's MIC. These properties make it ideal to develop software that can exploit all available computing resources of the platform on which it is run.

This paper discusses the acceleration of an existing JP3D codec with OpenCL as a real-world case. Although this specific topic is interesting in itself, the main purpose of the paper is to illustrate how an existing sequential software can be accelerated relatively easily. We will explain the basic steps that need to be taken to modify the existing code, discuss the development of the OpenCL code and emphasize the main factors that need to be taken into account to obtain a satisfactory performance. Our implementation serves as a practical example. We will discuss the performance obtained on a variety of hardware. The main idea is to show the speedups that are possible with a relatively small effort. Nevertheless, the results are interesting in themselves because, as far as we know, they concern the only parallel implementation of a JP3D codec.

Most of this paper focuses on compression. Nevertheless, we also accelerated decompression using the same methodology and we will show results for both compression and decompression.

The remainder of this paper is organized as follows: in Section II we discuss JPEG 2000 Part 10 and OpenCL. Related work is presented in Section III. Section IV presents a methodology to accelerate an existing software with OpenCL. In Section V we discuss our implementation and the applied optimization techniques. In Section VI we present the accelerators and the images used for testing our implementation and discuss the obtained results. Finally, Section VII concludes the paper.

II. Background

A. JPEG 2000 Part 10

JPEG 2000 is a standard for compression of digital images. It is more recent than JPEG and uses a number of techniques that result in a superior coding performance compared to JPEG. For a thorough discussion of the principles of JPEG 2000 we refer to [17]. A comprehensive overview of the baseline JPEG 2000 standard and its extensions can be found in [15].

JPEG 2000 is divided in 14 parts. Part 1 describes the core coding system while the other parts describe extensions of the baseline standard. Part 10 - JP3D - [6] is an extension for encoding and decoding volumetric images. It provides the same functionality as the baseline JPEG 2000 standard and exploits all dimensions of the image to provide a superior compression

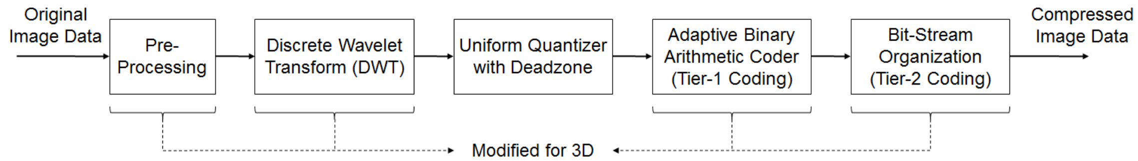

Fig. 1. Block diagram of a JP3D encoder

performance. Of course, a codec that implements volumetric JPEG 2000 can also code and decode two-dimensional images.

We explain briefly how a JP3D codec works with reference to figure 1. First, in order to decrease the memory requirements, the image may be divided in tiles, which for three-dimensional images correspond to cuboid subvolumes. Furthermore, for color images, different components are coded independently from each other. After preprocessing the image, which includes a DC level shift and possibly a multi-component transform, a discrete wavelet transform is applied. This results in a decomposition of the image in subbands. These subbands are divided in three-dimensional code-blocks that are processed by the EBCOT Coder in Tier-1. Before Tier-1 and for lossy compression, quantization is applied to the wavelet coefficients. During Tier-1, code bits and side information are generated for each code-block. Finally, in Tier-2 the resulting encoded code-blocks are organized and written to file following the rules of the JPEG 2000 syntax.

Decompression is the inverse process of compression. First, Tier-2 reads the encoded code-blocks from file. Next, Tier-1 decodes each code-block. Once all code-blocks have been decoded an inverse DWT is applied to the resulting data. Finally, postprocessing is applied to the result to obtain the raw image. Each of these steps is the inverse of its counterpart in the compression process.

As said any volumetric JPEG 2000 codec can also be used to encode and decode two-dimensional images. In the remainder of this paper, however, we will systematically refer to the elements of an image as voxels, which can be seen as the three-dimensional equivalent of pixels.

B. OpenCL

OpenCL [16] is an open standard maintained by the Khronos group for parallel programming of modern processors found in today's heterogeneous systems. Using OpenCL it is possible to write code that can run on GPUs of different brands, multi-core CPUs and even FPGAs. It is very similar to CUDA, a proprietary framework to program NVIDIA GPUs [14].

OpenCL models a heterogeneous system as a platform that consists of a host and of one or more devices that correspond to the accelerators that can be used. The main program runs on the host and controls all OpenCL related activities using an API. Code that runs on the devices is specified in a language based on C99 with a number of extensions to facilitate the parallel nature of the code but also with a number of limitations. This code is executed on the device by a number of work items or threads that are organized in work groups. The programmer must express the code as a kernel function that defines the work that needs to be executed by a single work item. The number of work items and their organization in work groups is also defined by the programmer.

