J Real-Time Image Proc (2016) 12:219–235
DOI 10.1007/s11554-015-0507-5

SPECIAL ISSUE PAPER

OpenCL-based optimization methods for utilizing forward DCT and quantization of image compression on a heterogeneous platform

Nasser Alqudami · Shin-Dug Kim

Received: 12 January 2015 / Accepted: 6 May 2015 / Published online: 22 May 2015

Abstract Recent computer systems and handheld devices are equipped with high computing capability, such as general purpose GPUs (GPGPU) and multi-core CPUs. Utilizing such resources for computation has become a general trend, making their availability an important issue for the real-time aspect. Discrete cosine transform (DCT) and quantization are two major operations in image compression standards that require complex computations. In this paper, we develop an efficient parallel implementation of the forward DCT and quantization algorithms for JPEG image compression using Open Computing Language (OpenCL). This OpenCL-based parallel implementation utilizes a multi-core CPU and a GPGPU to perform DCT and quantization computations. We demonstrate the capability of this design via two proposed working scenarios. The proposed approach also applies certain optimization techniques to improve the kernel execution time and data movements. We developed an optimal OpenCL kernel for a particular device using device-based optimization factors, such as thread granularity, work-items mapping, workload allocation, and vector-based memory access. We evaluated the performance in a heterogeneous environment, finding that the proposed parallel implementation was able to speed up the execution time of the DCT and quantization by factors of 7.97 and 8.65, respectively, obtained from 1024×1024 and 2084×2048 image sizes in 4:4:4 format.

Keywords Forward DCT · Quantization · Image compression · Parallel image processing · Heterogeneous computing · OpenCL

1 Introduction

General-purpose computing on GPUs [1, 2] became very popular, especially after the appearance of CUDA and Open Computing Language (OpenCL) technologies supported by some hardware vendors. CUDA was the first technology to allow a CPU-based program to run on the GPU, but it works only on NVIDIA hardware platforms. Therefore, OpenCL [3, 4] was introduced to overcome such limitations. Currently, many vendors have released their own framework and OpenCL implementations, such as AMD, NIVIDA, and Intel. OpenCL is an open-standard parallel programming language for heterogeneous architectures that supports code portability. Therefore, an application written in OpenCL can run on different architectures without code modification [3]. The availability of multi-core CPUs and many-core graphics processing units (GPUs) enables programmers to solve computational problems efficiently in heterogeneous environments.

In recent computer systems, heterogeneous architecture exists not only in clusters and desktop and notebook computers, but also goes beyond that to portable and handheld devices such as mobile phones and tablets. The new generation of smartphones, such as the Samsung Galaxy S5 LTE version [5], is powered by a 2.5 GHz quad-core CPU, Snapdragon 801 system-on-chip with Adreno 330 GPU. Low-power handheld devices will benefit from gradual increases in multi- and many-core hardware, replacing the specialized embedded hardware accelerators (such as DSP) with real-time software solutions. This can

N. Alqudami (✉) · S.-D. Kim
Department of Computer Science, College of Engineering, Yonsei University, 134 Sinchon-dong, Seodaemun-gu, Seoul 120-749, Republic of Korea
e-mail: nasser@yonsei.ac.kr

S.-D. Kim
e-mail: sdkim@yonsei.ac.kr

be accomplished by combining GPGPUs with traditional multi-core CPUs to achieve real-time processing for heavy applications such as image and video codecs. Once such a scenario is achieved, the embedded hardware coprocessors will disappear, which will lead to reductions in power consumption and hardware chip complexity.

Discrete cosine transform (DCT) [6] and quantization are important parts of many image and video compression standard methods, such as JPEG, MPEG, H.261, and HEVC [28–32]. DCT operation is one of the main stages in the JPEG codec and is known as a heavy and intensive task that requires the majority of the execution time [7–10]. For that reason, most applications are accelerated via an embedded hardware codec in a low-computation device, in order to enhance the user-response and provide real-time processing [9–12].

Despite that this work focuses on the JPEG image compression, the proposed methods and parallelization approach are very promising and can be very useful for any DCT-based image and video compression applications, such as JPEG, MPEG-1, MPEG-2, H.261, H.263, H.264, and most recently H.265/HEVC [28–32] as well as any other digital signal processing methods that use the same algorithms.

In this paper, we mainly focus on the forward DCT and quantization parts of a JPEG encoder in a heterogeneous-based platform. We spread the computations of the DCT and quantization across multi- and many-core CPUs and GPUs to accelerate the performance. Moreover, we develop an efficient OpenCL-based parallel implementation for the forward DCT and quantization algorithms through experiments using AMD OpenCL platform drivers for both the CPUs and GPUs.

Most of the developed parallel implementations were CPU-based or GPU-based only. Our idea is, instead of offloading all work to the GPU, the CPU can participate by doing a small portion of the work, which will keep both devices busy and greatly reduce the cost of data movements across the PCIe bus.

The objective of this work was to study the performance issues for image compression on a heterogeneous platform by distributing the workload among the CPUs and GPUs. Initially, we started with the forward DCT and quantization stages of the JPEG image compression, applied to a case study to determine the optimal performance of each device by applying different optimization methods. Later, this work will be extended to involve all the other stages of the JPEG encoder. The target of this work is the multi-core-based low-power handheld and portable devices, in which a real-time software image codec will replace the hardware one in the future. The remainder of the paper is organized as follows: Sect. 2 presents the related work, while Sect. 3 describes the background for the DCT and quantization algorithms. Section 4 describes the design and the proposed methods along with various optimization techniques. The experimental results and performance evaluation are presented in Sects. 5 and 6, respectively. Finally, the conclusion and future work are described in Sect. 7.

2 Related work

The discrete cosine transforms (DCT) and quantization are widely used in many images and video compression standards. The DCT is implemented in many ways, including matrix-multiplication-based and FFT-based approaches. DCT is an intensive task in digital signal processing and multimedia compressions which has been accelerated in hardware and software platforms. In terms of the hardware platforms, DCT and quantization are parts of the DSP processor for the image codec [9–12], and in the case of software platforms, a DCT algorithm has been targeted on the GPU via OpenGL [13], Cg [14], CUDA [15, 16], and OpenCL [17, 18].

The majority of accelerated DCT implementations are GPU-based, which is done by offloading the work to the GPU and leaving the CPU to do other tasks. However, our work is to utilize the CPU and GPU in the system to cooperate in handling the DCT and quantization computations. This is done by assigning each device part of the data to keep both devices busy in handling such computations, instead of relying on one computational device.

3 Background

3.1 JPEG image compression standard

The sequential JPEG method is a DCT-based compression algorithm. The source color image consists of three components: red, green, and blue. Each component of the digital image is represented as a rectangular array of samples, arranged in x columns and y rows. The JPEG encoder diagram contains forward discrete cosine transforms (DCT) and quantization as two separated consecutive steps. In the encoding process, the input components of a digital image are divided into blocks of size 8×8. By applying forward DCT to each block, 64 DCT coefficients can be obtained [19–21]. These 64 coefficients are quantized by applying the quantization step.

The JPEG baseline is mainly divided into five steps, as shown in Fig. 1: color conversion and optional down-sampling, forward DCT, quantization, and entropy coding (Huffman or Arithmetic).

In the DCT sequential-mode encoder, the JPEG diagram shows how single-component compression works in a fair

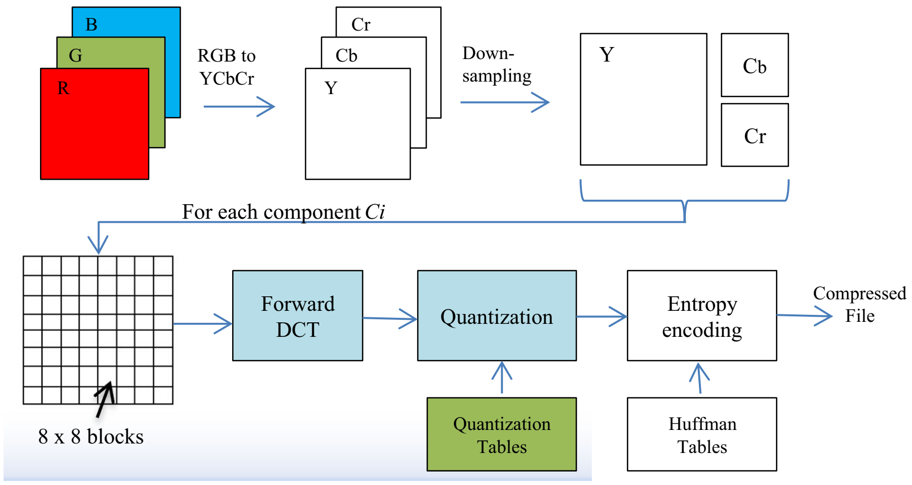


Fig. 1 JPEG encoder block diagram

way. The 8 × 8 block is input, makes its way through each processing stage, and then yields the output in a compressed data stream [22].

3.2 Discrete cosine transform (DCT)

The JPEG baseline compression algorithm is based on the DCT of non-overlapping, 8 × 8 blocks of samples of a component of a digital image. DCT transforms an image in the spatial domain into another in the frequency domain. Each pixel in the original image is assumed to represent a value between 0 and 255. Before DCT, a level shift is done by subtracting 128 from each pixel, making the pixel value range from −128 to 127. Then, each image component is partitioned into 8 × 8 blocks, and these blocks are processed one by one from left to right and top to bottom [22].

The FDCT of an 8 × 8 block of samples, $f(x, y)$: (y, x = 0, 1,…, 7), is defined as follows:

$$F(u,v) = \frac{C_u}{2}\frac{C_v}{2}\sum_{y=0}^{7}\sum_{x=0}^{7} f(x,y) \times \cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right] \quad (1)$$

$$C_u C_v = \begin{cases} 1/\sqrt{2} & \text{for } u,v=0 \\ 1, & \text{otherwise} \end{cases},$$

where $f(x, y)$ is the pixel at position (x, y) in the block, $F(u, v)$ is the transformed (u, v) DCT coefficient.

