Heterogeneous acceleration of volumetric JPEG 2000 using OpenCL

Cornelis, Jan G.; Lemeire, Jan; Bruylants, Tim; Schelkens, Peter

Published in:
International Journal of High Performance Computing Applications

DOI:
10.1177/1094342016646438

Publication date:
2017

Heterogeneous acceleration of volumetric JPEG 2000 using OpenCL

Jan G. Cornelis[1,3], Jan Lemeire[1,2,3], Tim Bruylants[1,2], and Peter Schelkens[1,2]

[1] *Vrije Universiteit Brussel (VUB), Electronics and Informatics (ETRO) Dept.*
[2] *Vrije Universiteit Brussel (VUB), Dept. Of Industrial Sciences (INDI)*
[3] *iMinds, Multimedia Technologies Dept., Belgium*

Abstract

This paper discusses an OpenCL version of a volumetric JPEG 2000 codec that runs on GPUs, multi-core processors or a combination of both. Since the performance critical part consists of a fine-grained (discrete wavelet transform) and coarse-grained algorithm (Tier-1), the best performance is obtained with a hybrid execution in which the discrete wavelet transform is executed on a GPU and Tier-1 on a multi-core. Using an Intel i7 multi-core in combination with a modest NVIDIA Quadro K620 GPU yields speedups greater than 10 compared with the original sequential code. The performance bottlenecks that arise on GPUs when parallelizing algorithms that are coarse-grained by nature are discussed and also the optimizations that are possible. A performance analysis reveals the inefficiencies and explains the deviations from the GPU peak performance.

1 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 to process them have become more and more complex. At the same time processors have become parallel, introducing the necessity to write parallel code to exploit their processing power. Furthermore, it has become possible to use different hardware accelerators, like GPUs, Intel's Xeon Phi and FPGAs, to speed up code execution. In this context, it is important to write code that can exploit the computing power of a given platform in an efficient manner. This should be done in a functionally and if possible performance portable way to minimize the maintenance overhead.

OpenCL is a standard developed in 2008 for parallel programming a variety of hardware accelerators. Contrary to NVIDIA's CUDA, a proprietary framework used to develop and run general purpose code on NVIDIA GPUs, OpenCL is an open standard. It holds the promise of writing code that can be run across a wide range of hardware accelerators such as GPUs, multi-core CPUs, FPGAs and other accelerators, like the Intel MIC. These properties make it ideal to develop software that can adapt easily to the platform specificities and that can exploit all its available computing resources.

JPEG 2000 Part 10 or JP3D is an extension to the JPEG 2000 image compression standard to compress volumetric images. JPEG 2000 provides superior compression performance and advanced functionality compared to JPEG, but unfortunately this comes at the cost of increased algorithmic complexity. This complexity together with the increasing size of the images to be processed make a JPEG 2000 codec a perfect example of a program in need of lots of computational power.

An existing volumetric JPEG 2000 codec was accelerated with OpenCL to obtain a parallel codec that can exploit both the processing power of discrete or integrated GPUs and of multi-core CPUs. This paper provides a detailed discussion of the methodology followed and the steps taken to modify an existing sequential application written in ANSI C using OpenCL. The main factors that impact the performance on these processors are emphasized and a detailed performance study of the most challenging parts of the codec is presented. The obtained results are interesting in themselves as they concern, to our knowledge, the only parallel implementation of a JP3D codec. Furthermore, contrary to existing work on JPEG 2000 compression and decompression our implementation can be run either on a multi-core CPU, a GPU or a combination of both in order to obtain the best possible performance.

Most of this paper focuses on the acceleration of compression. Nevertheless, decompression was also accelerated using the same methodology and results for both compression and decompression will be shown.

The remainder of this paper is organized as follows: Section 2 discusses JPEG 2000 Part 10 and OpenCL. Related work is presented in Section 3. Section 4 presents the methodology followed, the parallel implementation and the applied optimization techniques. Section 5 reports on the results obtained for a variety of images that were encoded and decoded using different hardware accelerators. Section 6 presents a detailed performance discussion in which the performance of the ported code on different hardware accelerators is analyzed. Finally, Section 7 concludes the paper.

2 Background

2.1 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. JPEG 2000 is typically deployed in professional applications. For a thorough discussion of its principles we refer to [Taubman and Marcellin(2002)]. A comprehensive overview of the baseline JPEG 2000 standard and its extensions can be found in [Schelkens et al.(2009)].

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 - ([Bruylants et al.(2007)]) is a three-dimensional extension to support the encoding of volumetric data sets. The IRIS-JP3D codec[1], which was accelerated, is an implementation of Part 1 and Part 10 of the standard. It can encode and decode both two-dimensional and three-dimensional images.

[1]`http://www.irissoftware.be)`

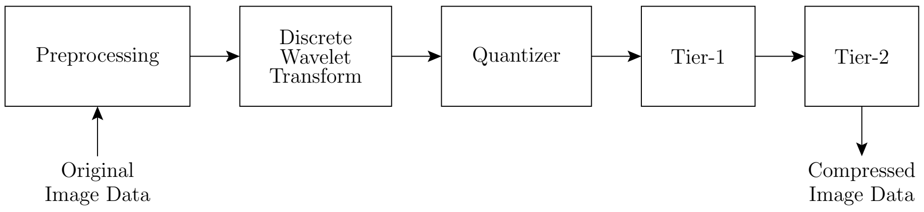

Figure 1: Block diagram of a JP3D encoder.

How a JP3D codec works can be briefly explained 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 (DWT) is applied. This results in a decomposition of the image in subbands. These subbands are further partitioned in three-dimensional code-blocks that are processed by the EBCOT Coder in Tier-1. For lossy compression a quantization is applied to the wavelet coefficients before Tier-1. During Tier-1, elementary bitstreams and side information are generated for each code-block. Finally, in Tier-2 the resulting bitstreams are reorganized and embedded in a codestream container according to the rules of the JPEG 2000 syntax specification. Decompression is the inverse process of compression. The steps are executed in the inverse order and each step performs the inverse operation of its encoding counterpart.

Because both two-dimensional and three-dimensional images are considered in this paper, image elements of both image types will be referred to as voxels, which can be seen as the three-dimensional equivalent of pixels.

2.2 OpenCL

OpenCL ([Stone et al.(2010)]) 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 similar to CUDA, a proprietary framework to program NVIDIA GPUs ([Nickolls et al.(2008)]).

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 the OpenCL API. Code that runs on the devices is specified in OpenCL C, 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 necessary to have a basic understanding of its architecture. Therefore, OpenCL presents a model of a device that should be kept in mind for efficiency. A device is represented as consisting

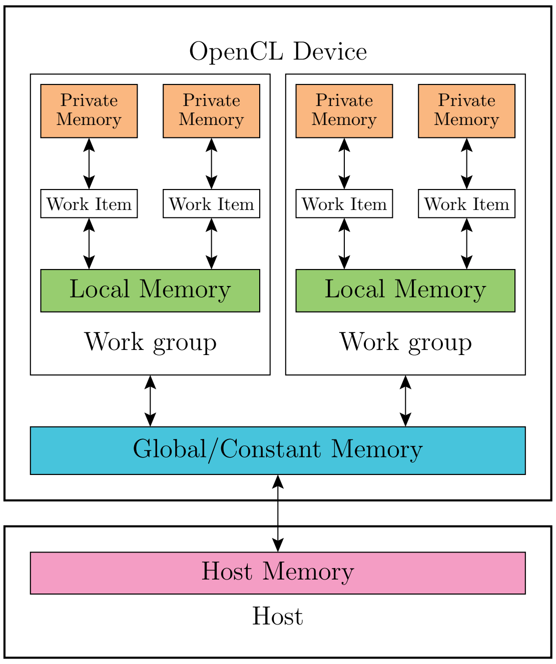

Figure 2: OpenCL memory model.