To write efficient code for a device it is important to understand its architecture. Therefore, OpenCL presents a model of a device that should be kept in mind for efficiency. A device is represented as consisting of compute units which in turn are made up of processing elements. Memory management of the device is explicit. For this purpose OpenCL presents a memory model that is easily mapped to the way memory is organized on a typical GPU (Figure 2). Global memory is accessible to all the work items running the kernel. It is used to store the input and output data that must be transferred between the host and the device. Local memory can be shared by the work items of the same work group. Finally, private memory can only be accessed by a single work item. On GPUs "global data" is typically stored in the external GPU RAM. "Local data" resides in a fast L1 cache and "private data" is stored most of the time in registers. The latter two memory types are part of a compute unit. Accessing the GPU RAM is a lot slower than accessing the L1 cache which in turn is slower than registers. Therefore, it is important to store data in private or local memory. Nevertheless, using too much of them per work group may result in a lower efficiency because less work groups can be run concurrently. Finally, it should be noted that private arrays may be stored in the external GPU RAM thus making their access slower than expected. Luckily, modern GPUs come equipped with automatic L1 and L2 caches that somewhat alleviate this problem.

III. Related Work

There already exists work on the acceleration of JPEG 2000 compression with GPU. All of this work uses CUDA. A number of complete implementations of baseline JPEG 2000 encoding exist. The first of these by Balevic et al. is [5]. In the implementation of Ahmadvand and Ezhdehakosh [4] encoding of a code-block in Tier-1 is parallelized. It is, however, not clear how this was done. Finally, Ciznicki et al. [7] encode multi-dimensional data but use baseline JPEG 2000 to do so.

There exists also a lot of work on accelerating parts of JPEG 2000. Once again all of them use CUDA. For example, Franco et al. [8] and Galiano et al. [9] present implementations of three-dimensional DWT on GPU. The acceleration of Tier-1 is also a popular subject: Roto Le et al. [10] and Wei et al. [18] both present fine-grain implementations of Tier-1 coding.

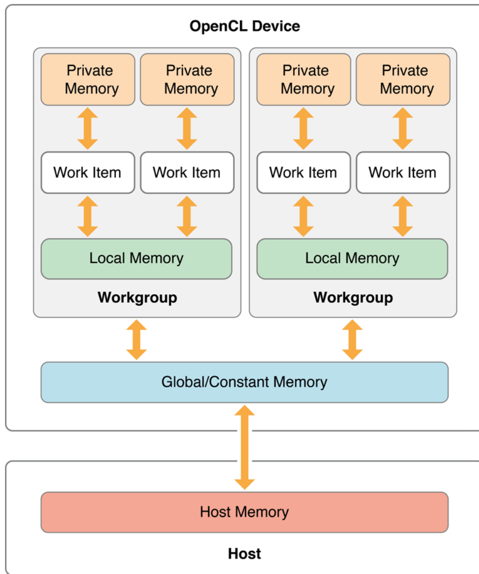

Fig. 2. OpenCL Memory Model

Matela et al. have done a lot of work on accelerating parts of JPEG 2000 using GPU. In [11] they focus on a two-dimensional DWT, while in [12] they reformulate Tier-1 in a way that it becomes appropriate for massive parallel architectures. Finally, in [13] they present an efficient implementation of arithmetic coding on GPU.

The main novelties of our work compared with the previous work concerns the acceleration of a volumetric codec both for encoding and decoding. Furthermore, we use OpenCL thus making it possible to use hardware of different types and different vendors.

IV. Methodology

A. Adapting the original application

There are a number of steps common to the acceleration of any application with modern accelerators using OpenCL.

- Identify the most compute intensive parts of the code. Most of the time they are known. If this is not the case profiling will quickly reveal the bottlenecks.
- Write OpenCL code to replace the compute intensive parts. Usually this corresponds to implementing the body of a loop in OpenCL. Sometimes, however, it may not be that straightforward e.g. certain functions contain loops where in the loop body a function is called that in turn contains another loop.
- Replace the deprecated functions by functions that launch the OpenCL kernels. This includes setting the arguments of the kernel and determining the size of the global and local work item space.
- Integrate OpenCL resource management in your program. OpenCL must be setup and torn down. Furthermore, memory needs to be allocated on the device and data must be transferred between host and device. Appropriate points for this should be identified keeping in mind that data transfers should be minimized as they contribute an important overhead to the application run time.

Typically, the last three steps represent an iterative process. The OpenCL code can only be tested once all the functions that launch them have been written. Once both have done, they have to be debugged and optimized.