The complexity of the direct 2-DCT algorithm is $O(n) = N^4$. It is time quadratic and intensive work to compute a single DCT block. There is an alternative way to compute the DCT algorithm as a one-dimensional row-column pass instead of two dimensions to reduce the complexity of the algorithm $O(n) = N^2 + N^2 = 2\,N^2$. The first 1D DCT is applied to the rows of the block, and the second one is applied to the columns of the result obtained after calculating the first DCT. The 2D DCT can be reduced to two 1D ones and defined by formulas (2) and (3):

$$F(y,u) = \frac{C_u}{2}\sum_{x=0}^{7} f(y,x)\cos\left[\frac{(2x+1)u\pi}{16}\right] \quad \text{1D row pass} \quad (2)$$

$$F(u,v) = \frac{C_v}{2}\sum_{y=0}^{7} f(y,u)\cos\left[\frac{(2y+1)v\pi}{16}\right] \quad (3)$$

1D column pass.

The DCT-based JPEG image compression is based on the AAN algorithm [23], which is one of the fastest DCT algorithms. In this paper, the floating DCT implementation was used, as it produces the highest quality images, even though it is slower than the other DCT implementations. However, modern computers are highly optimized to perform the floating point operations much faster than before, especially on the discrete GPU. In order to speed up the 2D DCT computations, the AAN algorithm is used to calculate the forward DCT and applied on each 8 × 8 block.

3.3 Quantization

Computation of the DCT of each block is followed by quantization of the DCT coefficients. The quantization process makes the JPEG baseline a lossy algorithm [19]. Each 8 × 8 block of DCT coefficients is quantized using uniform quantizers whose step sizes are determined based on the human visual perception model.

Table 1 is the default Luma quantization matrix. The corresponding Chroma (Cb and Cr) quantization matrix is listed in Table 2. The compression ratio can be controlled by multiplying the quantization matrices by a quantizer scale.

The JPEG rule for quantizing the DCT coefficients is described by the following equation:

Table 1 JPEG default quantization table for luminance

Luminance quantization table							
16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Table 2 JPEG default quantization table for chrominance

Chrominance quantization table							
17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

$$Yq(u,v) = \left\lfloor \frac{Y(u,v)}{Q(u,v)} + 0.5 \right\rfloor, \quad 0 \le u,v \ll 7, \tag{4}$$

where $Y(u, v)$ is the unquantized DCT coefficient, $Yq(u, v)$ is the corresponding uniformly quantized coefficient, $Q(u, v)$ is the quantization step size for the (u, v)th coefficient, and x is the operation of rounding to the nearest integer [21].

These are the quantization table coefficients that are recommended in the Annex of the JPEG standard. We use the default quantization table with a scaling factor of 75 as the default parameters in the JPEG standard compression.

4 Design and methods

4.1 Proposed OpenCL parallel diagram

In this section, we describe the OpenCL parallel design for forward DCT and quantization of a JPEG encoder in a heterogeneous platform, where the computational devices differ in their performance capabilities and characteristics. In our system design, we targeted the OpenCL parallel implementation on both multi-core CPU and GPU devices because they are widely used at present in personal computers, notebooks, and handheld devices.

Two working scenarios (WS1 and WS2) have been considered with our design to perform the DCT and quantization computations, (1) the naïve way and (2) the combined way, as shown in Fig. 2. In the first scenario, DCT and quantization are processed separately in two consecutive steps. In the second scenario, DCT and quantization are merged and processed in a single step. Further details about these scenarios are provided later.

In the proposed design, the computation is performed in parallel on CPU and GPU computing devices having different workload sizes. The input data are a digital image with three color components along with luminance (Y) and chrominance (C_bC_r) components called YC_bC_r. We assume that the color space conversion and down-sampling have taken place in an early stage of the encoder. The image components are partitioned between the devices: the Y component is given to the CPU, and the C_bC_r components are given to the GPU. Moving the data from/to the targeted device is handled using two techniques: (1) memory copy operations (write-read) and (2) memory-mapping operations (zero-copy). OpenCL kernels represent the actual code for the DCT and quantization procedures that run on a particular device. The workflow can be summarized as follows: A host thread dispatches Y→ CPU and C_b→ GPU at the same time, and each device starts to execute the OpenCL kernel(s) to perform computations on the input data. As soon as the GPU finishes the first task, it immediately starts to process the C_r component as the kernel execution on C_b overlaps with C_r data transfer. The data transfer and kernel execution are coordinated using events to monitor the completion of each task before a new one starts.

4.2 Overlapping OpenCL Kernel execution with data transfer

In OpenCL, kernel execution and data transfer operations run sequentially (write, execution, and read), as shown in Fig. 3a. First, the program copies the data from host to device (write: H→D). Once data transfer completes, the kernel execution is performed, and then the output data must be copied back from device to host (read: D→H). Applying the sequential mechanisms on the GPU to process the C_bC_r components, the six operations must run in sequence, creating a large amount of idle time for the GPU because it must wait for the C_b read and write operations to be completed before beginning to process the C_r component. Therefore, to reduce the idle time for the GPU, we overlap the data transfer with the kernel execution by using non-blocking call operations (asynchronous data transfer). Figure 3b shows the overlap approach for kernel execution and data transfer operations on the GPU. Since an OpenCL

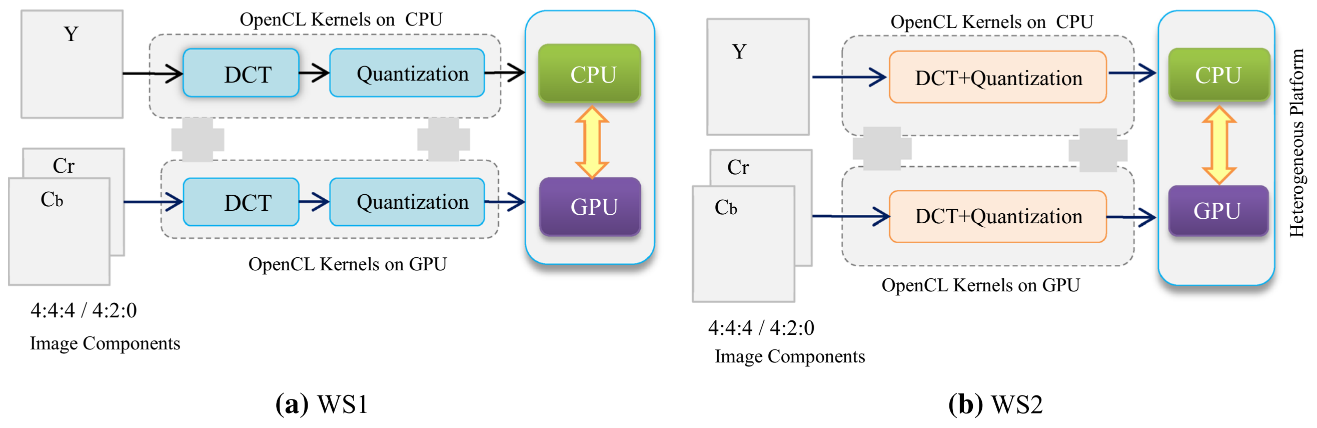


Fig. 2 Proposed structure design of OpenCL parallel implementation for DCT and quantization of JPEG encoder, using two scenarios **a** naïve way and **b** combined

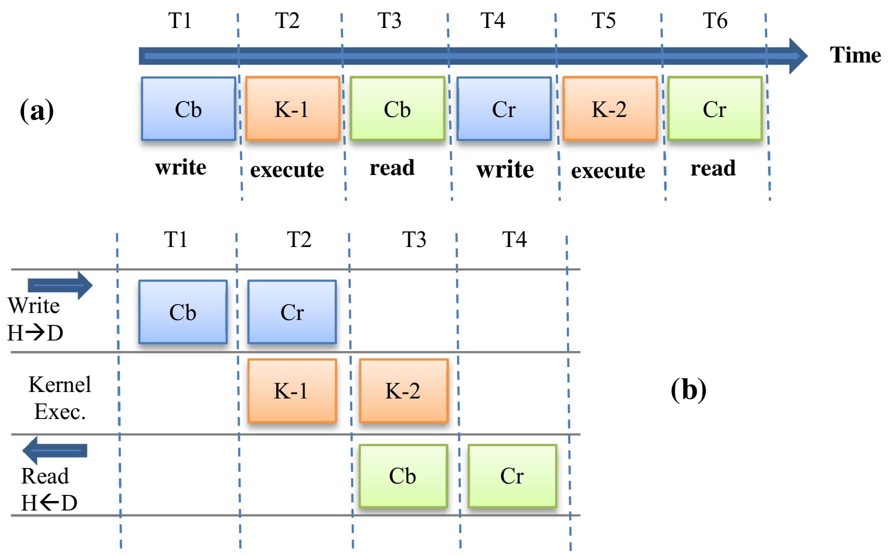


Fig. 3 OpenCL data transfer and kernel execution on the GPU; **a** sequential and **b** overlapped

device can run only one kernel at a time, the C_bC_r components will be processed one by one on the GPU side.

The operations in the overlap approach are executed in the following order:

T1 Execute write-1 and write C_b data from host to device corresponding to the K-1 kernel
T2 Start K-1 kernel execution, which overlaps with operation write-2 for the C_r (e.g., while GPU works on C_b, the C_r write operation can be initialized)
T3 The k-2 kernel execution overlaps the read-1 operation for C_b
T4 Execute the read-2 operation for C_r

Write-1 and write-2 represent the data transfers from CPU memory to GPU memory (H→D), while read-1and read-2 represent the data transfers from GPU memory to CPU memory (D→H) for the C_bC_r components. K-1 and k-2 represent the OpenCL DCT and quantization kernels execution.

