Parallel LDPC Decoding on a Heterogeneous Platform using OpenCL

Jung-Hyun Hong[1], Joo-Yul Park[2] and Ki-Seok Chung[2]
[1] Department of Electronics and Computer Engineering, Hanyang University, Seoul, Korea
[e-mail : jhhong34@hanyang.ac.kr]
[2] Department of Electronic Engineering, Hanyang University, Seoul, Korea
[e-mail : jooyul.park@gmail.com, kchung@hanyang.ac.kr]
*Corresponding author: Ki-Seok Chung

Received October 28, 2015; revised March 15, 2016; accepted May 18, 2016; published June 30, 2016

Abstract

Modern mobile devices are equipped with various accelerated processing units to handle computationally intensive applications; therefore, Open Computing Language (OpenCL) has been proposed to fully take advantage of the computational power in heterogeneous systems. This article introduces a parallel software decoder of Low Density Parity Check (LDPC) codes on an embedded heterogeneous platform using an OpenCL framework. The LDPC code is one of the most popular and strongest error correcting codes for mobile communication systems. Each step of LDPC decoding has different parallelization characteristics. In the proposed LDPC decoder, steps suitable for task-level parallelization are executed on the multi-core central processing unit (CPU), and steps suitable for data-level parallelization are processed by the graphics processing unit (GPU). To improve the performance of OpenCL kernels for LDPC decoding operations, explicit thread scheduling, vectorization, and effective data transfer techniques are applied. The proposed LDPC decoder achieves high performance and high power efficiency by using heterogeneous multi-core processors on a unified computing framework.

Keywords: Error correcting code, LDPC decoder, parallel processing, heterogeneous computing, OpenCL

This research was supported by the MSIP (Ministry of Science, ICT&Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2016-H8501-16-1005) supervised by the IITP (Institute for Information&communications Technology Promotion).

1. Introduction

Modern wireless devices transmit and receive high-rate data in real time, and the complexity of digital signal processing applications is increasing rapidly. Therefore, various hardware accelerators are commonly used to efficiently process computationally intensive applications. Typically, mobile devices are equipped with a multi-core central processing unit (CPU) and a multi-core graphics processing unit (GPU). Due to these heterogeneous processing units, the potential for application-specific customization and parallelization of mobile applications has increased considerably.

However, it is not easy to take full advantage of heterogeneous devices simultaneously because they have distinct hardware structures and instruction sets. Furthermore, most hardware vendors support programming models that work only for their own computing platforms [1]. Therefore, diverse programming skills and detailed knowledge of hardware architectures are required for programmers to efficiently implement a target application. Consequently, it is desirable to have a standard programming framework independent of specific computing platforms.

Recently, Open Computing Language (OpenCL) has been developed to provide a framework that supports heterogeneous computing platforms. A key advantage of OpenCL is that programmers can access computing resources using standard runtime application programming interfaces (APIs) and libraries. If an application is designed to be compliant with the OpenCL specification, designers can cope with rapidly evolving hardware architectures and provide an optimized solution for the target device [2]. Therefore, parallelization of digital signal processing applications using the OpenCL framework can support various protocols and multiple code rates on heterogeneous platforms to achieve both high portability and high performance.

The Low Density Parity Check (LDPC) code is one of the strongest linear block error correcting codes, which detect and correct errors caused by unreliable communication channels. LDPC coding shows good bit error rate curves with few error floor issues; therefore, it is widely considered attractive for high-speed wireless communication applications such as local and metropolitan area networks, satellite communication, and mobile broadcasting [3]. The LDPC code has been adopted by more than 200 industrial standards such as IEEE 802.11 standards and the next generation standard of digital video broadcasting (DVB-S2X) [4]. This article introduces a parallel software decoder of LDPC codes for the China Multimedia Mobile Broadcasting (CMMB) standard on a mobile device. CMMB is a mobile television and multimedia standard developed and specified by the State Administration of Radio, Film, and Television (SARFT) of China [5].