B. Writing OpenCL code

We consider the development of OpenCL code along three axes that the programmer needs to keep in mind to write code with a satisfactory performance. In Section V, we will use this schema to discuss the OpenCL code we developed to accelerate our application.

1) Data placement: in OpenCL memory must be managed explicitly (see Subsection II-B). The programmer must decide for all data where it is placed keeping in mind the available resources and the resulting performance impact.
2) Work space configuration: it is necessary to determine how much work is assigned to a single work item. Furthermore, it must be decided how these work items are organized in work groups. The performance impact of this choice cannot be overestimated.
3) Architecture specific optimizations: although OpenCL strives to make it possible to write code that can be run across a variety of hardware, it may be necessary to apply architecture specific optimizations to maximize the performance on specific hardware.

Apart from these, a number of things must be kept in mind. First, it is necessary to write correct code. Depending on the code it can be very hard to do so immediately and debugging will be necessary. We used the `cl_amd_printf` pragma that enables the use of `printf` statements in OpenCL code. We could use this pragma both on AMD GPUs and on an Intel multi-core. We debugged our code on the multi-core because using the pragma on the AMD GPU slowed down the execution dramatically. The pragma is not supported on NVIDIA GPUs.

Secondly, when translating existing C code to OpenCL, it is necessary to take into account its restrictions. All constructions that are not supported by OpenCL have to be replaced by equivalent ones. A complete list of these restrictions can be found in [3]. In Subsection V-B we will present a number of techniques we applied to make existing code OpenCL compliant.

V. Implementation

As mentioned before we started from an existing JP3D codec written in ANSI C and modified the code to be accelerated with OpenCL. Doing so, we followed the methodology sketched in Section IV.

The most compute intensive parts of volumetric JPEG 2000 are the discrete wavelet transform and Tier-1. Nevertheless, we decided to port also the image preparation given that it concerns an embarrassingly parallel problem ideal for implementation on GPU. Note, in relation with Figure 1, that in our

case quantizing was done as part of Tier-1. We did not port Tier-2 because it contributes very little to the overall run time and concerns an inherently serial algorithm.

The data transfer between the host and the GPU is kept to a minimum: the raw image data is transferred once to the GPU before the processing starts, while the result is retrieved at the end of Tier-1. Finally, Tier-2 is run on the host as before to create the resulting JPEG 2000 file.

In the following sections we focus on the acceleration of the DWT and Tier-1 for coding. They provide practical examples of the principles discussed in Subsection IV-B. We do not discuss decoding in this section because of its significant similarity. In section VI, however, we show results for both coding and decoding.

A. Discrete Wavelet Transform

The Discrete Wavelet Transform or DWT is an important and computationally intensive part of the JPEG 2000 compression and decompression algorithm. Its goal is to divide the image into subbands and to decrease its entropy thus making it more suitable for compression. Our DWT implementation uses the lifting scheme. For clarity we briefly sketch how this scheme works on a one-dimensional sequence of samples. Depending on the type of wavelet transform one or two pairs of prediction and update lifting steps are executed alternately on these samples. First, a prediction step calculates the high-pass coefficients, positioned at odd indexes, from their respective even positioned neighbors. Subsequently an update step calculates the evenly positioned low-pass coefficients, using the respective high-pass neighbors that result from the prediction step. Due to the resulting low- and high-pass coefficients being interleaved, we then need to rearrange these coefficients. The low-pass coefficients are placed in the lower half of the sequence, the L-subband. The high-pass coefficients are placed in the upper half, the H-subband. The L-subband can be used to decode a low resolution version of the image while both subbands must be used to decode a full resolution version of the image. Depending on the desired number of resolution levels the process may be repeated on the L-subband.

A DWT on a three-dimensional image corresponds to 3 one-dimensional transforms, one in each direction on each one-dimensional sequence in the given direction e.g. on the rows in the X direction and on the columns in the Y direction. Note that for each resolution level a DWT is applied in every direction. We will refer to the different transformations in every direction as X-DWT, Y-DWT and Z-DWT.

The OpenCL code was written from scratch given the relative simplicity of the algorithm and the difference between a fine-grain parallel implementation and a sequential implementation. We created separate kernels for each DWT rather than fusing all kernels into one like for example [11]. Their solution, however, does not comply with the standard as it does not take into account value exchange between tile borders. As will become clear later, taking value exchange into account adds some overhead to the code. Furthermore, this overhead will become larger for a fused kernel and may possibly annul what is gained by a more complex fused kernel.

1) Data placement: Each voxel is accessed at least three times: once for the computation of its own DWT coefficient, once for the computation of its left neighbor coefficient and once for the computation of its right neighbor coefficient. Therefore, the best option is to let each work group store all voxels for which it needs to compute the DWT coefficients in local memory and to execute the computation in local memory. This is many times faster than doing the same in global memory. At the end of the kernel the results are written to global memory.