4.3 Leverage multiple devices in OpenCL

To leverage multiple devices in OpenCL to work in parallel, we create a shared context for all computing devices and one command queue per device. A separate thread for each command queue is used in the AMD OpenCL implementation to send tasks to the command queue associated with the target device. Therefore, each queue receives and processes computations in parallel with other queues, effectively achieving parallel execution. Figure 4 shows the thread control flow in the OpenCL host program and the commands execution order. Initially, the host thread proceeds until the OpenCL run-time creates a thread for the CPU and another thread for the GPU. The two threads run

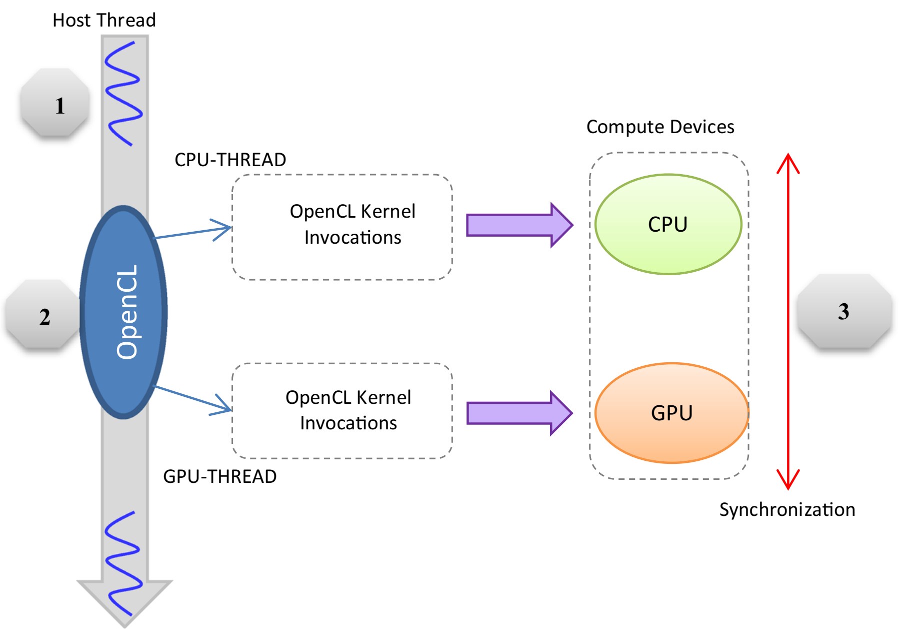


Fig. 4 Thread control flow in the OpenCL host program

in parallel, each executing the kernels and memory operations in the command queue on its device. There is one synchronization point at the application level; here, a thread execution can stop to wait for the other thread device to join it, and then the host thread can continue to terminate the program.

4.4 Working scenarios employing the DCT and quantization kernels

In accordance with our previous design, we highlight two working scenarios in which we perform the DCT and quantization computations using either the naïve way or the combined way. We implement both scenarios using OpenCL to study their impacts on the execution time. In the naïve way, the DCT and quantization computations are handled in two consecutive steps. First, the DCT kernel is invoked, followed by quantization kernel invocation, and a barrier ensures that the second step cannot start execution until the first step finishes. In contrast, in the combined scenario, the DCT and quantization computations are handled in a single step; therefore, only a single kernel invocation is needed, followed by a barrier. The OpenCL API, which used for both scenarios, summarized in Table 3.

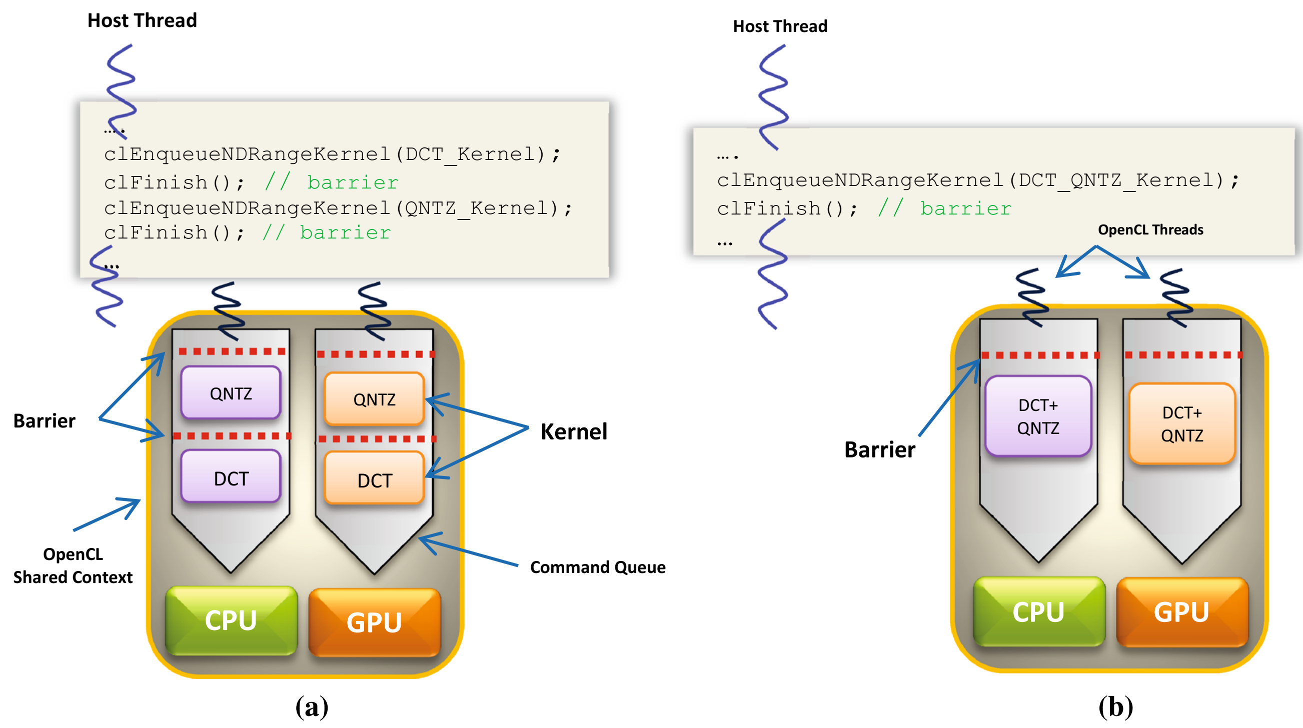
Figure 5 shows the differences between the two working scenarios for the DCT and quantization computations in our OpenCL parallel implementation. The main advantages of the combined approach are (1) reducing the API calls by half and (2) increasing the data locality and reducing the memory access time overhead to the off-chip memory. With only one fetch, the DCT and quantization computations are performed on the 8 × 8 data block, then writes the results back to the global memory; this replaces the two fetches and two writes used in the naïve approach.

4.5 OpenCL work-item mapping

In this section, we introduce two types of OpenCL work-items mapping. The mapping defines the work size given to the OpenCL work-item. In our parallel implementation, the work size is the relationship between the 8 × 8 blocks and the number of OpenCL work-items because the basic unit of processing in DCT and quantization is an 8 × 8 block. (In other words, how many work-items will be used to process a single 8 × 8 block) The mapping types are (1) one work-

Table 3 OpenCL API calls for naive and combined approaches to working scenarios

Naïve way (WS1)	Combined way (WS2)
clCreateKernel(DCT_KERNEL);	clCreateKernel(DCT_QNTZ_KERNEL);
clCreateKernel(QNTZ_KERNEL);	clEnqueueNDRangeKernel(DCT_QNTZ_KERNEL);
clEnqueueNDRangeKernel(DCT_KERNEL);	clFinish(DCT_QNTZ_KERNEL);//barrier
clFinish(DCT_KERNEL);//barrier	
clEnqueueNDRangeKernel(QNTZ_KERNEL);	
clFinish(QNTZ_KERNEL);//barrier	


```
….
clEnqueueNDRangeKernel(DCT_Kernel);
clFinish(); // barrier
clEnqueueNDRangeKernel(QNTZ_Kernel);
clFinish(); // barrier
…
```

```
….
clEnqueueNDRangeKernel(DCT_QNTZ_Kernel);
clFinish(); // barrier
…
```

Fig. 5 DCT and quantization working scenarios; **a** naïve and **b** combined

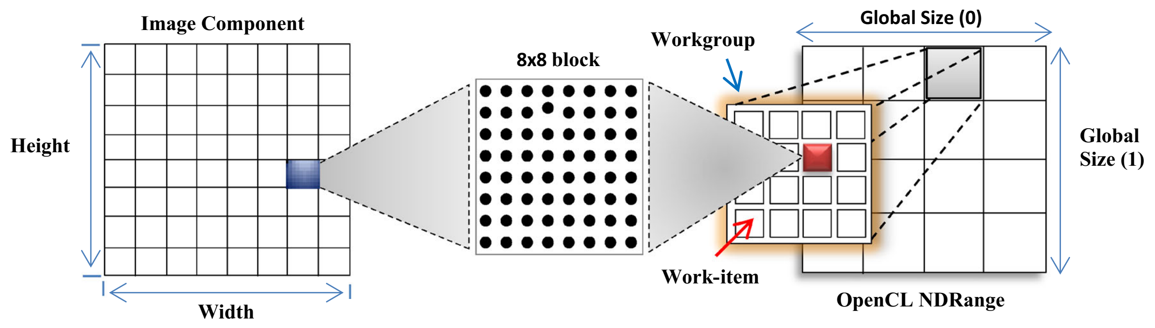


Fig. 6 OpenCL work-item mapping-1 (one work-item processes in one 8 × 8 block)