An LDPC decoding algorithm can correct errors by repeatedly computing and exchanging messages. The amount of computation depends on the size of a sparse parity check matrix called H-matrix. As the size of an H-matrix increases, the amount of computation grows rapidly. Therefore, designing parallel LDPC decoders using multi-core processors has been actively studied to provide reliable high-speed data transmission [6]-[10]. However, even if most approaches could reduce the decoding time significantly, hardware resource utilization would be insufficient because existing models were parallelized for specific devices with hardware-dependent programming models.

This article presents effective parallelization techniques for LDPC decoders on a heterogeneous mobile platform with an OpenCL framework. Many high performance mobile processors include both CPU and GPU cores on a single silicon die to enable low power consumption and effective communication using a shared memory system. In general, task-level parallel applications run faster on multi-core CPUs, whereas data-level parallel applications tend to run faster on multi-core GPUs. Specifically, the address generation step in LDPC decoding is suitable for task-level parallelization, and the iterative decoding steps in LDPC are suitable for data-level parallelization. To improve the performance of the proposed LDPC decoder, explicit thread scheduling, vectorization, and effective data transfer techniques are applied. The proposed LDPC decoder satisfies the performance requirement of the CMMB standard and achieves both high performance and low power consumption by using both CPU and GPU cores intelligently.

The remainder of this article is organized as follows. Section 2 presents an overview of the OpenCL framework for heterogeneous computing. Section 3 provides a brief review of LDPC decoding. The proposed LDPC decoder customized for heterogeneous platforms is explained in Section 4. The experimental environment and results are presented in Section 5. Section 6 concludes this article with suggestions for future work.

2. Overview of OpenCL Framework for Heterogeneous Programming

OpenCL is an open industry standard computing framework for programming heterogeneous devices. The OpenCL framework includes a programming language, APIs, and libraries to support software development. The OpenCL specification is defined in a hierarchy of models: platform model, execution model, memory model, and programming model [2].

2.1 Platform Model

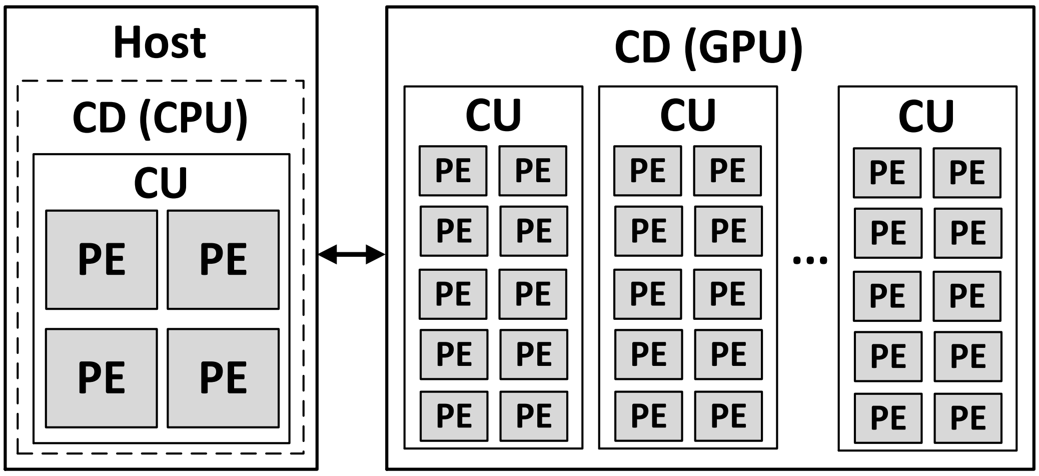


Fig. 1. OpenCL platform model for the proposed LDPC decoder

Fig. 1 shows the OpenCL platform model for the proposed LDPC decoder. The platform model for OpenCL consists of a host connected to one or more OpenCL compute devices (CDs), which are divided into one or more compute units (CUs). Each CU is further divided into one or more processing elements (PEs). OpenCL C functions, called kernels, are executed by each PE in parallel.