2) Work space configuration: A naive mapping would assign one voxel to each work item. But this is wasteful because during every step of the DWT only half of the voxels is updated. Therefore, one should assign at least two voxels to each work item. We chose four voxels as a performance compromise on the three GPUs we used for testing (for their details see Table II). On the AMD GPU the performance decreased for an increasing voxel count, while it increased on the NVIDIA Fermi GPU. Finally, on the NVIDIA Kepler GPU we obtained the best performance for four voxels per work item.

The work items must be organized in work groups. Initially, we used one-dimensional work groups and mapped these to the different rows, columns, ...The major disadvantage of this approach was the very bad global memory access of Y-DWT and Z-DWT due to the stride between voxels that are accessed in global memory by adjacent work items. To solve this problem we borrowed an idea from [5]: we use two-dimensional work groups of 16×16 work items whereby the first dimension of the work group is always mapped to the X-axis of the data. The other dimension will be mapped to the Y-axis or the Z-axis. In this configuration one work group is mapped to a tile of 64×16 or 16×64 voxels. But because the DWT coefficient of a voxel depends on its neighbors the image tiles overlap each other in the concerned direction. Thus, one work group computes 16×64 coefficients but only stores 16×56 of them.

3) Architecture specific optimizations: There are different ways to map the work item space on the data space. In our approach the first dimension of the work item space is always mapped on the X-axis of the data for optimal memory access, but the second and third dimensions could be mapped to either the Y- or Z-axis. For Z-DWT we tried both. Mapping the third dimension to the Z-axis was 33% faster on the AMD GPU than mapping to the Y-axis, while on the NVIDIA Fermi GPU it was 10% faster. On the NVIDIA Kepler it was 8% slower. These differences can be understood by considering that the mapping will determine the order of the work groups and the data they access. Therefore, one mapping might exploit the memory access subsystem of a GPU in a more efficient manner than the other. Because it is very difficult to predict the exact impact of a mapping we advice to experimentally determine the best choice.

Finally, we optimized the access of local memory for

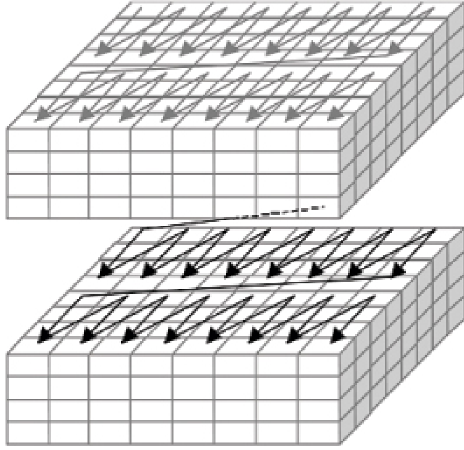

Fig. 3. Bit-plane traversal of code-blocks in Tier-1

NVIDIA GPUs. On these GPUs local memory is organized in 32 banks whereby consecutive words are stored in consecutive banks. Furthermore, the memory accesses of 32 work items are processed as one request. Therefore, access to local memory is optimal if all 32 work items access a different bank. If this is not the case bank conflicts arise incurring a performance penalty. To reduce the number of bank conflicts we changed the dimensions of the local memory used from 16×64 to 16×65. The resulting code ran 22% faster on the NVIDIA Fermi GPU 44% faster on the NVIDIA Kepler GPU. The change had no effect for the AMD GPU.

B. Tier-1

Tier-1 concerns the compression of the subbands. To do so they are divided into three-dimensional code-blocks that are processed independently. Each significant bit-plane of the code-block is traversed three times according to a predefined pattern (Figure 3). Each voxel is associated with three state variables that are updated during the traversal. Depending on its state and that of its neighbors, state updates are applied and code bits may be generated. Also, side information to guide Tier-2 is generated for each code-block.

As it is originally formulated, processing a code-block is inherently serial. There exists work that reformulates the algorithm such that it exhibits parallelism for a single code-block [12], [10], [18]. Although verifying the method and extending it to a third dimension might be relatively straightforward, rewriting the existing code to reflect those changes, while keeping the existing functionality, would have taken a long time. Furthermore, we could not find work that does the same for decompression.

In our approach, the available parallelism is limited by the number of code-blocks that need to be processed. For small images and large code-blocks the parallelism will be lowest. Because we use OpenCL, however, it is possible to use accelerators that are more appropriate for this kind of coarse-grain parallelism, without changing the code. We ran our code on an Intel multicore CPU. The resulting performance is included in Section VI.