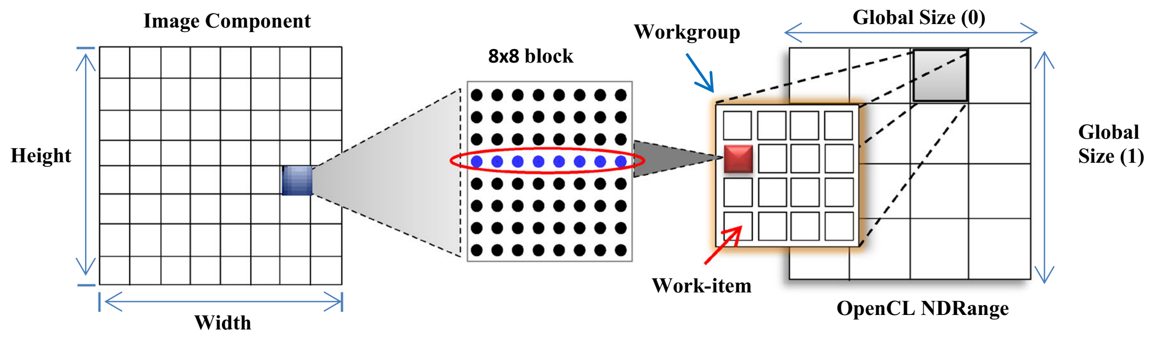


Fig. 7 OpenCL work-item mapping-2 (eight work-item processes in one 8 × 8 block)

Table 4 Summary of OpenCL work-items mapping for each working scenario

Work-item mapping for working scenario-1 (two steps)			Work-item mapping for working scenario-2 (one step)		
Kernel	CPU	GPU	Kernel	CPU	GPU
DCT	1:1 (mapping-1)	8:1 (mapping-2)	DCT_QNTZ	1:1 (mapping-1)	8:1 (mapping-2)
QNTZ	1:1 (mapping-1)	8:1 (mapping-2)			

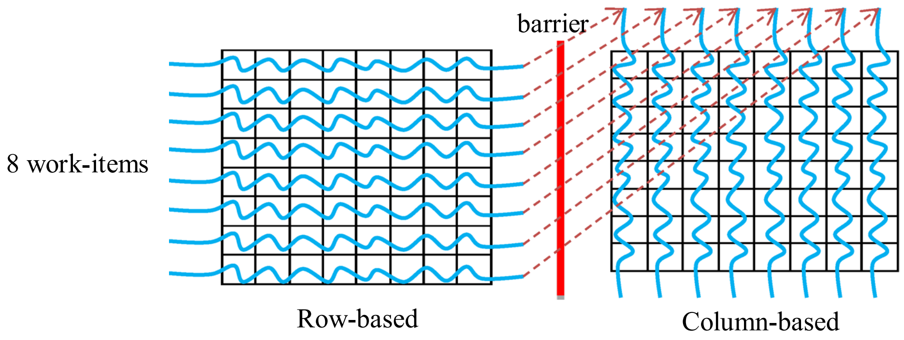


Fig. 8 OpenCL DCT block processing on the GPU via mapping-2

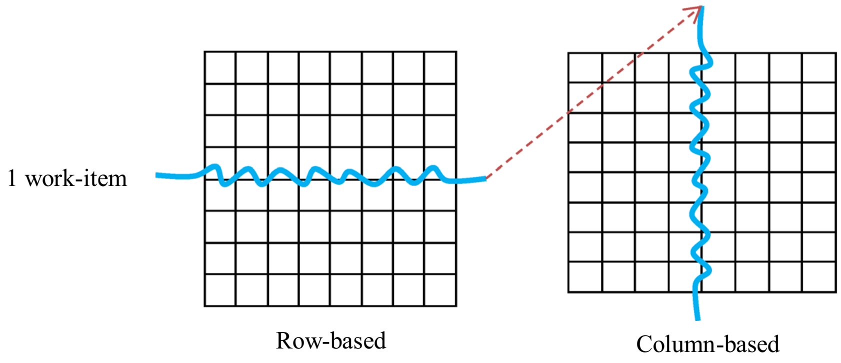


Fig. 9 OpenCL DCT block processing on the CPU via mapping-1

item per 8 × 8 block, as shown in Fig. 6, and (2) eight work-items per 8 × 8 block, as shown in Fig. 7. Note, for example in the second mapping, one work-item processes one row of an 8 × 8 block, and eight work-items process an entire 8 × 8 block. We examined two types of mapping, because the CPU and GPU have different numbers of hardware computing units that run work-items. Therefore, we can adjust the granularity as is suitable on each device. For example, the CPU has only a few cores, which represent the computing units. Each computing unit on the CPU has only one processing element, which runs only one work-item at a time; therefore, a coarse-grain process is highly preferred. In contrast to the GPU, which has many computing units, each computing unit has many processing elements that run 16–32 work-items at the same time (e.g., the number of work-items running in parallel is a hardware-dependent factor); therefore, a lightweight process is highly preferred in the GPU. Table 4 summarizes the work-items mapping policy we applied on each device for each working scenario performing the DCT and quantization computations in the OpenCL parallel implementation.

For OpenCL DCT and quantization kernel invocations, mapping-1 and mapping-2 are applied to the CPU and GPU, respectively. The OpenCL DCT kernel on the GPU uses synchronization (barrier) to ensure the correct execution sequence of the work-items as the DCT algorithm processes the 8 × 8 block (see Fig. 8). This is done in row and column fashion: work-items process first the rows and then the columns, as shown in Fig. 8. Mapping-2 synchronization causes excessive overhead on the CPU, which results in excessive context switching between the work-items because the CPU core can run only one work-item at a time. Therefore, we applied mapping-1 for the DCT kernel on the CPU, where a single work-item processes an entire 8 × 8 block, as shown in Fig. 9.

4.6 Device-based kernel optimization

A highly useful feature in OpenCL is the kernel code portability, enabling the same kernel code to be compiled and run on various OpenCL-enabled devices [24–27], such

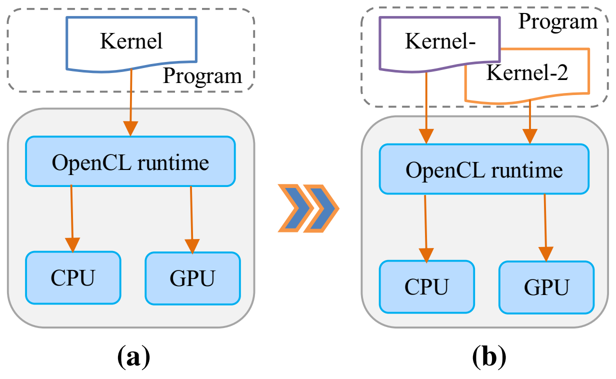


Fig. 10 OpenCL kernel codes for **a** naïve and **b** proposed

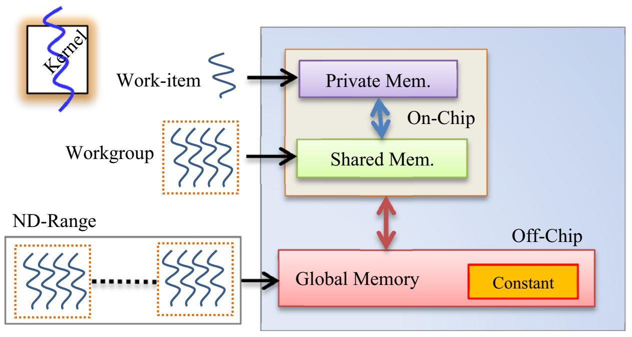


Fig. 11 Memory hierarchy in the GPU and correlations with OpenCL work-items

as CPUs, GPUs, and DSPs. With this cross-platform technology, a kernel code can be written once and can run on all devices, despite their differing hardware architectures. However, a kernel code does not deliver the same performance on two computing devices with different architectures, such as a CPU and a GPU. This is because the OpenCL represents all of the target machines with a unified memory and execution model. In a heterogeneous system, the devices have different architectures and performance capabilities; therefore, applying one particular optimized code for a specific device, such as a GPU, will not suit another, such as a CPU. We propose developing a specific version of the kernel code for each device. Figure 10 shows the differences between the naïve and proposed approaches. In our OpenCL parallel design, we apply the proposed approach to separate the kernel code into two versions—kernel-1 for CPUs and kernel-2 for GPUs—instead of one kernel for both of them. Our proposed approach, shown in Fig. 10 (b), allows us to develop a kernel code and apply specific optimization techniques that suit each device to achieve the optimal performance.

In our OpenCL implementation, each of the DCT, quantization, and DCT_QNTZ kernels is developed into two versions, the CPU and GPU kernel versions. The GPU kernel versions for DCT and DCT_QNTZ are optimized to use a low-latency local shared memory called an on-chip memory (see Fig. 11). The local shared memory is visible only to work-items within the same workgroup and is as fast as the registers; therefore, it significantly reduces the memory access overhead. The local shared memory is used as a temporary buffer to hold the elements of the DCT block during calculations of the DCT row–column-based computations. All memory access during the DCT block processing occurs between the private and shared memories. (For example, on-chip memory accessing takes fewer cycles than off-chip memory accessing, which takes a few hundred cycles). Once the computations finish, the DCT block is flushed back to the global memory (off-chip, high-latency memory). The default memory in the GPU is the global memory; therefore, using the local shared memory requires manually writing a code in the kernel. However, the local shared memory in the CPU kernels does not greatly benefit performance, because the OpenCL driver maps the local shared memory into the host CPU memory. Thus, local shared memory and global memory accesses are achieved the same speed on the CPU. Figure 11 illustrates the GPU memory hierarchy according to the OpenCL memory model, and correlations with work-items.