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 memory organization of 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. 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. However, the number of work groups that can run concurrently depends on their memory requirements. Too high requirements will result in a lower occupancy and lower efficiency. Finally, it should be noted that private arrays may be stored in the external GPU RAM if they do not fit in registers or L1 cache ([NVIDIA(2009)]).

3 Related Work

A body of work concerns the acceleration of JPEG 2000 encoding on GPU. All of this work uses CUDA. A number of complete implementations of baseline JPEG 2000 encoding exist. The first of these is [Balevic et al.(2009)]. In the implementation of [Ahmadvand and Ezhdehakosh(2012)] encoding of a code-block in Tier-1 is parallelized. Unfortunately, the paper does not provide details on the parallelization strategy. Finally, [Ciznicki et al.(2011)] encode multi-dimensional data but use baseline JPEG 2000 to do so.

Many works focus on accelerating separate parts of JPEG 2000. Once again all of them use CUDA. [Franco et al.(2010)] and [Galeano et al.(2012)] present implementations of three-dimensional DWT on GPU. The acceleration of Tier-1 is also a popular subject: [Le et al.(2011)] and [Wei et al.(2012)] both present fine-grained implementations of Tier-1 encoding.

Matela et al. have done a lot of work on accelerating parts of JPEG 2000 using GPU. In [Matela(2009)] they focus on a two-dimensional DWT, while in [Matela, Rusnak and Holub(2011)] they reformulate context modeling, a part of Tier-1, such that it becomes appropriate for massive parallel architectures. Finally, in [Matela, Šrom and Holub(2011)] they present an efficient implementation of arithmetic coding, another part of Tier-1, on GPU. In this paper, they report run times to compress images both on their implementation and existing sequential and parallel implementations of JPEG 2000.

In [Ciznicki et al.(2014)] the performance of different JPEG 2000 implementations is compared. Interestingly, they use the same two-dimensional images as the ones of this work. Among others the encoders from [Balevic et al.(2009)] and [Ciznicki et al.(2011)] are included in their tests.

The work described in this paper differs from existing work in two ways. First, it concerns the acceleration of a volumetric codec both for encoding and decoding. Secondly, because of the use of OpenCL the accelerated codec can use both GPUs and multi-core CPUs to obtain the best possible performance across a large variety of hardware platforms.

4 Parallelization

4.1 Granularity

One of the most important aspects of parallelization is the *granularity* of the parallel decomposition. As will be discussed in detail GPUs are fine-grained, highly parallel devices that only attain maximal performance for fine-grained parallel programs. In this context, it is important to define the granularity of the problem decomposition and the granularity of the hardware device that performs multi-threading.

The granularity of a problem decomposition is determined by the number and size of tasks into which the problem is decomposed ([Grama et al.(2002)]). A problem decomposition is fine-grained if there are many small tasks.

Fine-grained multi-threading switches between threads on each instruction, resulting in interleaved execution of multiple threads. Efficient fine-grained multi-threading is only possible if switching between threads is virtually free. Coarse-grained multi-threading switches threads only on costly stalls, such as second level cache misses. Consequently, the requirements on thread switching are less restrictive ([Patterson and Hennessy(2012)]). Another way to look at this distinction is to say that fine-grained multi-threading architectures maximize throughput while coarse-grained architectures minimize latency.

Multi-core CPUs implement coarse-grained multi-threading, while GPUs implement fine-grained multi-threading. Furthermore, GPUs execute in SIMT fashion - multiple threads execute the same code - resulting in a need for regular computations and data access. On GPUs branching, code that causes different threads to follow different execution paths, can deteriorate the performance dramatically. The impact of granularity differences on the execution of the code is discussed in detail in section 6.

4.2 Adapting the original application

Modifying an application with OpenCL is an iterative process consisting of a number of different steps. First, it is necessary to identify the parts of the code that are most time consuming. For these parts it is determined whether they can be decomposed and if so what the granularity of such a decomposition is. If a fine-grained decomposition is possible, the OpenCL implementation is targeted towards GPUs, otherwise multi-core CPUs are targeted. After implementing the concerned parts in OpenCL, the original functions are replaced by functions that launch the OpenCL kernels. This includes setting the arguments of the kernel and determining the sizes of the global and local work item space. Finally, it will also be necessary to integrate OpenCL management: OpenCL must be set up and torn down. Furthermore, memory needs to be allocated on the devices and data must be transferred between different memory spaces. Appropriate points to do this must be identified keeping in mind that data transfers should be minimized as they add an overhead to the program that cannot be neglected. Once the aforementioned steps have been performed, it is necessary to test the new implementation. Profiling the new implementation will reveal new bottlenecks paving the way towards further optimizations.

4.3 Writing OpenCL code

To produce OpenCL code with satisfactory performance, optimizations are organized along three axes:

1. Data placement: in OpenCL memory must be managed explicitly (see Subsection 2.2). 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 class specific optimizations: rather than optimizing code for a specific GPU or a specific CPU, only optimizations are implemented that are expected to have a beneficial or no effect on all platforms belonging to a specific class of architectures i.e. a distinction is made between optimizations for fine-grained GPUs and optimizations for coarse-grained multi-core processors. The need of device specific OpenCL device code is discussed in more depth in [Seo et al.(2011)].

Furthermore, it is important to - besides writing correct code - take into account the restrictions of OpenCL when translating C code to it. All constructions that are not supported by OpenCL must be replaced by equivalent ones.

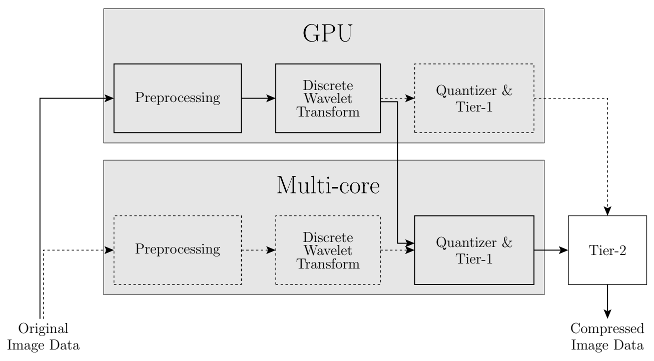

Figure 3: Possible codec configurations. The preferred configuration is suggested by the thick lines.

A complete list of these restrictions can be found in [Khronos(2012)]. In Subsection 4.6 a number of techniques that were applied to make the code OpenCL compliant are presented.

4.4 Implementation

The hybrid codec is based on IRIS-JP3D, an existing JP3D codec written in ANSI C. It was accelerated with OpenCL following the methodology sketched in Section 4.2.

The most time consuming parts of volumetric JPEG 2000 are the DWT (Discrete Wavelet Transform) and Tier-1. For the largest three-dimensional test image they contribute 26% and 68% of the run time respectively. Also image preprocessing has been parallelized: it concerns a level-shift and, for multi-component images, a component transformation, both embarrassingly parallel problems. Quantizing is done as part of Tier-1. Tier-2 is not ported because it concerns an inherently serial algorithm and contributes little to the overall run time. As will become clear, image preprocessing and DWT are both fine-grained algorithms, while Tier-1 is coarse-grained. Therefore, the implementations of image preprocessing and DWT are targeted towards the GPU, while Tier-1 is targeted towards the multi-core CPU. This explains the preferred codec configuration shown in Figure 3. As the figure suggests, all OpenCL parts may be run either on the GPU or on the multi-core. This is especially interesting for Tier-1: for large images the absolute performance of Tier-1 can be greater on high-end GPUs than on multi-core processors.

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 either at the end of the DWT or at the end of Tier-1. To minimize modifications to the existing software, memory optimizations such as the use of mapped memory were not applied. Finally, Tier-2 is run on the host as before to create the resulting JPEG 2000 file.