TABLE I
OPTIMAL CONFIGURATION FOR DIFFERENT IMAGES. THE CONFIGURATION IS SHOWN AS THE NUMBER OF WORK GROUPS TIMES THE WORK GROUP SIZE

Image	**AMD** 28 compute units	**NVIDIA Tesla** 14 compute units	**NVIDIA GeForce** 4 compute units
flower_foveon	221×4	111×8	56×16
artificial	193×8	97×16	97×16
bridge	178×16	89×32	45×64
big_tree	434×16	109×64	109×64

The OpenCL code is based on a function in the original code that is called to encode a single code-block. Translating the code of this function and all the functions on which it depends - about 1500 lines of ANSI C - to OpenCL proved to be the most labor intensive part of the project.

1) Data placement:

- The code-block coefficients are read from global memory and stored in a private array. As mentioned in Subsection V-B these arrays are typically stored in the GPUs RAM but cached in fast L1 cache. Furthermore, this way of working is closely related to the original implementation where the code-block DWT coefficients are first stored in a temporary buffer.
- The state variables are also stored in a private array. We did consider the possibility to use local memory rather than private arrays but the large amount of data for one work item - 4 KB for the state variables and 16 KB for the DWT coefficients - would decrease the potential occupancy and hence decrease the resulting performance dramatically.
- The code words were written immediately to global memory. Choosing some intermediate and faster memory proved more difficult because encoded bits are generated at different locations in the code. Furthermore, because we are compressing data it is impossible to say how much memory is needed for the output. Our solution was to allocate as much memory for the output as for the input and to foresee for each code-block as much memory as for the code-block itself.

2) Work space configuration: For a certain image size the work group size makes a big impact on the performance of our implementation. The work groups may not be too big for two reasons. First, due to the large amount of resources used by one work item, large work groups may cause a low occupancy, hence decreasing performance. Secondly, the total number of work groups will be too small to have an equal distribution of work groups among the compute units. This will be especially the case for small images and devices with a great number of compute units. This is illustrated by table I that shows the optimal configuration for the tested GPUs for the 2-dimensional test images (their details can be found in Table II) using 64×64 code-blocks. Because the choice seems to be determined mainly by the size of the image and the device on which the code is run, the optimal work group size can be decided depending on the image that is to be encoded.

TABLE II
2D AND 3D TEST IMAGE CHARACTERISTICS

Image Name	Dimensions	Size [MB]
flower_foveon	2268×1512	9.8
artificial	3072×2048	18.0
bridge	2749×4049	31.8
big_tree	6088×4555	79.3
mri_ventricles	$256 \times 256 \times 124$	7.8
mrt8_angio2	$256 \times 320 \times 128$	10.0
mrt8_angio	$412 \times 512 \times 112$	22.5
stent8	$512 \times 512 \times 174$	43.5
backpack8	$512 \times 512 \times 373$	93.3
vertebra8	$512 \times 512 \times 512$	128.0

3) Translating C to OpenCL: The greatest difficulty encountered was to replace constructions that are not supported by OpenCL. We illustrate the measures we took for three instances of this problem:

- **Self-reference.** A state transition table was implemented as an array of structs that used self-referencing pointers to determine the state to be taken according to the symbol encountered. We replaced the array of structs by different arrays of which two are used to hold the indexes of the next state for symbols 0 or 1. Of course, all references to this array throughout the code had to be modified accordingly.
- **Dynamic memory allocation.** A state register list that is constructed by the arithmetic coder and that is used to harvest side-information for Tier-2 was implemented as a linked list using self-referencing structs and dynamic memory allocation. This construction was replaced by allocating a fixed size array as a member of the arithmetic coder and to keep a current index to keep track of the head of the list. The concerned code was updated accordingly.
- **Function pointer.** The bit plane traversal function used a function pointer argument to call the appropriate function according to which pass was being applied. We replaced the function pointer by using a type argument and calling the appropriate function from a `switch` statement on this type.

VI. RESULTS

A. Test Setup

The performance of the codec was tested on two-dimensional color images from [2] and three-dimensional gray-scale images from [1]. Table II shows the image dimensions and the resulting size.

The OpenCL code was run on three different GPUs and a multi-core CPU. Their characteristics are listed in table III.

Apart from the complete codec run times shown at the end, all results concern coding or decoding of a single component. This is especially important for the two-dimensional images, which consist of three color components. All results are for lossless coding.