The chrominance quantization table coefficients are loaded into a constant memory, which is cacheable memory and visible to all work-items on the global ND-Rage. Constant memory is a small part of the global memory. It supports read-only operations during the kernel executions.

4.7 Optimizing the memory accesses

The OpenCL kernels versions for the CPU and the GPU have been optimized to use vectors; therefore, all accesses (read/write) from/to the memory are handled based on vectors, instead of scalars. Using vectors significantly reduces the memory access overhead and enhance the path utilizations. In our OpenCL kernels, we employ *vload8*() and *vstore8*() instructions to load and store a vector of size eight. Only one memory transfer (of 256-bits) is required to read/write one row (eight elements) of an 8×8 block. Thus, only eight read/write operations are necessary to load/store an 8×8 block from/to the global memory, in contrast to the 64 read/write operations needed when dealing with scalars. Figure 12 shows the OpenCL scalar and vector data types.

5 Experimental results

In this section, we discuss the experimental results of sequential and parallel implementations. In the sequential version, the forward DCT and quantization source codes were compiled and executed in two consecutive stages

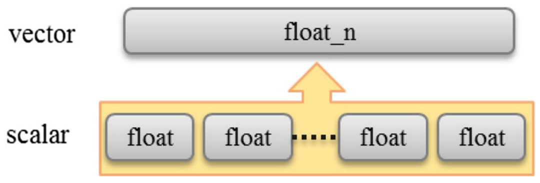


Fig. 12 OpenCL kernels use vectors rather than scalars

using a single host thread on the CPU, and the results were obtained for comparison purposes. The source codes of the forward DCT and quantization implementations are C-based, as given in the Turbo-JPEG library. In the JPEG library, the forward DCT has different implementations; we chose the float-type implementation for its accuracy and because floating point operations are highly optimized on modern computers, especially on the GPU. The default JPEG standards quantization tables and image quality values (e.g., quality = 75) are used to produce the quantization coefficients.

The source code for our parallel implementation is developed using OpenCL, which consists of two parts: a host program and kernel codes. The host CPU program is c-based with OpenCL API calls, and the kernel codes are the DCT and quantization implementations. Three kernels—DCT-kernel, QNTZ-kernel, and DCT-QNTZ-kernel—were developed in two versions (CPU-based and GPU-based) to reflect the proposed working-scenarios and optimization methods described early in the design section.

The inputs for our experiment are color images with different sizes in full/down-sampled resolution formats (e.g., 4:4:4/4:2:0). The execution times of the sequential and parallel implementations are measured in milliseconds, whereas the OpenCL kernels execution times and data-transfer overheads are calculated for each device based on the given workload. Our experiments are carried out on a heterogeneous platform consisting of a multicore CPU and a GPU. The specifications of the test platform are given in Table 5.

The total execution time of an OpenCL program consists of the kernel execution and data transfer overhead on each device. The OpenCL kernel execution is effected by workgroup size. Workgroup size represents the number of work-items per workgroup and is considered to be state-of-art in terms of the kernel execution time. To evaluate the effect of workgroup size for the OpenCL DCT and quantization kernels on both CPUs and GPUs, we conduct our experiment by considering different workgroup sizes with each kernel invocation (e.g., 64/128/256). We vary the number of work-items in a workgroup by changing the *LocalWorkSize* parameter before each kernel call and maintain the same workgroup size for each kernel invocation on the CPU and GPU at a time. Figure 13 shows the

Table 5 Test platform specifications

Hardware platform	Software platform
CPU model: AMD Athlon^{TM} II X2 250	Operating system: Windows 7, 32 bits Professional Edition
# of compute units: 2 (dual-core)	Software environment: AMD APP SDK v2.8, OpenCL API Compatibility 1.2 and Microsoft Visual Studio 2010
Frequency: 3.0 GHz	
CPU memory size: 3 GB	
GPU model: AMD Radeon^{TM} HD 6850	
# of compute units: 12	
Core frequency: 702 MHz	
GPU memory size: 1 GB	

performance of the OpenCL kernels' execution times for different workgroup sizes on the CPU and the GPU.

Each OpenCL kernel runs in two versions (CPU-based and GPU-based) in which different work-item mappings and workload allocations are applied. For example, a CPU-based kernel applies mapping-1 and works on the Y component, whereas a GPU-based kernel applies mapping-2 and works on C_bC_r components. Therefore, each work-item on the CPU and GPU has a different thread granularity.

According to the results, shown in Fig. 13, the best execution times for the DCT and quantization kernels are recorded for a workgroup size of 256. This execution time is the pure kernel execution without the data movements overhead. The OpenCL kernels' execution time decreases as the workgroup size increases. The rates of decreasing are different on the CPU and the GPU. However, the optimal kernel execution time is achieved with 256 as the maximum workgroup size for both the CPU and the GPU, and where they show their best performances.

Therefore, in our experiments, we chose 256 as the maximum workgroup size for the optimal execution time to calculate the DCT and quantization kernels.

Figure 14 shows the difference between the working scenarios in the total kernel execution time without considering the data transfer overhead. The DCT and quantization kernels' execution times on the CPU and GPU were accumulated for each working scenario separately, with different image sizes and formats. As shown in the results plotted in Fig. 14, working scenario-2 (WS2) achieves lower execution time than working scenario-1 (WS1) with medium and large images (e.g., 1024 × 1024 and 2048 × 2048 in 4:4:4 and 4:2:0 formats). However, WS2 achieves roughly the same or larger execution times than WS1 with an image size of 512 × 512. Therefore, working scenario-2 improves the kernel execution time by reducing the memory access overhead and increasing the data

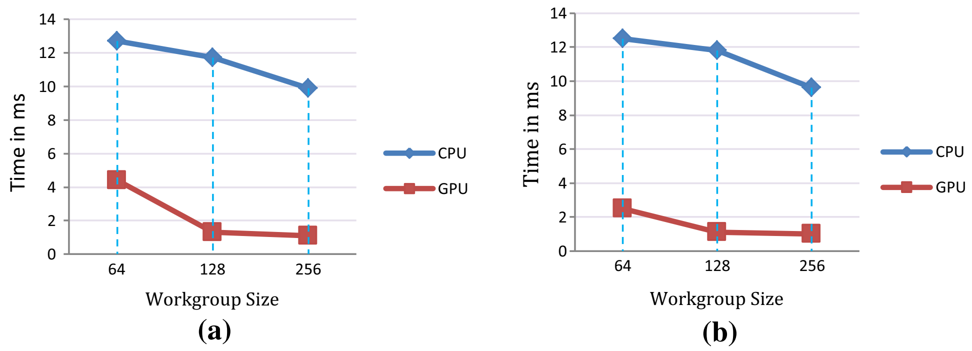


Fig. 13 Performance of OpenCL kernels with different workgroup sizes on the CPU and the GPU; **a** DCT kernel execution time and **b** quantization kernel execution time

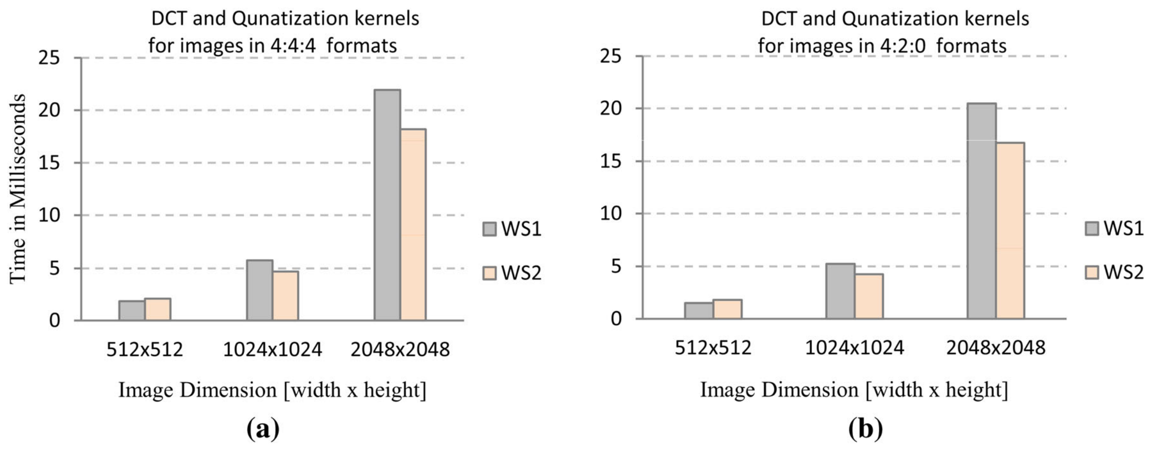


Fig. 14 Comparison of the working scenarios (WS1 and WS2) for the total kernels execution times with images in **a** 4:4:4 formats and **b** 4:2:0 formats

locality. Therefore, working scenario-2 is usually optimal with large size images.

Data transfer overhead is one of the most important issues in OpenCL applications. We investigate the amount of overhead generated by data movements in host-to-device and device-to-host transfers (e.g., H2D-write and D2H-read, respectively) for the targeted devices. We applied multiple mechanisms to investigate the impact of each memory transfer from device to device and measure the size of the generated overhead. Figure 15 shows the applied data transfer mechanisms and the amount of overhead on each device.