The following sections focus on the acceleration of the DWT and Tier-1 for encoding. They provide practical examples of the principles discussed in Subsection 4.3. Decoding is not discussed because of its significant similarity. Section 5 shows results for both encoding and decoding.

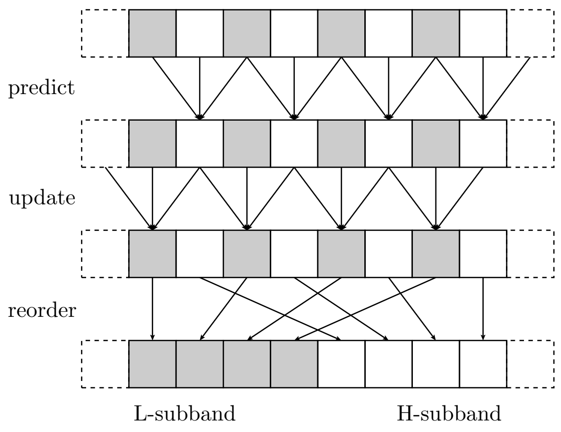

Figure 4: Lifting scheme for 5 × 3 DWT.

4.5 Discrete Wavelet Transform

The DWT is an important and computationally intensive part of the JPEG 2000 algorithm. The DWT decomposes the image into subbands, which results in a sparse representation that depicts a lower entropy than the original spatial domain representation. The DWT implementation uses the lifting scheme ([Sweldens(1996)]). Figure 4 shows how this scheme works on a one-dimensional sequence of 8 samples using a 5 × 3 wavelet. 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, they need to be reordered. 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 is repeated on the L-subband.

A DWT on a three-dimensional image corresponds to 3 one-dimensional transforms, one in each direction on a one-dimensional array in a given direction (X, Y or Z). Note that for each resolution level a DWT on the low-pass subband is applied in every direction. In the remainder of this paper the different transforms are referred to as X-DWT, Y-DWT and Z-DWT. X-DWT, for example, corresponds to performing the transform on all rows of the image.

The OpenCL code was written from scratch given the difference between a fine-grained parallel implementation and a sequential implementation. Separate kernels for each DWT were created rather than fusing all kernels into one like for example in [Matela(2009)]. This implementation, however, does not comply with the standard given that it does not account for value exchange between borders of the image partitions that are processed by separate work groups. As will become clear later, taking value exchange into account adds overhead to the code. Furthermore, although writing a two-dimensional fused kernel might still be feasible although more complex, the additional third dimension increases both the overhead introduced due to the overlap and the complexity of the code exponentially. Next, the optimization steps along the three axes are discussed.

4.5.1 Data placement

Each voxel is accessed at least four times: twice 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. This will speedup the computation substantially compared to doing the same in global memory. At the end of the computation, the results are written to global memory.

4.5.2 Work space configuration

A naive mapping assigns one voxel to each work item. 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. The choice of four voxels is a performance compromise between the three GPUs used for testing (for their details see Table 2). 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 the best performance was obtained when mapping four voxels to every work item.

The work items must be organized in work groups. Initially, one-dimensional work groups were used and mapped onto the voxels in the appropriate direction (X, Y or Z). A major disadvantage of this approach is the pessimal 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 an idea was borrowed from [Balevic et al.(2009)]: two-dimensional work groups of 16×16 work items are used whereby the first dimension of the work group is always mapped to the X-axis of the data. The other dimension is mapped to the Y-axis or the Z-axis. In this configuration one work group is mapped to an image partition of 64×16 or 16×64 voxels. But because the DWT coefficient of a voxel depends on its neighbors the image partitions must overlap each other in the concerned direction. Thus, one work group computes 16×64 coefficients but only stores 16×56 of them.

4.5.3 Architecture specific optimizations

There are different ways to map the work item space on the data space. Of course, 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 both were tried. 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, it is best to experimentally determine the best choice.

Finally, the access of local memory was optimized for 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 the dimensions of the array in local memory were changed from 16×64 to 16×65. The resulting code ran 22% faster on the NVIDIA Fermi GPU and 44% faster on the NVIDIA Kepler GPU. The change had no effect for the AMD GPU: as mentioned in [AMD(2012)] on GPUs of the GCN architecture family possible conflicts are resolved in hardware.

4.6 Tier-1

Tier-1 concerns the compression of the subbands created by the DWT. To do so they are divided into code-blocks that are processed independently. Depending on the dimensionality of the image, two- or three-dimensional code-blocks are used. The size of a code-block is somewhat restricted: two-dimensional code-blocks may contain up to 2^{12} DWT coefficients, while their width and height must be a power of 2. Three-dimensional code-blocks may contain up to 2^{18} coefficients. A code-block is sliced into bit-planes and each significant bit-plane of the code-block is traversed three times according to a predefined pattern (Figure 5). A bit is processed - possibly leading to the emission of code bits - during one of the three passes depending on its state and the state of its neighbors. Because this state depends on the traversal up to this point, entropy coding is an inherently sequential process. There exist three special modes (restart, reset and causal) that allow for more parallelism ([Taubman and Marcellin(2002)]), however, our codec does not use these modes because they decrease the compression performance and at the same time increase the code complexity. There exists work that reformulates the algorithm such that it exhibits parallelism for a single code-block ([Matela, Rusnak and Holub(2011)], [Le et al.(2011)], [Wei et al.(2012)]). This is only possible for encoding because it requires complete information, for decoding an equivalent approach is impossible.

Because of the above, processing a complete code-block defines the finest possible granularity of a parallel decomposition. In the OpenCL version of Tier-1 a single work item encodes a complete code-block. Therefore, the preferred execution of Tier-1 is on a multi-core CPU. Nevertheless, it is possible to execute Tier-1 on a GPU. Section 5 shows results both for multi-core CPUs and for a number of GPUs.

The OpenCL code is a translation of the original function 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. The greatest difficulty encountered was to replace constructions that are not supported by OpenCL. The following measures were taken for the three most important transformations:

- **Self-reference.** A state transition table was implemented as an array

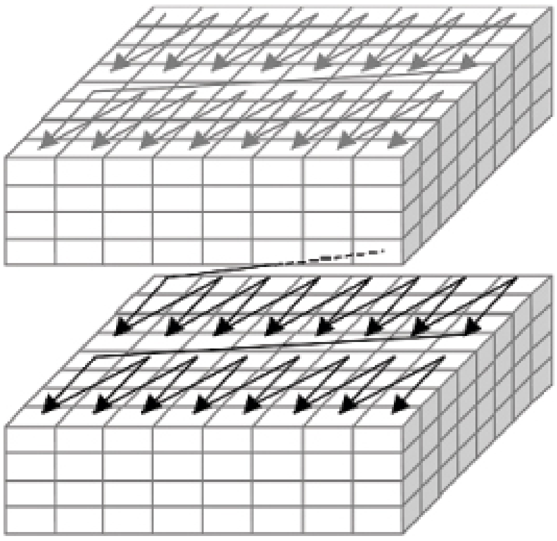
Figure 5: Bit-plane traversal of a code-block in Tier-1.

of structs that used self-referencing pointers to determine the next state according to the symbol encountered. The array was replaced 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 linked list that used self-referencing structs and dynamic memory allocation was replaced by a fixed size array and a current index that keeps 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. The function pointer was replaced by using a type argument and calling the appropriate function from a `switch` statement on this type.

Next, the optimization steps along the three axes are discussed.

4.6.1 Data placement

- The code-block coefficients are read from global memory and stored in a private array. As mentioned in Subsection 4.6 on GPUs these arrays are typically stored in the GPU's 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. The use of local memory, rather than private arrays, was considered but the large amount of data for one work item - 4 KB for the state variables and 16 KB for

Table 1: Optimal configuration for different images. The configuration is shown as the number of work groups times the work group size.