Although OpenCL provides functional portability by defining abstract device architecture, architecture-specific features of heterogeneous devices should be taken into account to improve the target kernel performance. As shown in **Fig. 1**, the multi-core CPU in the proposed platform model is defined as the CD as well as the host. In general, CPU cores are designed to process control-intensive applications, and they are adequate for task-level parallelization [11]. On the other hand, an advantage of the GPU is high computational throughput gained by using hundreds of PEs that support efficient context switching between groups of threads [12]. Therefore, GPUs are optimized for data-level parallelization. For the proposed decoder, the parallelization characteristics of LDPC decoding steps were considered carefully, and appropriate parallelization techniques were applied to kernels depending on target devices.

2.2 Execution Model

The execution model is defined by how PEs execute kernels. When a kernel is assigned to a processor for execution by the host, an index space is defined by the host program. A single kernel instance at a point in the index space is called a work-item, and work-items are grouped into a work-group. Work-groups are organized and assigned to a CU that contains multiple PEs on which each work-item is executed [2]. The host defines the context for kernel execution to manage OpenCL objects such as memory, program, kernel, and event. As shown in **Fig. 2**, the CPU and the GPU are defined in the same context to share memory objects and apply effective synchronization techniques using event objects.

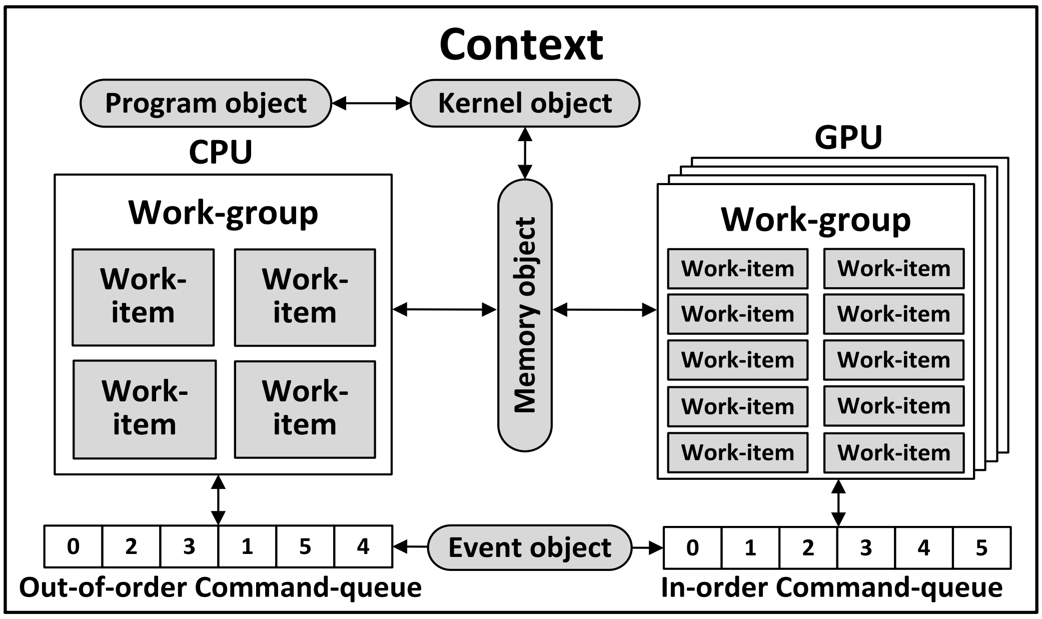


Fig. 2. OpenCL execution model for the proposed LDPC decoder

A command-queue coordinates communication activities between the host and CDs. In this work, each CPU and GPU has its own command-queue within a single context. In particular, the CPU uses an out-of-order command-queue to implement task-parallel programming. The GPU uses non-blocking commands for in-order queues to reduce the global synchronization overhead that can be incurred by unnecessary command batching [12].

2.3 Memory Model

Work-items that execute a kernel use the abstract memory hierarchy defined in the OpenCL memory model. In general, the host and CDs are physically independent and have separate memory spaces. Therefore, memory allocations on each device and explicit data transfers between the devices are required. However, as shown in **Fig. 3**, the CPU and the GPU of the target platform share the physical memory region defined as the global memory. Therefore, applying conventional data transfer techniques for the proposed LDPC decoder causes unnecessary memory allocations and data copies on the unified memory architecture. In this article, a method called zero-copy is applied to the proposed kernels to increase performance that directly accessing a buffer object in the global memory avoiding unnecessary memory allocations and runtime data transfers [13].