TABLE IV
PERFORMANCE OF ONE DWT

GPU	2D throughput	3D throughput
Tesla C2050	3293 MPix/s	1850 MVox/s
GeForce GTX 650 Ti	3263 MPix/s	1512 MVox/s
AMD Radeon HD 7950	8577 Mpix/s	5038 MVox/s

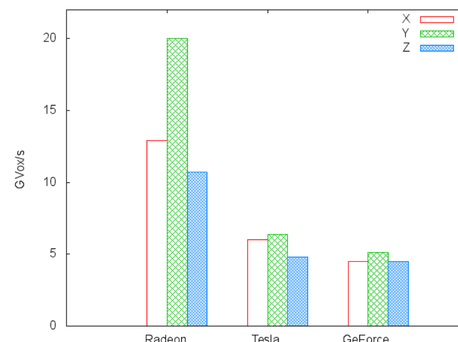

Fig. 4. Performance of X-, Y- and Z-DWT on a $512 \times 512 \times 512$ image

B. DWT

Table IV shows the performance of a single DWT using a 5×3 wavelet, both for two- and three-dimensional images. The AMD GPU outperforms the NVIDIA GPUs because of its higher memory bandwidth and its greater number of compute units.

Figure 4 shows the performance achieved in each direction for the $512 \times 512 \times 512$ image. For all GPUs the Y-DWT has the best performance. In this case the adjacent voxels accessed by groups of 16 work items begin at boundaries that are multiples of 16 bytes. This is not the case for X-DWT because the accessed tiles overlap along the X-axis. Therefore the adjacent voxels may belong to different memory segments, hence causing extra memory transactions. The Z-DWT performs worse on the AMD and the NVIDIA Fermi GPUs. This can probably be explained by the very large distance between chunks of data that are accessed by a group of work items. Because on GPUs data access of a group of work items is handled as one request, it is more advantageous if the accessed elements lay close to each other in memory. The NVIDIA Kepler GPU is least impacted by the direction of the DWT.

The speedups shown in figure 5 concern both forward DWT (coding) and inverse DWT (decoding) for both two- and three-dimensional images. Here we consider the complete DWT process that includes several levels of DWT and the data transfers necessary on the GPU because we cannot perform the DWT in-place. The speedup dip for the *mrt8_angio* image is caused by a performance peak of the sequential implementation.

C. Tier-1

Table V shows the performance of Tier-1 in Mpix/s for the different devices and for two- and three-dimensional images. A higher throughput is obtained for larger images. To illustrate

TABLE III
HARDWARE USED FOR OUR TESTS.

Architecture	Fermi	Kepler	GCN	Bloomfield
Vendor	NVIDIA	NVIDIA	AMD	Intel
Name	Tesla C2050	GeForce GTX650 Ti	Radeon HD 7950	iCore 7
# compute units	14	4	28	8
# compute elements	448	768	1792	8
Clock frequency (MHz)	1150	928	800	3200
Memory Bandwidth (GB/s)	144	86	240	26

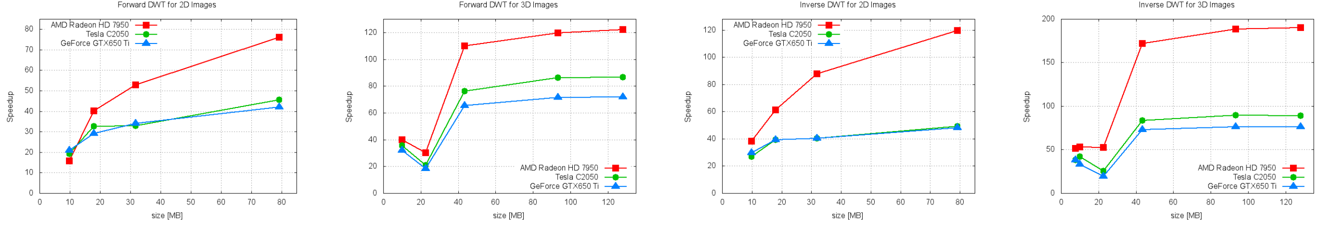

Fig. 5. Speedup of DWT for 2-dimensional and 3-dimensional images

TABLE V
PERFORMANCE OF TIER-1 FOR DIFFERENT DEVICES AND DIFFERENT CODE-BLOCK SIZES

GPU	block size	2D throughput	3D throughput
Intel iCore 7	2^{12}	16 - 24 MPix/s	16 - 43 MVox/s
	2^{10}	16 - 24 MPix/s	14 - 36 MVox/s
Tesla C2050	2^{12}	11 - 29 MPix/s	15 - 83 MVox/s
	2^{10}	22 - 55 MPix/s	27 - 78 MVox/s
GeForce GTX 650 Ti	2^{12}	9 - 20 MPix/s	14 - 56 MVox/s
	2^{10}	20 - 29 MPix/s	21 - 51 MVox/s
AMD Radeon HD 7950	2^{12}	7 - 14 MPix/s	5 - 40 MVox/s
	2^{10}	15 - 30 MPix/s	10 - 87 MVox/s