Image	**AMD**	**NVIDIA Tesla**	**NVIDIA GeForce**
	28 compute units	14 compute units	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 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, when compressing data it is difficult to predict how much memory is needed for the output. Therefore as much memory is allocated for the output as is needed to store the DWT-coefficients that are encoded. In this configuration the encoded bits of a code-block are stored in a chunk of memory that is as large as the one in which the code-block itself is stored.

4.6.2 Work space configuration

Because on GPUs whole work groups are assigned to compute units, depending on the image and code-block sizes, the work group size employed to execute Tier-1 will have a great impact on the performance. Especially on GPUs with many compute units the work group size must be chosen such that every compute unit gets to do more or less the same amount of work. Table 1 shows the optimal configuration for the tested GPUs for the two-dimensional test images (their details can be found in Table 2) using 64×64 code-blocks. For this work the optimal size was determined empirically and passed as a parameter to the program. Nevertheless, it is possible to automatically determine this size given that it solely depends on the number of code-blocks that have to be encoded and the device therefor used.

4.6.3 Architecture Specific Optimizations

No architecture specific optimizations were applied to the Tier-1 code. However, due to OpenCL restrictions all memory needed by the algorithm is allocated upfront rather than throughout the process using dynamic memory allocation. As Section 6 will show, the removal of expensive `malloc` calls, resulted for some images in a super-linear speedup on the multi-core CPUs compared with the sequential version run on the same CPU.

5 Results

The codec was tested on two-dimensional color images retrieved from `http://www.imagecompression.info/test_images` and three-dimensional gray-scale

Table 2: 2D and 3D test image characteristics.

Image Name	Dimensions	Components	Voxels [MVoxels]
flower_foveon	$2268 \times 1512 \times 1$	3	9.8
artificial	$3072 \times 2048 \times 1$	3	18.0
bridge	$2749 \times 4049 \times 1$	3	31.8
big_tree	$6088 \times 4550 \times 1$	3	79.3
mri_ventricles	$256 \times 256 \times 124$	1	7.8
mrt8_angio2	$256 \times 320 \times 128$	1	10.0
mrt8_angio	$412 \times 512 \times 112$	1	22.5
stent8	$512 \times 512 \times 174$	1	43.5
backpack8	$512 \times 512 \times 373$	1	93.3
vertebra8	$512 \times 512 \times 512$	1	128.0

Table 3: Hardware used for the tests: GPUs

Architecture	Fermi	Kepler	GCN
Vendor	NVIDIA	NVIDIA	AMD
Name	Tesla C2050	GeForce GTX 650 Ti	Radeon HD 7950
# compute units	14	4	28
# compute elements	448	768	1792
Clock frequency (MHz)	1150	928	800
Memory Bandwidth (GB/s)	126	86	240
Device OpenCL Version	1.1	1.2	1.2

images from `http://www.volvis.org`. Table 2 shows the dimensions of the tested images, their number of components and the resulting voxel count.

The codec was run on a variety of GPUs and Intel multi-core CPUs. No problems occurred on NVIDIA GPUs of generation Fermi or later. It is also possible to run the codec on AMD GPUs of the GCN family. Running the codec on the Intel multi-core CPUs was a mixed success: the code worked correctly on a laptop's Intel Core i5-4200U processor[2]. On the more powerful, but less recent, Intel Core i7-3930K CPU encoding images functioned correctly, but decoding failed. The fact that these problems occur on older multi-cores suggest they are due to issues in Intel's early OpenCL support. An overview of the hardware used in our tests is given in tables 3 and 4. We also tried to run our codec on two processors of the Intel Xeon family, but failed.

The GPUs were used to test acceleration on GPU only: image processing, DWT and Tier-1 were all accelerated on the GPU. The multi-core CPUs were used in a hybrid configuration: the Intel i7 was used to accelerate Tier-1 together with the NVIDIA Quadro K620 to accelerate image processing and DWT. The Intel i5 was used in cooperation with its integrated graphics card the Intel HD Graphics 4400. Note that for the Intel i7 the AMD driver was used while for the Intel i5 and the Intel HD Graphics the Intel driver was used.

In the following subsections performance is defined as the throughput achieved and expressed in MVoxels/s. Given an image of dimension $w \times h \times d$ consisting

[2]Although for very large images we exceeded the maximum OpenCL buffer size device limit.

Table 4: Hardware used for the tests: CPUs

Architecture	**Sandy Bridge E**	**Haswell**
Vendor	Intel	Intel
Name	i7-3930K	i5-4200U
# compute units	6×2	2×2
# compute elements	6×2	2×2
Clock frequency (MHz)	3200	1600
Memory Bandwidth (GB/s)	51.2	25.6
Device OpenCL Version	1.2	1.2

of n components and a time t needed to process the image, the performance P is given by:

$$P = \frac{n \times w \times h \times d}{t} \tag{1}$$

5.1 Codec Performance

Tables 5 and 6 show the throughput of the different codec versions for lossless coding of two- and three-dimensional images. For two-dimensional images code-blocks of size 64×64 were used for Tier-1 (Section 4.6), while for three-dimensional images code-blocks of size $16 \times 16 \times 16$ were used. Although it is possible to use larger code-blocks for three-dimensional images, this was not done. Larger code-blocks decrease the available parallelism and increase the resource usage of an OpenCL work item. Furthermore, as shown in [Bruylants et al.(2015)], very large code-blocks can decrease the compression efficiency.

The overhead due to the setup of OpenCL and the compilation of the OpenCL code are not taken into account. On the AMD GPU it takes 3 seconds to compile OpenCL code, while on the multi-core it takes 0.4 seconds. NVIDIA cached the compiled kernels, however, we needed 0.3 seconds to set up OpenCL. The run time can be deduced from the reported performance using Equation (1) with the values reported in Table 2. For encoding, the sequential performance refers to the original codec run on the Intel i7. For decoding, the sequential performance was measured on the Intel i5.

Given a sufficiently powerful multi-core the best performance is obtained with a hybrid version. This is very clear for small images, but also for large images the version that uses the NVIDIA Tesla C2050 exclusively is only marginally better. This is even more noteworthy given that the hybrid version uses a low-end GPU.

These results are comparable with the ones reported in [Ciznicki et al.(2014)]. From the graph in this paper one can deduce that they obtain a performance of around 32 MVox/s for the same two-dimensional images. It should be noted, however, that they do not take into account reading the data from file and that they use another kind of component transformation in the image preprocessing step. Furthermore, they achieve these results on a very high-end GPU the NVIDIA GTX 580.

Table 5: Encoding performance in MVoxels/s for the original version run on the Intel i7, the GPU accelerated versions and the hybrid version run on the Intel i7 and the NVIDIA Quadro K620.

image	**seq**	**tesla**	**geforce**	**amd**	**hybrid**
flower_foveon	3.1	10.2	8.8	2.7	28.0
artificial	3.1	14.6	12.8	4.6	31.6
bridge	2.2	15.1	14.3	6.0	22.9
big_tree	2.0	24.0	18.0	8.4	23.8
mri_ventricles	2.0	12.1	11.8	2.5	17.6
mrt8_angio2	2.2	14.9	11.8	3.1	20.4
mrt8_angio	2.1	22.3	11.3	6.2	20.9
stent8	2.9	29.6	18.2	10.5	30.6
backpack8	3.1	32.1	21.1	15.0	32.6
vertebra8	3.8	45.9	26.9	23.7	38.8

Table 6: Decoding performance in MVoxels/s for the original version run on the Intel i5, the GPU accelerated versions and the hybrid version run on the Intel i5 and the Intel HD integrated graphics card.