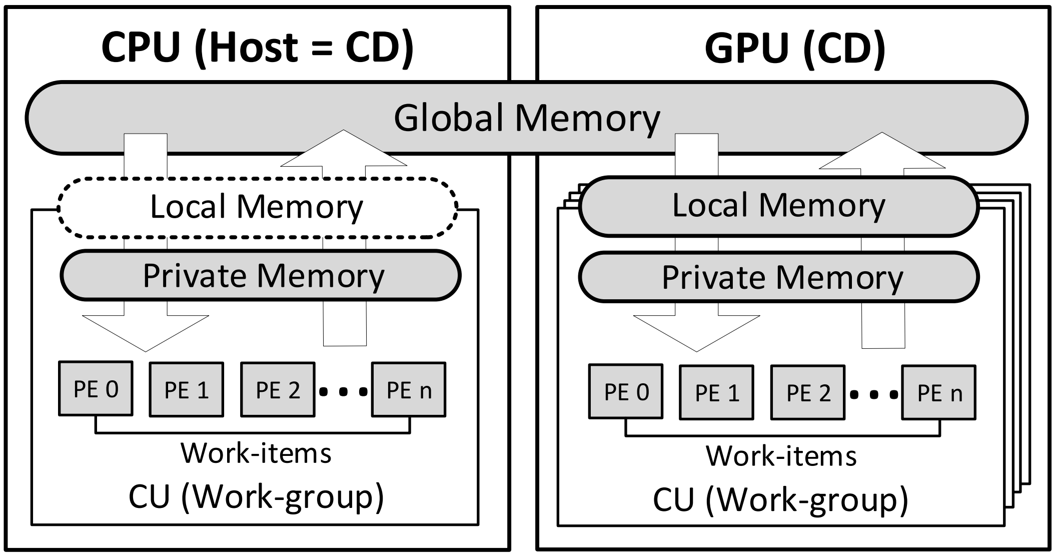


Fig. 3. OpenCL memory model for the proposed LDPC decoder

In the OpenCL framework, the local memory is shared by all work-items within a single work-group while the global memory is visible to all work-groups. On the GPU, this memory space is implemented as software-managed on-chip caches which have much shorter latency and higher bandwidth than the global memory. Therefore, the local memory of the GPU provides efficient communication and data transfer methods between work-items of the same work-group. However, all CPU memory objects are cached by hardware, and explicit management of the local memory can cause unnecessary overhead during kernel execution. Therefore, CPUs cannot take advantage of the OpenCL local memory region [12]. In this article, data transfer optimization methods using local memory for the GPU kernels are implemented. Details will be explained in Section 4.

2.4 Programming Model

As shown in **Fig. 4**, a task-level parallel execution method uses an out-of-order command-queue that can run multiple commands concurrently as soon as the device is ready. LDPC kernels for the CPU work in a pipeline manner with explicit synchronization techniques using event objects [12]. Event objects encapsulate the states of the issued commands, and a list of events can be passed to the command-queue as a dependency list. Issued commands will not begin executing until all of the input events have been completed.

The OpenCL framework provides a data-parallel programming model with concurrently executed work-items in a work-group [2]. In this work, explicit vectorization with single instruction multiple data (SIMD) instructions was applied to GPU kernels to fully utilize the large number of PEs in the GPU. A detailed description of the implementation will be given in Section 4.