The experiment used two data transfer mechanisms for the CPU—clWriteBuffer/clReadBuffer (mem-copy) and clMapBuffer/clUnMapBuffer—and pinned host memory for the GPU (zero-copy). Figure 15 shows the amounts of overhead for these mechanisms on the CPU and GPU. In the write-read mechanism, the amount of overhead increases with image size on both the CPU and GPU; however, it is larger on the GPU because the data are copied back and forth through the PCIe bus, which has limited bandwidth. The write-read mechanism also generates

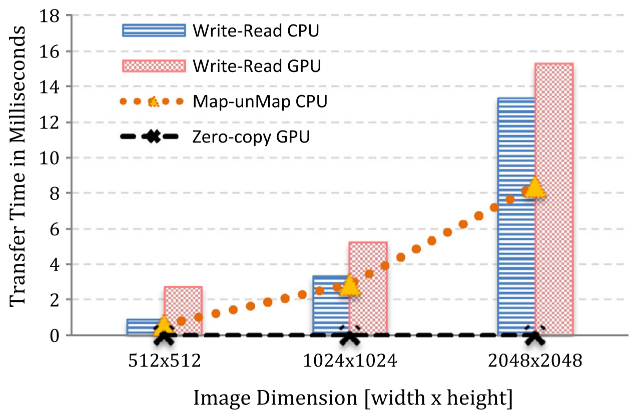
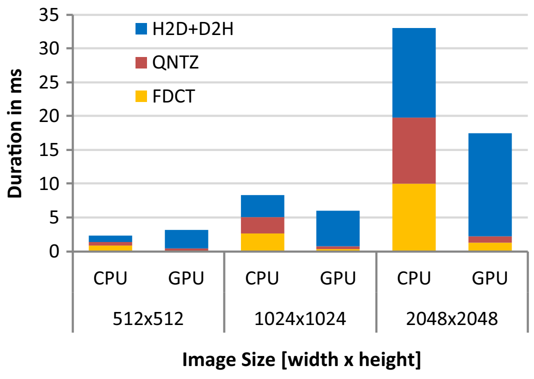


Fig. 15 The data-transfer overhead on the CPU and the GPU using different mechanisms

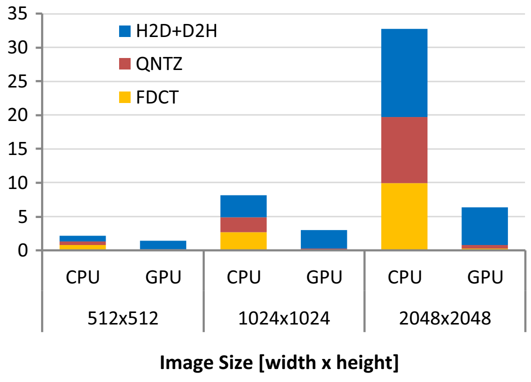
serious overhead on the CPU because OpenCL needs to copy the data between the buffers on the host-side. Performing memory copy requires pinning and unpinning in the buffer. To reduce the overhead in the CPU, a map/unmap mechanism (2) is applied, which generates less overhead than a write/read operation. Memory mapping is very inexpensive and avoids a great deal of data copying by mapping the OpenCL buffer into a host memory address and thus accessing the buffer with full host memory bandwidth. To avoid copying large amounts of data from/to the GPU, a zero-copy mechanism is applied in which a buffer resides in the pre-pinned host memory (pinned memory), and the GPU kernels access the pinned buffers through the PCIe bus with the maximum bandwidth. The zero-copy buffer mechanism generates very minimal overhead, although it adds some amount of overhead to the GPU kernels' execution time because the buffer resides on the host's pinned memory.

To study the execution time behavior for the proposed working scenarios with the related data transfer mechanisms, execution times were calculated for different image sizes in different formats, as shown in Figs. 16, 17, and 18. The results reveal the execution times for OpenCL kernels (DCT and quantization), including the associated data-transfer overhead on both CPU and GPU devices. H2D + D2H represent the data-transfer overhead, and DCT, QNTZ, and DCT + QNTZ represent the kernels' execution times.

Figure 16 shows the details of execution time for working scenario-1 with the write-read data transfer mechanism for each device with different image sizes and formats. The execution time consists of the OpenCL kernels' (FDCT, QNTZ) execution time, in addition to the H2D + D2H transfer time obtained using the write-read mechanism. As shown in the related graph, the kernels' execution times are minimal on the GPU compared to the CPU, due to the higher computational capability of the GPU. However, the H2D + D2H overhead is larger on the GPU than the CPU because the buffers must be copied from/to the GPU memory. Similarly, Fig. 17 shows the execution time for our experiment with working scenario-2 and the same write-read data transfer mechanism. Obviously, the kernel (DCT + QNTZ) execution times are

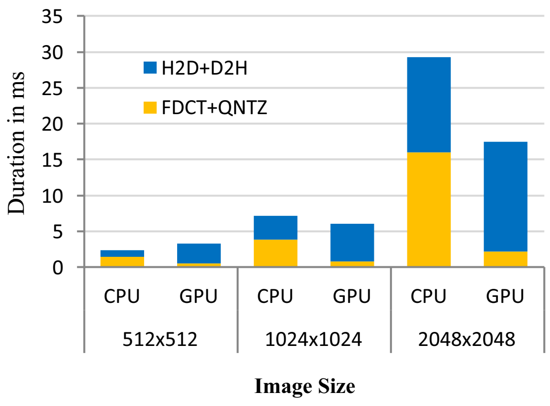


(a)

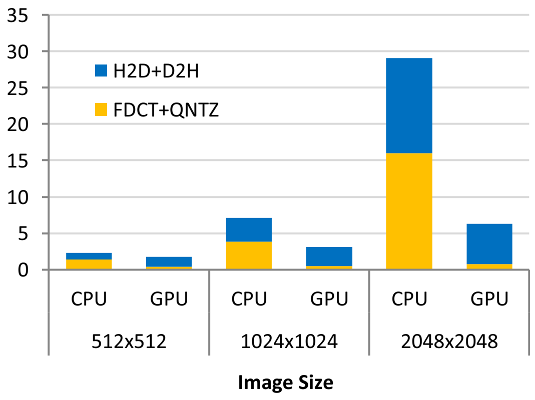


(b)

Fig. 16 OpenCL kernels + overhead on the CPU and the GPU using working scenario-1 with the write-read data transfer mechanism for images in **a** 4:4:4 formats and **b** 4:2:0 formats



(a)



(b)

Fig. 17 OpenCL kernels + overhead on the CPU and the GPU using working scenario-2 with the write-read data transfer mechanism for images in **a** 4:4:4 formats and **b** 4:2:0 formats

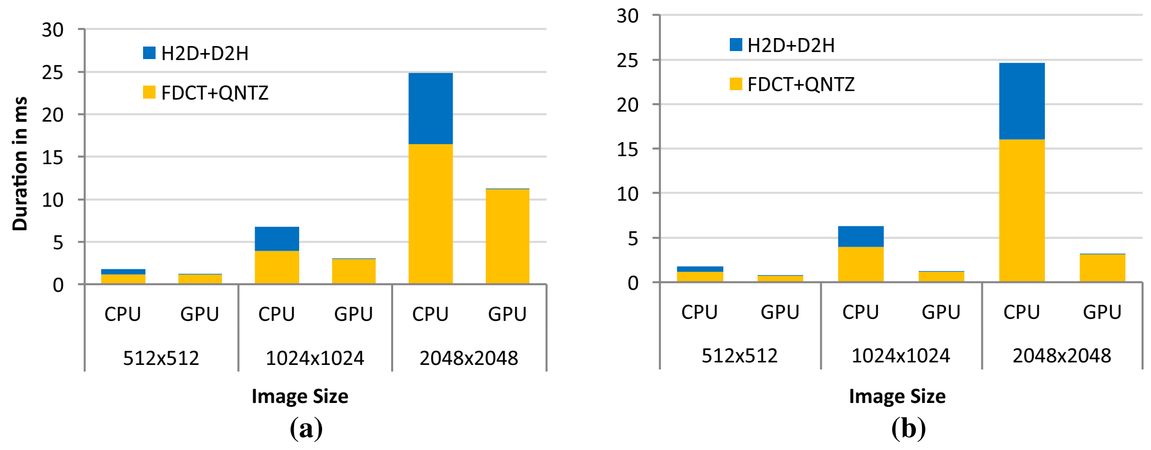
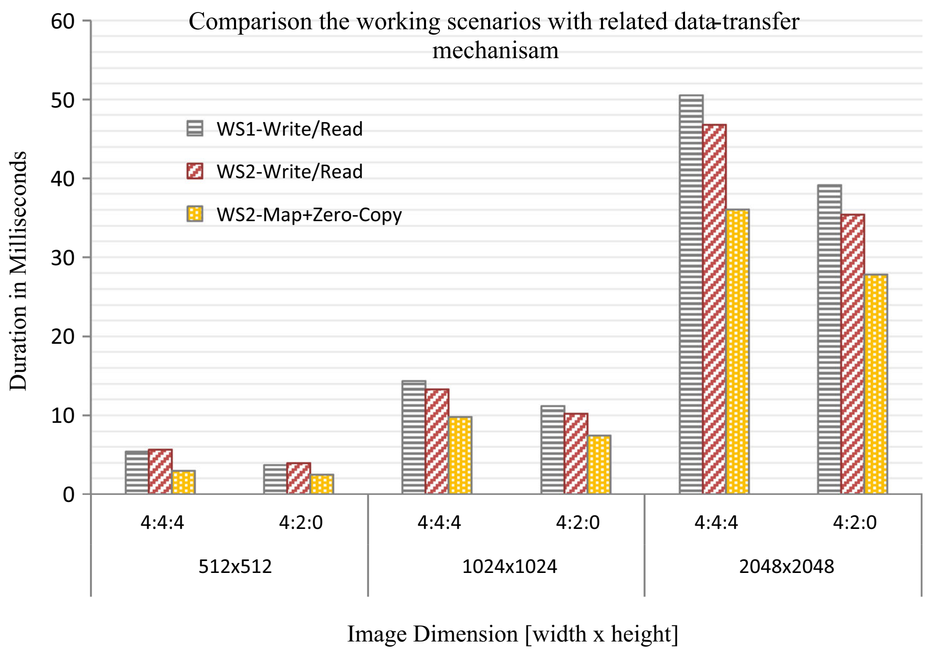