image	**seq**	**tesla**	**geforce**	**amd**	**hybrid**
flower_foveon	2.4	10.5	9.2	7.1	8.5
artificial	2.4	14.6	13.8	11.1	9.5
bridge	1.6	14.7	14.4	12.2	6.8
big_tree	1.5	22.8	18.1	20.2	6.7
mri_ventricles	1.5	14.8	14.6	10.2	6.1
mrt8_angio2	1.7	20.0	17.5	12.9	6.8
mrt8_angio	1.6	32.6	22.9	20.9	6.6
stent8	2.3	40.7	29.9	32.3	9.9
backpack8	2.5	51.6	33.6	41.6	N/A
vertebra8	3.0	69.5	51.5	69.6	N/A

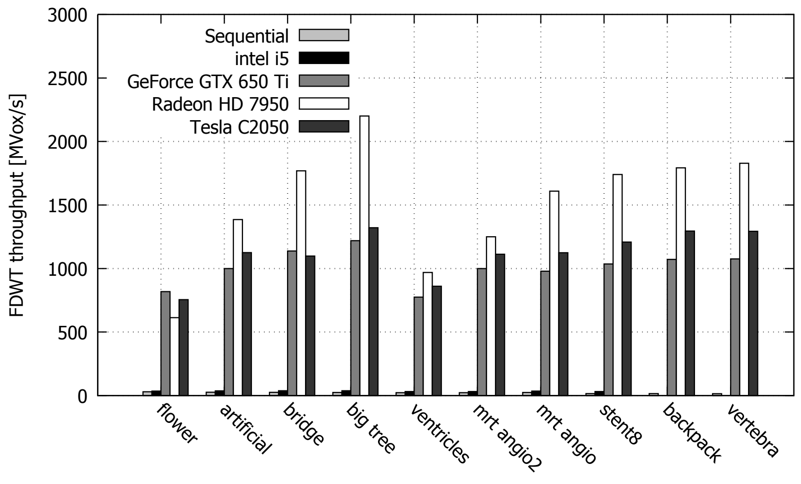

Figure 6: Forward DWT throughput on different devices for different images.

5.2 Discrete Wavelet Transform

Figure 6 shows the performance of the DWT implementation using the 5×3 wavelet when encoding the two- and three-dimensional test images. The reported performance concerns a multilevel DWT and includes the data transfers necessary on the GPU because the DWT cannot be performed in place.

5.3 Tier-1

Figure 7 shows the performance of the Tier-1 implementation when encoding the two- and three-dimensional test images. The best choice for our Tier-1 implementation is clearly the Intel multi-core. Only for very large two-dimensional and three-dimensional images the performance of the NVIDIA Tesla C2050 is similar to the one obtained on the multi-core.

The throughput obtained for decoding is shown in Figure 8. Because of the difficulties experienced on the Intel i7 multi-core when decoding, the Intel i5 multi-core is used for comparison with the GPUs. This multi-core obviously has a lot less processing power than the Intel i7 but is still able to obtain a respectable performance respective to the sequential performance. It is interesting to note that the AMD GPU does a lot better for decoding than for coding. This is caused by the fact that Tier-1 needs to do less work and requires less resources for decoding than for coding.

6 Performance Analysis

Figure 9 shows the run time distribution for lossless sequential compression of the largest three-dimensional image on the Intel i7 and for hybrid sequential compression on the Intel i7 and the NVIDIA Quadro K620. Clearly, Tier-1 remains the largest bottleneck, yet the contribution of DWT to the run time decreases dramatically while the relative contribution of other parts - image

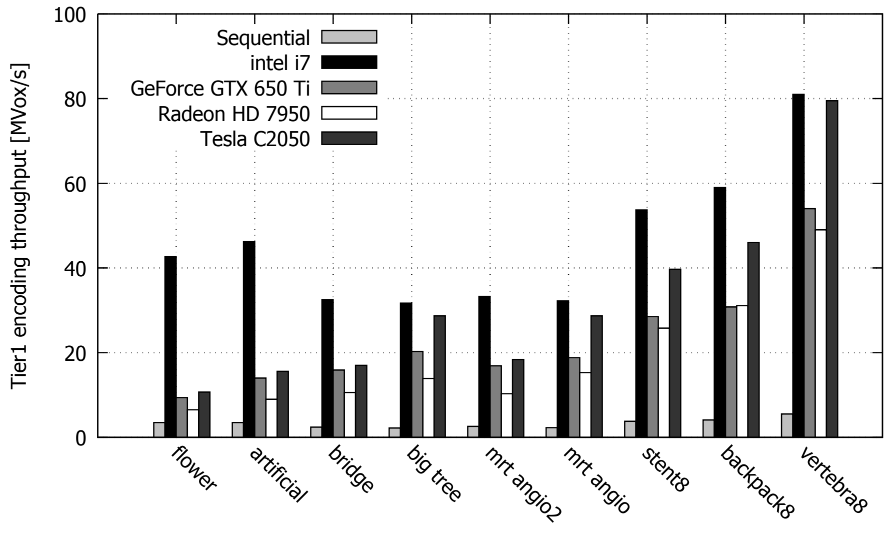

Figure 7: Tier-1 encoding throughput on different devices for different images.

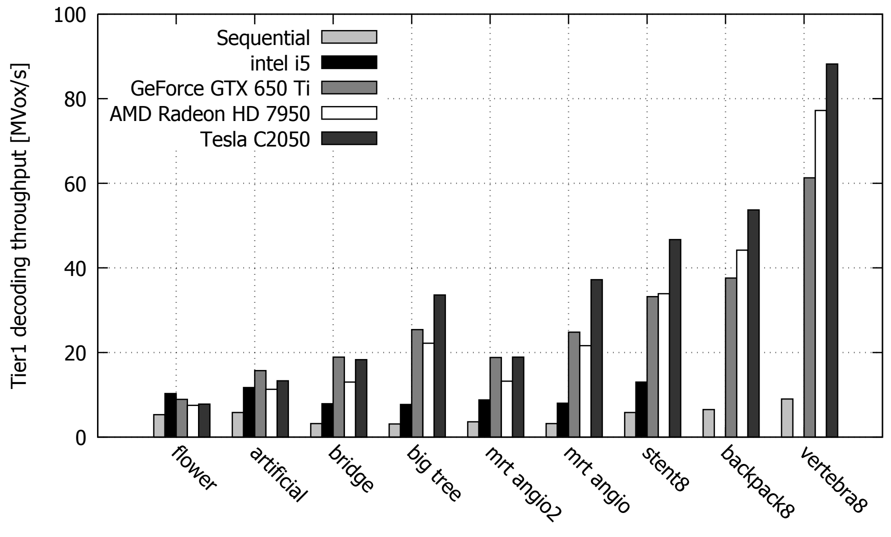

Figure 8: Tier-1 decoding throughput on different devices for different images.

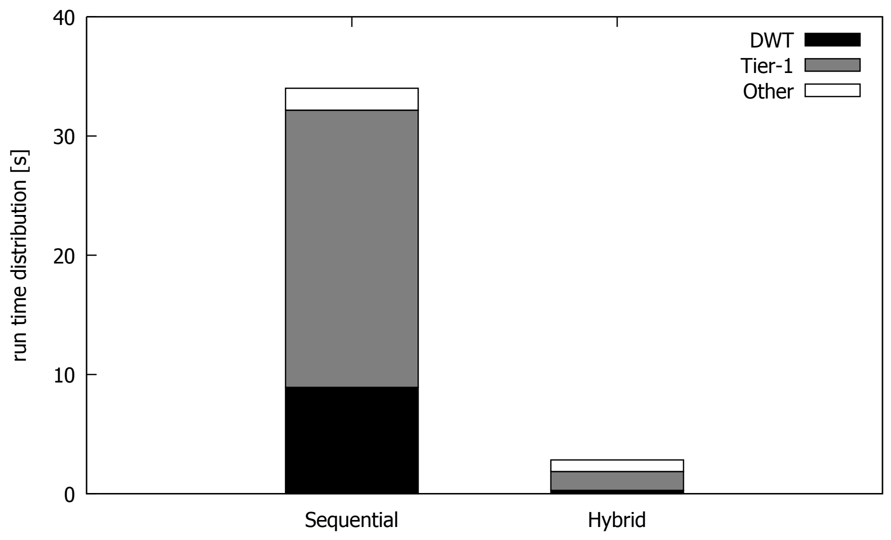

Figure 9: Run time distribution for sequential and hybrid compression of the vertebra8 image on the Intel i7 multicore and the NVIDIA Quadro K620 GPU.

preprocessing and Tier-2 - go up. In the following subsections we will discuss the performance characteristics of DWT and Tier-1 in more detail.