the impact of the code-block size we include numbers for two different code-block sizes: 4096 voxels and 1024 voxels. For smaller code-blocks the total amount of work is greater because the achieved compression is smaller. This is made clear by the better performance of the multi-core for larger code-blocks. On the other hand smaller code-blocks are more beneficial for the GPU especially for small images and for images exhibiting a high entropy. Using smaller code-blocks increases the throughput for all two-dimensional images on all GPUs. On the NVIDIA GPUs and for large three-dimensional images using smaller code-blocks is not advantageous because the greater amount of overall work undoes the effects of the increased parallelism. On AMD GPUs smaller code-blocks are always beneficial. This is explained by the fact that the AMD GPU is the most “fine-grained” device i.e. it has the highest number of compute units, and the smallest amount of resources per compute unit.

Figure 6 shows the speedup obtained for Tier-1 both for coding and decoding. For coding we also show numbers for the Intel iCore7. As expected we get a better speedup on GPU for larger images because of the greater parallelism. The same observation is valid when comparing the two-dimensional and three-dimensional images: for the same size the three-dimensional images contain more voxels than

TABLE VI
CODING PERFORMANCE IN MVOXELS PER SECONDS FOR THE ORIGINAL VERSION, THE GPU ACCELERATED VERSIONS AND THE HYBRID ACCELERATED VERSION

image	seq	tesla	geforce	amd	hybrid
flower_foveon	4.4	10.2	8.8	2.7	18.6
artificial	4.6	14.6	12.8	4.6	19.6
bridge	2.9	15.1	14.3	6.0	14.8
big_tree	2.7	24.0	18.0	8.4	14.7
mri_ventricles	2.7	12.1	11.8	2.5	12.6
mrt8_angio2	3.1	14.9	11.8	3.1	13.7
mrt8_angio	3.0	22.3	11.3	6.2	14.3
stent8	4.1	29.6	18.2	10.5	22.3
backpack8	4.4	32.1	21.1	15.0	24.2
vertebra8	5.3	45.9	26.9	23.7	30.6

the two-dimensional images contain pixels because the two-dimensional images are color images and the components are processed serially. It is interesting to note that the AMD GPU is better at decoding images. We believe this is caused by the fact that Tier-1 requires less resources for decoding than for coding.

D. Overall

Tables VI and VII show the throughput of the different codec versions for lossless coding of two- and three-dimensional images using code-blocks of 4096 voxels The hybrid version is one where image preparation and discrete wavelet transform are run on the AMD GPU and Tier-1 is run on the multi-core. Note we do not take into account some sources of overhead concerning the set up of OpenCL and the compilation of the OpenCL code. On the AMD GPU it took 3 seconds to compile the OpenCL code, while on the multi-core it took 0.4 seconds. NVIDIA cached the compiled kernels, however, we needed 0.3 seconds to set up OpenCL.

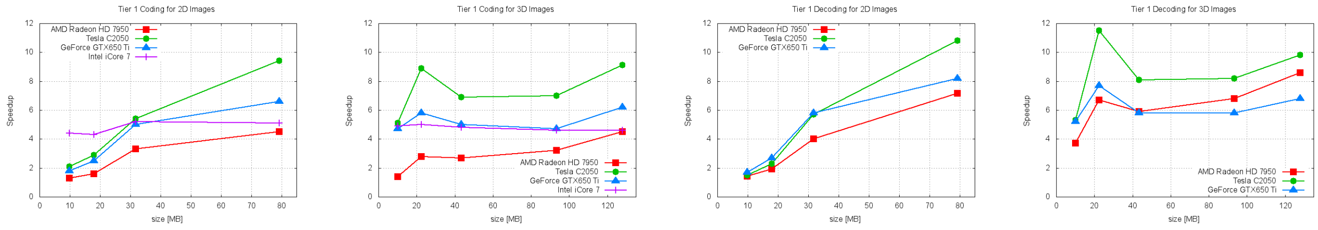

Fig. 6. Speedup of Tier-1 for 2-dimensional and 3-dimensional images

TABLE VII
DECODING PERFORMANCE IN MVOXELS PER SECONDS FOR THE DIFFERENT VERSIONS

image	seq	tesla	geforce	amd
flower_foveon	4.5	10.5	9.2	7.1
artificial	4.8	14.6	13.8	11.1
bridge	2.9	14.7	14.4	12.2
big_tree	2.7	22.8	18.1	20.2
mri_ventricles	2.8	14.8	14.6	10.2
mrt8_angio2	3.4	20.0	17.5	12.9
mrt8_angio	3.1	32.6	22.9	20.9
stent8	4.5	40.7	29.9	32.3
backpack8	5.2	51.6	33.6	41.6
vertebra8	5.5	69.5	51.5	69.6

VII. CONCLUSION

In this paper we presented and illustrated a methodology to accelerate an existing sequential software using OpenCL and modern accelerators such as GPUs and multi-core CPUs. We considered the development of the OpenCL code along three axes to obtain a satisfactory performance: data placement, work space configuration and architecture specific optimizations.