Fig. 18 OpenCL kernels + overhead on the CPU and the GPU using working scenario-2 with memory map for the CPU and zero-copy on the GPU data transfer mechanism for images in **a** 4:4:4 formats and **b** 4:2:0 formats

Fig. 19 Comparison of the total OpenCL execution time (kernels plus overhead) among the workings scenarios and the related data-transfer mechanisms



reduced on both the CPU and the GPU, whereas the H2D + D2H overhead remains the same as with working scenario-1 because the same mechanism is used. As shown in Fig. 18, the story is changed for working scenario-2 with map-unmap for the CPU and zero-copy for the GPU mechanism, where the overhead in both the CPU and the GPU is greatly reduced. However, the kernels' execution time (FDCT + QNTZ) increased on the GPU. These results show a reflexive relation between the kernels' execution times and overhead on a GPU using the zero-copy buffer mechanism. The kernels' execution times increased on the GPU due to the increased time required to access the buffer residing on the host pinned memory through the PCIe bus. Despite the increased kernels' execution times on the GPU, working scenario-2 with the zero-copy mechanism shows the optimal execution time. According to the results plotted in Figs. 16, 17, and 18, the optimal execution time for the OpenCL kernels is achieved using working scenario-2, and the lowest overhead is achieved using memory mapping on the CPU and zero-copy buffer on the GPU.

Figure 19 summarizes the results plotted in Figs. 16, 17, and 18, providing a clear comparison among those working scenarios applying their related data transfer mechanisms. The total OpenCL execution time is the accumulated execution times of the kernels and H2D + D2H overhead on both the CPU and the GPU. As shown in the graph, working scenario-2 with the map + zero-copy mechanism achieved the lowest execution time for all the image sizes and formats. This scenario (WS2) achieves the optimal execution time because it reduces the kernels' (FDCT, QNTZ) execution time by combining the kernels into a single step instead of two separate consecutive steps. Additionally, WS2 reduces overhead using memory mapping on the host CPU and zero-copy on the GPU, which avoids data copying to the GPU memory. These results are very promising, especially for a platform that includes a CPU and a GPU on a single die and shares the same memory, such as an APU (application processor unit) or a mobile platform.

6 OpenCL performance evaluation

In this section, we evaluate the results obtained for our OpenCL parallel implementations and compare them to the sequential implementation in terms of execution time. The comparison involves three versions of OpenCL parallel implementations of the proposed working scenarios (WS1 and WS2), each with a particular data transfer mechanism. Figure 20 compares the performances of parallel and serial executions, where a shorter execution time is a batter. It is clear that the OpenCL parallel implementations achieve significantly better performance than the sequential CPU-based program. Great reductions in execution time (e.g., 80 %) are obtained for working scenario-2 with its related mechanism of data transfer (WS2-MAP-ZC in OpenCL) with various image sizes in full and down-sampled resolution formats.

Table 6 summarizes the serial and parallel execution times. The serial execution times were obtained by running the C-based program with a single thread. The parallel execution times were obtained by running the OpenCL implementations on a heterogeneous platform, consisting of a multicore CPU and a GPU. The overall OpenCL execution time involves the OpenCL kernels' execution times based on the data transfer time for a particular device with its given workload.

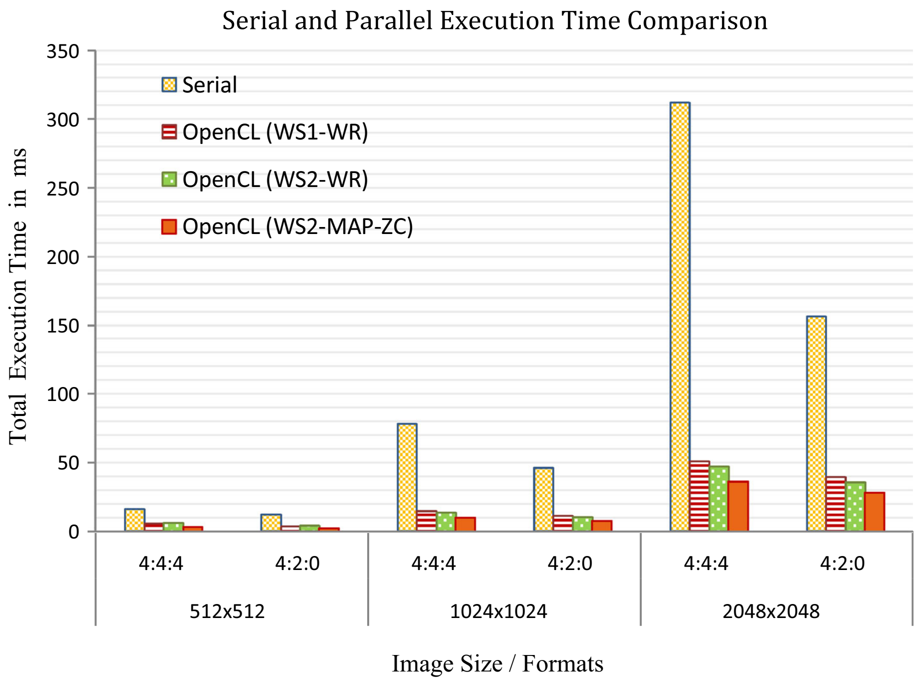


Fig. 20 The overall OpenCL execution times with different working scenarios and related data-transfer mechanisms, comparing parallel with serial execution times

Table 6 Serial and parallel execution times (in milliseconds)

Image size	Image formats	CPU-only	Overall OpenCL parallel execution time on (CPU + GPU)		
		Serial	WS1-WR	WS2-WR	WS2-MAP-ZC
512 × 512	4:4:4	16	5.43	5.68	2.99
	4:2:0	12	3.66	3.96	2.44
1024 × 1024	4:4:4	78	14.32	13.25	9.78
	4:2:0	46	11.17	10.18	7.42
2048 × 2048	4:4:4	312	50.55	46.79	36.08
	4:2:0	156	39.11	35.39	27.80

WS working scenario, *WR* write-read, *MAP-ZC* memory map with zero-copy buffer

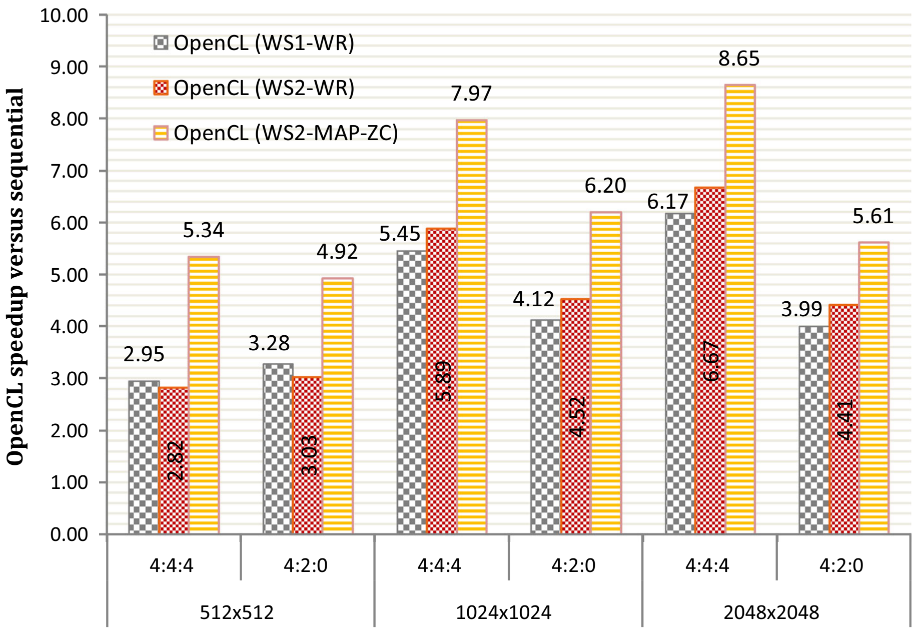


Fig. 21 Speedup using OpenCL parallel DCT and quantization compared to using the serial program

The OpenCL speedups were calculated based on the overall OpenCL execution time, as described below:

$$\text{Speedup} = \frac{T_{\text{serial}}}{T_{\text{parallel}}}, \tag{5}$$

where T_{serial} is the sequential execution time for a single thread on the CPU and T_{parallel} is the parallel execution time for an OpenCL implementation on the CPU + GPU.

$$T_{\text{parallel}} \text{ is defined by } T_p = T_{\text{kernel}} + T_{\text{overhead}}, \tag{6}$$

where T_{kernel} is the total of the execution times of the OpenCL kernels on the CPU and the GPU, and T_{overhead} is the total of the data transfer overhead on the CPU and the GPU.

The calculated speedups for the OpenCL implementations relative to the sequential program implementations are calculated using the speedup formulas defined in (5) and (6). These speedup results are calculated for each working scenario, separately, along with its particular data-transfer mechanism. Figure 21 shows the speed gains for the OpenCL parallel program over the sequential single-thread C-based program, which are significant in both working scenarios, particularly working scenario-2. The maximum speedups are 7.97 and 8.65 with large images—1024 × 1024 and 2048 × 2048, respectively, in 4:4:4 formats—which occur when using the OpenCL implementation employing working scenario-2 with the zero-copy buffer. The speedups were 6.20 and 5.61 with images of the same two sizes in down-sampled resolution 4:2:0 format. Even though working scenario-2 with write-read mechanism (WS2-WR) achieves relatively good speedup compared to working scenario-1 with the same data transfer mechanism (WS1-WR), except in one case, it achieves a slightly lower speedup when medium-sized images (e.g., 512 × 512 in both 4:4:4 and 4:2:0 formats) are used as inputs. Overall all, significantly better performance and maximum speedup gains were obtained using the second working scenario (e.g., SW2-MAP-ZC). This study's results demonstrate that merging the DCT and quantization computations into a single step improves the execution time significantly and achieves optimum performance.