6.1 Discrete Wavelet Transform

Because of the fine-grained parallel implementation of DWT, its execution on GPU is expected to attain the peak performance. The compute intensity of the algorithm is very low: for the DWT that uses the 5×3 wavelet, only 3 useful operations are performed for every 8 bytes that are accessed in global memory. The algorithm is clearly memory bound wherefore its data throughput should be compared with the theoretical memory bandwidth of the GPU on which it is run. Figure 10 shows the performance in voxels per second of a single execution of the X-DWT, Y-DWT and Z-DWT kernels on an image of size $512 \times 512 \times 512$. The throughput is computed by dividing the number of voxels by the execution time of the kernel executing the DWT. On all GPUs the Y-DWT kernel performs most efficiently: the AMD performs best, attaining a data throughput of 172 GB/s or 72% of its bandwidth. The Tesla attains 44% of its bandwidth (55 GB/s) and the GeForce 51% (44 GB/s). Note that these figures take into account the overlap between the image partitions: for each voxel 8.6 bytes are accessed.

The difference in performance depending on the direction of the DWT can be easily understood. Because on GPUs the memory requests of small groups of threads that are executed together, the ideal memory request of such a group concerns data that can be got from memory in as few transactions as possible. This is the case for Y-DWT in which 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 partitions overlap along the X-axis. Therefore the adjacent voxels may belong to different memory segments,

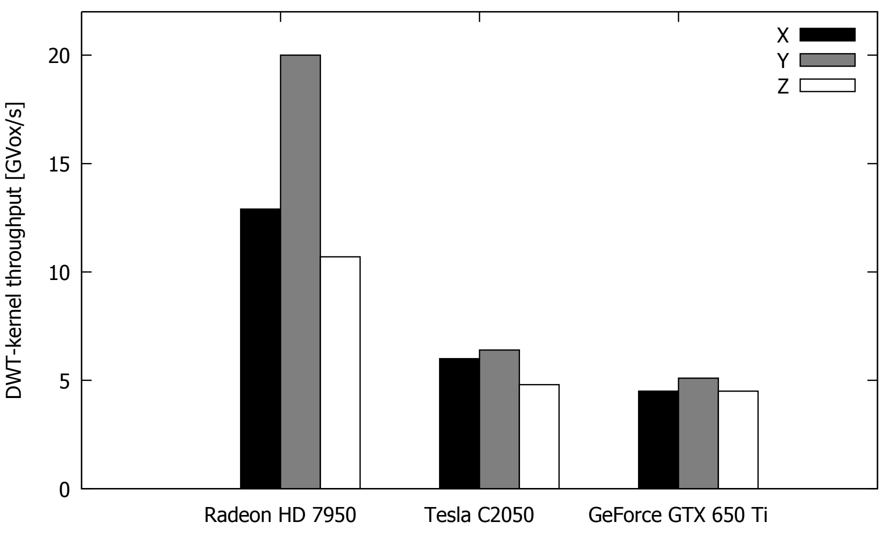

Figure 10: Performance in GVox/s of the X-, Y- and Z-DWT kernels on a $512 \times 512 \times 512$ image on the GPUs from table 3.

causing extra memory transactions. The Z-DWT performs worse on the AMD and the NVIDIA Fermi GPUs. In this case the data concerning a single request is located in memory areas that are very far apart[3].

The fine-grained implementation is not well suited for multi-core processors. On the Intel i5 multi-core we only obtain a modest performance gain. The limited performance of the multi-core itself is one part of the explanation, but the main cause lies in the fine-grained decomposition of the operation for GPUs. On multi-core processors running a great number of threads that perform small tasks will result in lots of overhead caused by the high thread switching cost ([Lee et al.(2015)]). Furthermore, we introduced excess computation in order to optimize the data access for typical GPU memory systems. It is possible to provide a version that is better suited for multi-core processors but given the ubiquity of OpenCL compliant graphical cards and their superiority for this kind of algorithm, this is unnecessary.

6.2 Tier-1

The implementation of Tier-1 is coarse-grained. The relatively small number of large tasks make it less suited for GPUs. The SIMT execution model followed by GPUs exacerbates the problem. On GPUs threads are organized in fixed size groups that execute in lockstep: groups of 32 threads called warps on NVIDIA GPUs and groups of 64 threads called wave fronts on AMD GPUs. Also the memory requests of such a group are bundled and handled as one. To obtain an optimal memory throughput it is necessary that the data needed by a single warp or wave front can be accessed in as few memory transactions as possible.

[3]Actually, this is the reason that work groups access image partitions that have one side mapped to the X-axis. If this were not done, the performance would plummet.

When threads of the same warp or wave front follow different execution paths, their execution must be serialized hence decreasing the performance.

On the GPU the Tier-1 implementation suffers from three main performance limiters:

1. A non-ideal access pattern: each thread is traversing its own code-block and the corresponding state data and writing to areas in memory that are very far apart.
2. Limited parallelism, especially for small images. The number of threads corresponds to the number of code-blocks that have to be encoded.
3. Branching: because all threads of a warp or wave front process different code-blocks, their execution paths will diverge.

To estimate the impact of these performance limiters a number of tests were run on an NVIDIA GeForce GT 640. This GPU is basically a scaled down version of the GeForce GTX 650, with twice as few cores and three times less memory bandwidth. The tests concerned two-dimensional images. To understand 1), the memory access of Tier-1 is mimicked without doing any computations. The following memory accesses are considered: the copy of the code-block to the allocated private array, the code-block bit-planes traversal (Figure 5), the access to the state array, and the irregular writing of code bytes to the output buffer. A 66×66 state array of bytes is needed during the process, which is stored in a private array as well. A code-block is typically traversed 20 to 30 times. The encoding of an image of 64×64 code-blocks takes on average 169 ms. The benchmark with 25 traversals takes 121 ms, which is about the same. This indicates that the program is memory-bound; additional computations do not increase the run time a lot. Given the run time and number of memory accesses, the attained bandwidth is only 12 GB/s and the average number of cycles per memory request per warp are 19.4 cycles. Bit-plane traversals takes most of the execution time. Elimination of the other memory accesses does not reduce the execution time significantly except for writing to the output buffer. This reduces the time to 95 ms.