We used a volumetric JPEG 2000 codec as a real-world example of software acceleration using OpenCL. Our approach resulted in obtaining a decent speedup for a relatively small effort. Furthermore, a number of interesting observations were made concerning the acceleration of the two most important parts of the codec: DWT and Tier-1. The former is ideal for acceleration on GPU and dramatic speedups are achieved, while the parallelism of the latter is too coarse-grain for the GPU. Therefore, the acceleration of Tier-1 was also run on the multi-core obtaining good speedups. Despite the coarse granularity of Tier-1 we achieved a good speedup. Encoding of large images was accelerated almost 9 times using an NVIDIA Tesla C2050, while decoding of large images was accelerated almost 13 times using both an NVIDIA Tesla C2050 and an AMD Radeon HD 7950.

In the future we hope to improve our implementation by the introduction of streaming and further optimization techniques.

ACKNOWLEDGMENT

This work was supported by the iMinds ICON MMIQQA project, “Multimodel Microscopic Imaging: Quality, Quantification and Acceleration”.

REFERENCES

[1] New “real world” medical datasets. http://www.volvis.org/. Accessed: 2014-07-01.

[2] The new test images - image compression benchmark. http://www.imagecompression.info/test_images/. Accessed: 2014-07-01.

[3] Opencl 1.2 reference pages. http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/. Accessed: 2014-09-01.

[4] M. Ahmadvand and A. Ezhdehakosh. Gpu-based implementation of jpeg2000 encoder. In *PDPTA 2012 Proceedings*, pages 682–688, 2012.

[5] Ana Balevic, Armin Weiss, Martin Heide, Simon Papandreou, and Norbert Fuerst. Cuj2k: Jpeg2000 encoder on cuda. http://cuj2k.sourceforge.net/. Accessed: 2014-07-01.

[6] Tim Bruylants, Adrian Munteanu, Alin Alecu, Rudi Deklerck, and Peter Schelkens. Volumetric image compression with jpeg2000. In *SPIE The International Society for Optical Engineering*, 2007.

[7] Milosz Ciznicki, Krzysztof Kurowski, and Antonio Plaza. Gpu implementation of jpeg2000 for hyperspectral image compression. In *SPIE Remote Sensing*, pages 81830H–81830H. International Society for Optics and Photonics, 2011.

[8] Joaquín Franco, Gregorio Bernabé, Juan Fernández, and Manuel Ujaldón. Parallel 3d fast wavelet transform on manycore gpus and multicore cpus. *Procedia Computer Science*, 1(1):1101 – 1110, 2010. ICCS 2010.

[9] V. Galiano, O. López, M.P. Malumbres, and H. Migallón. Gpu-based 3d wavelet transform. 2012.

[10] Roto Le, IR. Bahar, and J.L. Mundy. A novel parallel tier-1 coder for jpeg2000 using gpus. In *Application Specific Processors (SASP), 2011 IEEE 9th Symposium on*, pages 129–136, 2011.

[11] Jiří Matela. GPU-Based DWT Acceleration for JPEG2000. In *MEMICS 2009 Proceedings*, pages 136–143, Brno, 2009.

[12] Jiri Matela, Vit Rusnak, and Petr Holub. Efficient jpeg2000 ebcot context modeling for massively parallel architectures. In *Data Compression Conference (DCC), 2011*, pages 423–432. IEEE, 2011.

[13] Jiří Matela, Martin Šrom, and Petr Holub. Low gpu occupancy approach to fast arithmetic coding in jpeg2000. In *Mathematical and Engineering Methods in Computer Science*, pages 136–145. Springer, 2012.

[14] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, 2014.

[15] Peter Schelkens, Athanassios Skodras, and Touradj Ebrahimi, editors. *The JPEG 2000 Suite*. John Wiley & Sons, Chichester, UK, 2009.

[16] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[17] D. Taubman and M. Marcellin. *JPEG2000 Image Compression Fundamentals, Standards and Practice: Image Compression Fundamentals, Standards and Practice*. The Springer International Series in Engineering and Computer Science. Springer US, 2001.

[18] Fang Wei, Qiu Cui, and Ye Li. Fine-granular parallel ebcot and optimization with cuda for digital cinema image compression. In *ICME*, pages 1051–1054. IEEE, 2012.