7 Conclusions and future work

An efficient OpenCL parallel implementation for the forward DCT and quantization of JPEG image compression is presented in this paper. We leverage the computing capabilities of modern computing systems to accelerate the computations on a heterogeneous platform consisting of a multicore CPU and a GPU. Two working scenarios are used to perform the DCT and quantization computations on the CPU and the GPU in parallel. The first scenario follows the classical processing method, as in baseline JPEG, where a computation is done in two consecutive steps; for example a DCT computation is performed first, followed by quantization, sequentially. In contrast, the second scenario performs the computations in a single step that merges the DCT and quantization steps. An optimized OpenCL kernel code has been developed for each step of the DCT and quantization computations, to reflect the working scenarios. Therefore, three OpenCL kernel codes (DCT, QNTZ, and DCT_QNTZ) were developed, each in two versions: CPU-based and GPU-based kernels. In these device-based kernels, we easily can apply certain optimization techniques that match the hardware architecture and assign different work-item mappings, granularities, workload sizes, and optimal workgroup sizes before launching each kernel on the target device. Two data transfer mechanisms were used in our OpenCL experiments. Analysis of the effects of these data-transfer mechanisms showed that a memory map mechanism is optimal for reducing the overhead on the CPU, and the pre-pinned host resident buffer with zero copy is the optimal choice to avoid copying data back and forth to/from the GPU memory across the PCIe bus.

We evaluated our OpenCL parallel implementation with various full- and down-sampled-resolution image sizes using both the investigated working scenarios. OpenCL implementation achieved significantly better performance than the sequential program. The proposed OpenCL working scenario-2 with optimal data transfer mechanism was able to achieve speedups of 7.97 and 8.65 relative to the sequential method, for large images of 1024×1024 and 2048×2048 in 4:4:4 format.

In this paper, we addressed a variety of factors affecting execution time, including working scenarios, data transfer mechanisms, workload allocation, work-items mapping, thread granularity, workgroup size, and device-based kernels, along with certain optimizations. Based on our experimental results, the key findings of our OpenCL implementation that affect performance are as follows:

1. Combine DCT and quantization computations into one step to increase data locality and reduce memory access overhead.
2. Use the optimal data transfer mechanism for the least overhead, such as memory mapping for the CPU.
3. Use a different mapping on each device, such as one work-item for one 8×8 block on the CPU and eight work-items for one block on the GPU for the DCT computations.
4. Use large workgroups to improve the kernel execution time.
5. Separate a kernel code into two versions, one for the CPU and one for the GPU.
6. Avoid using a barrier in the CPU kernel to avoid excessive context-switch overhead.
7. Optimize the kernel to write/read from/to the memory using vectors rather than scalars, to fully utilize the path and reduce memory accessing.
8. Optimize the GPU kernel code to take advantages of on-chip local shared memory.
9. Assign proper granularity on each device, for example, coarse-grain for the CPU and fine-grain for the GPU.

These highlighted issues affect OpenCL performance, as shown by the experimental results obtained for the DCT and quantization computations.

In this paper, we started with DCT and quantization two-stage of JPEG image compression as a case study to determine the optimal performance of each device in a heterogeneous system using OpenCL. Later on, this work will be extended to involve all other stages of the JPEG encoder as future work by disturbing the workload between the CPU and GPU to enhance the performance and achieve the real-time processing. The target of this work is to develop an efficient OpenCL-based parallel implementation for the JPEG encoder that will run in different hardware as OpenCL supports code portability. This will be very useful for the low-power and handheld devices, where a real-time software image codec will replace the hardware codec in the future.

Acknowledgments This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and future Planning (NRF-2012R1A1A2043400).

References

1. John, O., Mike, H., David, L., Simon, G., John, S., James, P.: GPU computing. Proc. IEEE **96**(5), 879–899 (2008)
2. Stephen, K., William, D., Brucek, K., Michael, G., David, G.: GPUs and the future of parallel computing. Micro IEEE **31**(5), 7–17 (2011)
3. John, S., David, G., Shi, G.: OpenCL: a parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(3), 66–73 (2010)
4. Barak, A., Ben-Nun, T., Levy, E., Shiloh, A.: A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In IEEE International Conference on Cluster Computing, Heraklion, Crete (2010)
5. Samsung Galaxy S5: Samsung, (Online). http://en.wikipedia.org/wiki/Samsung_Galaxy_S5. Accessed 2 Jan 2015
6. Yun, H.S., Shi, Q.: Image and video compression for multimedia engineering. CRC Press, New York (2008)
7. Ruby, L., John, B., Joel, L., Kenneth, S.: Real-time software MPEG video decoder on multimedia-enhanced PA-7100LC processors. Hewlett-Packard J. **46**(2), 60–68 (1995)
8. Furht, B.: A survey of multimedia compression techniques and standards. Part I: JPEG standard. Real-Time Imaging **1**(1), 49–67 (1995)
9. Agostini, L., Bampi, S.: Integrated digital architecture for JPEG image compression. In: European Conference on Circuit Theory and Design, Espoo, Finland (2001)
10. Rabadi, W., Talluri, R., Illgner, K.: Programmable DSP platform for digital still cameras. Texas Instrutments (2000)
11. Li, S., Qu, X., Li, Q.: Implementation of the JPEG On DSP processors. Appl. Mech. Mater. **34–35**, 1536–1539 (2010)
12. Min, J., Markandey, V.: Optimizing JPEG on the TMS320C6211 2-level cache DSP. Digital Signal Processing Solutions (2000)
13. Mohanty, S.P.: GPU-CPU multi-core for real-time signal processing. In: International Conference on Consumer Electronics ICCE ‘09 (2009)
14. Tokdemir, S., Belkasim, S.: Parallel processing of DCT on GPU. In: Data Compression Conference (DCC), Snowbird, UT (2011)
15. Duo, L., Ya, F.X.: Parallel program design for JPEG compression. In: 9th International Conference on Fuzzy Systems and Knowledge Discovery (2012)

16. Yang, Z., Zhu, Y., Pu, Y.: Parallel image processing based on CUDA. In: International Conference on Computer Science and Software Engineering (2008)
17. Nvidia SDK 9.52 code samples—transform, discrete cosine (Online). http://developer.download.nvidia.com/SDK/9.5/Samples/gpgpu_samples.html. Accessed 11 Dec 2014
18. AMD APP SDK Samples—DCT, AMD (Online). http://amddevcentral.com/tools/hc/AMDAPPSDK/samples/Pages/default.aspx. Accessed 11 Dec 2014
19. Kou, W.: Digital image compression algorithms and standards. Kluwer Academic Publishers, Dordrecht (1995)
20. Mitchell, J.L., Pennebaker, W.B.: JPEG still image data compression standard. International Thomson, New York (1993)
21. Thyagrajan, K.S.: Still image and video compression with Matlab. Wiley, New York (2011)
22. Wallace, G.K.: The JPEG still picture compression standard. IEEE Trans. **38**, xviii–xxxiv (1991)
23. Yukihiro, A., Takeshi, A., Nakajima, M.: A fast DCT-SQ scheme for images. Trans. IEICE **E-71**(11), 1095–1097 (1988)
24. OpenCL: Khronos Group (Online). http://www.khronos.org/opencl/. Accessed 11 Dec 2014
25. Gaster, B., Howes, L., Kaeli, D., Mistry, P., Schaa, D.: Heterogeneous computing with OpenCL. Elsevier, Amsterdam (2012)
26. The OpenCL Specification Version: 1.1, Khronos OpenCL Working Group (2010)
27. Ralf Karrenberg, S.H.: Improving performance of OpenCL on CPUs. In: The 21st international conference on Compiler Construction, Berlin, Heidelberg (2012)
28. Pourazad, M., Doutre, C., Azimi, M., Nasiopoulos, P.: HEVC: the new gold standard for video compression: how does HEVC compare with H.264/AVC? IEEE Consum. Electron. Mag. **1**, 36–46 (2012)
29. Goldman, M.: High-efficiency video coding (HEVC): the next-generation compression technology. SMPTE Motion Imaging J. **121**(5), 27–33 (2012)
30. Pastuszak, G.: Hardware architectures for the H.265/HEVC discrete cosine transform. IET Image Process. (2014). doi:10.1049/iet-ipr.2014.0277
31. Meher, P., Park, S., Mohanty, B., Lim, K., Yeo, C.: Efficient integer DCT architectures for HEVC. IEEE Trans. Circuits Syst. Video Technol. **24**(1), 168–178 (2014)
32. Xun, C., Qunshan, G.: Improved HEVC lossless compression using two-stage coding with sub-frame level optimal quantization values. Image Processing (ICIP), 2014 IEEE International Conference, pp. 5651–5655, 27–30 (2014)

Nasser Alqudami received his B.S. degree and M.S. degree in computer science from Mosul University, Iraq and University Science of Malaysia (USM), Malaysia, in 2002, and 2009, respectively. Now, he is a Ph.D. student in the supercomputing lab., department of computer science, Yonsei University. His research interests include parallel and distributed computing, heterogeneous computing, and image compression.

Shin-Dug Kim received his Ph.D. degree in electrical and computer engineering in 1991 from Purdue University, USA. Now, he is a professor at the department of computer science, Yonsei University, and also head of the supercomputing laboratory. His research interests include parallel computing, grid computing, computer architecture, and ubiquitous computing.