The impact of 2) is determined using a series of images that consist of identical sub-images of 64×64 voxels (Figure 11), where each image in the series is twice as large as its predecessor. Encoding these images without performing a DWT ensures that all threads encode identical code-blocks and makes it possible to study the run time in function of the number of code-blocks that are encoded. The first image in the series consists of 64 code-blocks. Execution of Tier-1 for this image on the GeForce GT 640 corresponds to the execution of two warps of 32 threads, or, one warp per compute unit. Figure 13 shows the run time in milliseconds for the concerned GPU and for the Intel iCore7-3930K. From this image it is clear that the run time of Tier-1 on the multi-core increases more or less linearly with the number of code-blocks. This is not the case for the GPU. For small images there is insufficient parallelism and the compute unit occupancy is too low for best performance. The run time only doubles from 2048 to 4096 code-blocks. This suggests that up to 32 warps can be run at the same time on a single compute unit. Work groups of 64 work items (2 warps) were used. For work groups of 32 work items execution takes longer: because only 16 work groups can be run concurrently on a single compute unit, work

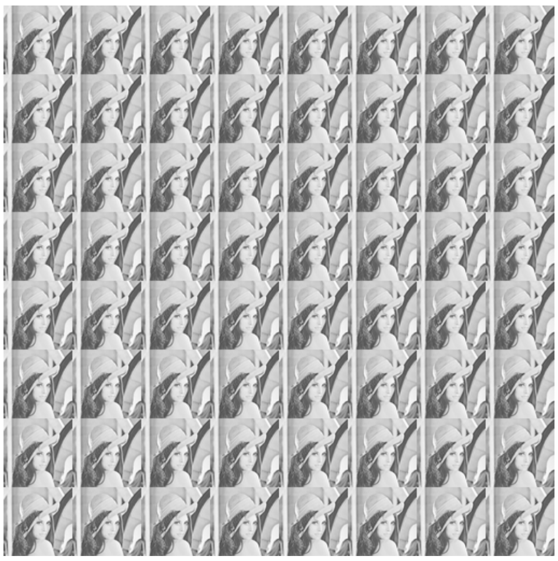

Figure 11: Tier-1 test image with identical sub-images.

groups consisting of 1 warp limit the concurrency to 16 warps per compute unit. For work groups consisting of more than 64 work items the run time does not decrease further. This experiment also explains why the use of smaller code-blocks increases the throughput of Tier-1 on the GPU for smaller images. If we use code-blocks of size 32×32, the number of code-blocks will be multiplied by 4 for an image of the same size, hence increasing the available parallelism by an equal factor.

The impact of 3) is estimated by a similar experiment that uses an image that is made up of four different 64×64 sub-images (Figure 12). The Tier-1 run time for these images is compared with the run times obtained for the four images of the same size that are each made up of one of the four sub-images. On the NVIDIA GeForce GT 6400, Tier-1 for the image with different sub-images was 64 % greater than the greatest run time of the four uniform images, while on the Intel iCore Tier-1 encoding of the composed image takes less than the maximal run time for the uniform image.

Finally, it is interesting to report the speedup of the OpenCL version run on the multi-core compared to the sequential version running on a single core of the same processor. Figure 14 shows speedups obtained for compressing images both on the Intel i5 and Intel i7 multi-core. It is interesting to note that despite the OpenCL overhead we manage to obtain a super linear speedup for both multi-cores: on the Intel i7 speedups greater than 12 are obtained, on the Intel i5 speedups greater than 4. The main reason here for can be found in the difference in memory management of both implementations: in the original version encoded bits are written to a buffer that is resized using `malloc` when it becomes too small, while in the OpenCL version fixed size memory to store the encoded bits is allocated before compression starts. Furthermore, instances of `malloc` pertaining to Tier-1 state management have also been replaced by static memory allocation.

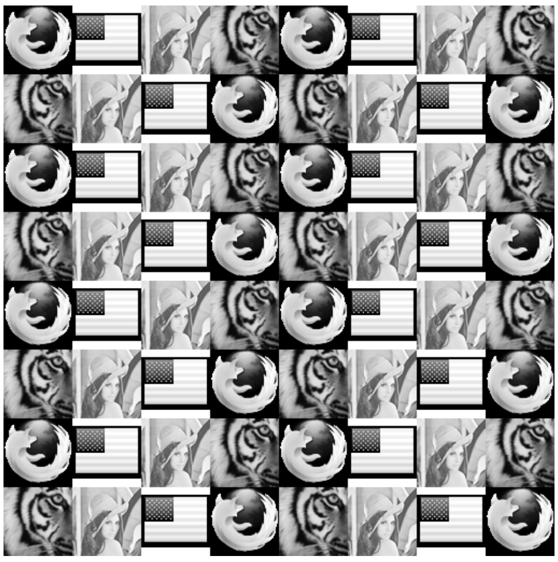

Figure 12: Tier-1 test image with different sub-images.

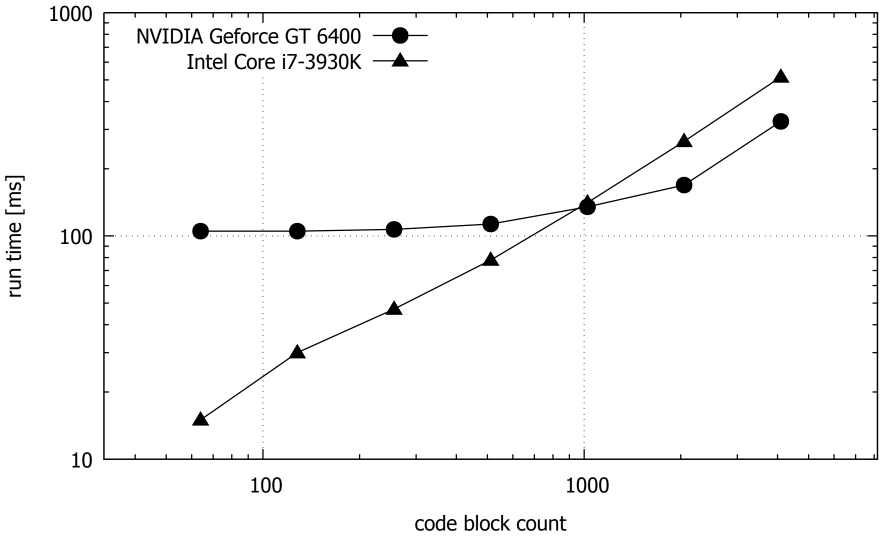

Figure 13: Tier-1 run time on GPU and multi-core in function of the available parallelism.

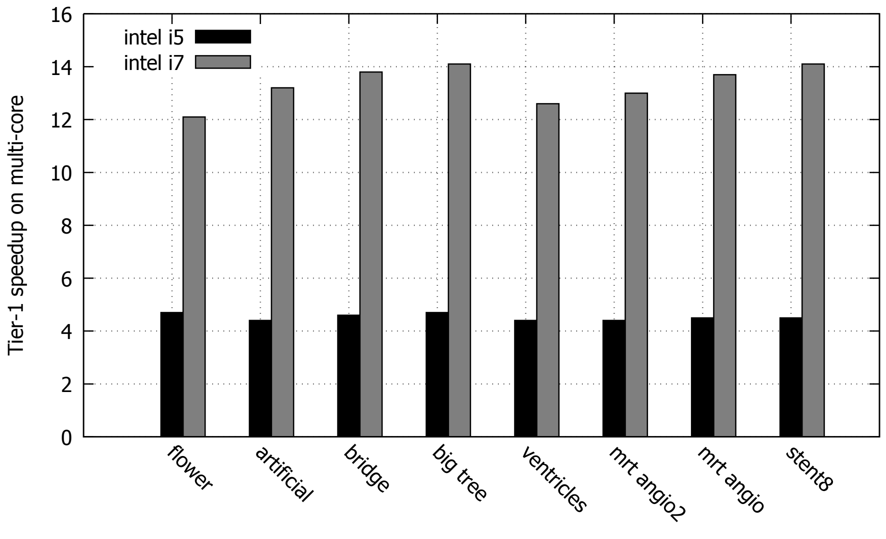

Figure 14: Tier-1 speedup on multi-core relative to the sequential version running on the same processor.

7 Conclusion

This paper discussed the acceleration of a sequential program with OpenCL to create a software that can use both GPUs and multi-core processors in a flexible manner. The approach followed was to target parts of the program to execute on the appropriate accelerator depending on the granularity of the parallel decomposition. Because we use OpenCL, however, it is possible to try alternative configurations depending on the platform on which the program is run.

We used a volumetric JPEG 2000 codec, IRIS-JP3D, as a real-world example of such an acceleration. The codec is characterized by two parts that consume most of its execution time: the DWT and the Tier-1 process. The first is a typical example of an algorithm that can be decomposed in a fine-grained manner and was targeted for acceleration on fine-grained GPUs. Tier-1 served as an example of an algorithm whose decomposition is rather coarse-grained and was targeted for acceleration on multi-core CPUs.

We discussed performance optimizations for code run on GPUs and provided a detailed discussion of the performance obtained for both parts across GPUs and multi-core CPUs. We obtained speedups up to a factor 12 combining the power of an Intel i7 multi-core with six hyper-threaded cores and a modest NVIDIA GPU of the latest generation. We were able to obtain a comparable performance for large images when solely using the NVIDIA Tesla C2050. A relatively modest Intel i5 multi-core combined with its integrated Intel HD Graphics GPU resulted in speedups up to 4.5 times.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: The research leading to these results has received funding from the European Research Council under the European Unions Seventh Framework Programme (FP7/2007-2013)/ERC Grant Agreement n.617779 (INTERFERE) and by the iMinds ICON MMIQQA project, Multimodel Microscopic Imaging: Quality, Quantification and Acceleration.

References

[Ahmadvand and Ezhdehakosh(2012)] Ahmadvand M and Ezhdehakosh A (2012) GPU-Based Implementation of JPEG 2000 Encoder. In: *The international conference on parallel and distributed processing techniques and applications (PDPTA)*, Las Vegas, NV, USA, 16-19 July 2012, pp.682-688. Athens: CSREA Press.

[AMD(2012)] AMD Radeon Graphics Technology(2012) AMD Graphics Cores Next (GCN) Architecture White Paper. Available at: `http://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf`. (Accessed on 26 April 2016).

[Balevic et al.(2009)] Balevic A, Fuerst N, Heide M, Papandreou S and Weiss A (2009) CUJ2K: JPEG 2000 Encoder in CUDA. Technical Report, Institute for Parallel and Distributed Systems, University of Stuttgart.

[Bruylants et al.(2007)] Bruylants T, Munteanu A, Alecu A, Deklerck R and Schelkens P (2007) Volumetric image compression with JPEG 2000. *SPIE Newsroom Biomedical Optics and Medical Imaging*. DOI:`http://dx.doi.org/10.1117/2.1200706.0779` (Accessed 5 August 2016).

[Bruylants et al.(2015)] Bruylants T, Munteanu A, and Schelkens P (2015) Wavelet based volumetric medical image compression. *Signal Processing: Image Communication* 31:112-133.

[Ciznicki et al.(2011)] Ciznicki M, Kurowski K and Plaza A (2011) GPU implementation of JPEG 2000 for hyperspectral image compression. In: *SPIE remote sensing*, Prague, Czech Republic, 19-22 September 2011, pp.81830H-81830H. Cardiff: SPIE.

[Ciznicki et al.(2014)] Ciznicki M, Kierzynka M, Kopta P, Kurowski K and Gepner P (2014) Benchmarking JPEG 2000 implementations on modern CPU and GPU architectures. *Journal of Computational Science* 5(2): 90-98.

[Franco et al.(2010)] Franco J, Bernabé G, Fernández J and Ujaldón M (2010) Parallel 3D fast wavelet transform on manycore GPUs and multicore CPUs. *Procedia Computer Science* 1(1): 1101-1110.

[Galeano et al.(2012)] Galiano V, López O, Malumbres M P and Migallón H (2012) GPU-based 3D Wavelet Transform. *Proceedings of the 12th international conference on computational and mathematical methods in sci-*

ence and engineering (CMMSE), La Manga, Spain, 2-7 July 2012, pp.580-590. Available at: `http://cmmse.usal.es/cmmse2016/sites/default/files/volumes/2-cmmse-2012.pdf` (Accessed 5 August 2016).

[Grama et al.(2002)] Grama A, Gupta A, Karypis G and Kumar V *Introduction to Parallel Computing.* 2nd ed.Harlow: Pearson Education.

[Khronos(2012)] Khronos (2012) OpenCL 1.2. Reference Pages. Available at: *https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/.* (Accessed 26 April 2016).

[Le et al.(2011)] Le R, Bahar I R and Mundy J L (2011) A novel parallel Tier-1 coder for JPEG 2000 using GPUs. *IEEE 9th symposium on application specific processors (SASP)*, San Diego, CA, USA, 5-6 June 2011, pp.129-136. Piscataway: IEEE.

[Lee et al.(2015)] Lee J H, Nigania N, Kim H, Patel K and Kim H (2015) OpenCL Performance Evaluation on Modern Multicore CPUs. *Scientific Programming, vol. 2015, Article ID 859491, 20 pages, 2015.* DOI:`http://dx.doi.org/10.1155/2015/859491`. (Accessed 5 August 2016).

[Matela(2009)] Matela J (2009) GPU-Based DWT Acceleration for JPEG 2000. *Annual doctoral workshop on mathematical and engineering methods in computer science (MEMCS)*, Znojmo, Czech Republic, 13-15 November 2009, pp.136-143. Brno, Czech Republic: Novpress S.r.o.

[Matela, Rusnak and Holub(2011)] Matela J, Rusnak V and Holub P (2011) Efficient JPEG 2000 EBCOT context modeling for massively parallel architectures. *Data compression conference (DCC)*, Snowbird, UT, USA, 29-31 March 2011, pp.423-432. Piscataway: IEEE.

[Matela, Šrom and Holub(2011)] Matela J, Šrom V and Holub P (2011) Low GPU occupancy approach to fast arithmetic coding in JPEG 2000. *Mathematical and engineering methods in computer science (MEMCS)*, Lednice, Czech Republic, 14-16 October 2011, pp.136-145. Berlin: Springer.

[Nickolls et al.(2008)] Nickolls J, Buck I, Garland M. and Skadron K (2008) Scalable parallel programming with CUDA. *Queue* 6(2): 40-53.

[NVIDIA(2009)] NVIDIA corporation (2009) NVIDIA. NVIDIA's Next-Generation CUDA Compute Architecture. Available at: `http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf` (Accessed 5 August 2016).

[Patterson and Hennessy(2012)] Patterson D and Hennessy J (2012) *Computer Organization and Design: The Hardware/Software Interface.* 4th ed. Boston: Morgan Kaufmann.

[Seo et al.(2011)] Seo S, Jo G and Lee J (2011) Performance characterization of the NAS Parallel Benchmarks in OpenCL. *IEEE international symposium on workload characterization (IISWC)*, Austin, TX, USA, 6-8 November 2011, pp.137-148. Piscataway: IEEE

[Schelkens et al.(2009)] Schelkens P, Skodras A and Ebrahimi T (Eds.) (2009) *The JPEG 2000 Suite*. Chichester: John Wiley & Sons.

[Stone et al.(2010)] Stone J E, Gohara D and Shi G (2010) OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12(1-3): 66-73.

[Sweldens(1996)] Sweldens W (1996) The Lifting Scheme: A Custom-Design Construction of Biorthogonal Wavelets. *Applied and Computational Harmonic Analysis* 3(2): 186-200.

[Taubman and Marcellin(2002)] Taubman D and Marcellin M (2002) *JPEG2000 Image Compression Fundamentals, Standards and Practice*. Boston, MA: Kluwer Academic Publishers.

[Wei et al.(2012)] Wei Fi, Cui Q. and Li Y (2012) Fine-Granular Parallel EBCOT and Optimization with CUDA for Digital Cinema Image Compression. *IEEE international conference on Multimedia and expo (ICME)*, Melbourne, Australia, 9-13 July 2012, pp.1051-1054. Piscataway: IEEE.