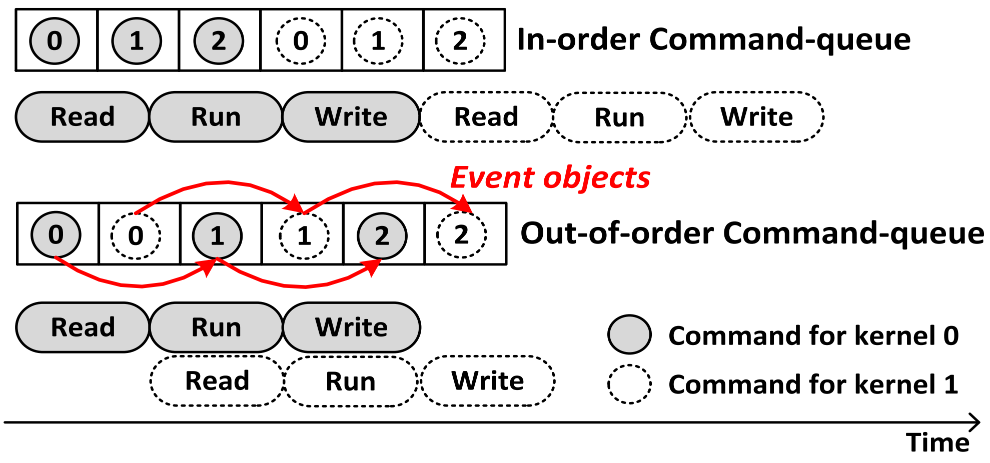


Fig. 4. OpenCL programming model for the proposed LDPC decoder

3. Review of LDPC Decoding Algorithm

3.1 LDPC Decoding Algorithm

LDPC is a linear block code, and decoding is carried out using a parity-check matrix called H-matrix. The rows and the columns of an H-matrix represent parity-check codes and symbols, respectively. LDPC codes are often represented by a bipartite graph in which the set of check nodes and the set of bit nodes compose two partite sets. The check nodes correspond to rows of the H-matrix, and the bit nodes correspond to its columns. When an H-matrix contains a fixed number of 1's in each row and each column, the weights of the column and the row are equal. An LDPC code with equal weights for both nodes is said to be regular. The (N, K) regular LDPC codes can be defined by an (M, N) H-matrix, where M and N represent the number of check nodes and bit nodes, respectively [3].

Table 1 shows the entire LDPC decoding algorithm. LDPC decoding consists of four main operations: initialization (INIT), check node processing (CNP), bit node processing (BNP), and parity check (PC). Most practical LDPC decoders are based on a concept of message passing called either belief propagation (BP) or sum-product algorithm (SPA). SPA is carried out by passing messages that contain an amount of belief quantified as 0 or 1 between adjacent nodes. Each node attempts to decode its own value based on the delivered messages. If the decoded value turns out to contain an error, the decoding process is repeated a pre-determined number of times. Typically, iterative LDPC decoding schemes use log-likelihood ratio (LLR) processing to deliver messages, which replaces expensive multiplication operations with inexpensive addition operations. Input LLR values received from the channel and the initial decision values are configured at the INIT step. Iterative decoding based on the delivered messages is processed during the CNP and BNP steps. The decoding process is iterated until the termination condition for the decoded words is satisfied at the PC step [10].

Table 1. LDPC decoding based on the sum-product algorithm (SPA)

Decoding Step	Algorithm
Initialization (INIT)	*H_{mn} : ***the value at (m,n) of H-matrix***
	set F_n = LLR for bit nodes ($n = 1,2,\dots,N$)
	set $Z_{mn} = F_n$ if H_{mn} is 1 for each (m,n)
Check node processing (CNP)	while ($i \le i_{max}$)
	*$N(m)$: ***the set of bit nodes including the check node m***
	for each check node
	where each (m,n) if H_{mn} is 1
	$T_{mn} = \prod_{n'\in N(m)\setminus\{n\}} \frac{1-e^{z_{mn'}}}{1+e^{z_{mn'}}}$
	$L_{mn} = ln\frac{1-T_{mn}}{1+T_{mn}}$
	end for
Bit node processing (BNP)	* ***M(n): the set of check nodes including the bit node n***
	for each bit node
	where each (m,n) if H_{mn} is 1
	$Z_{mn} = F_n + \sum_{m'\in M(n)\setminus\{m\}} L_{m'n}$
	$Z_n = F_n + \sum_{m\in M(n)} L_{mn}$
	end for
	*$\widehat{C}$: ***the decoded word***
	for each n ($n = 1,2,\dots,N$)
	$\widehat{C}=[\widehat{C}_1\dots\widehat{C}_N]$, $\widehat{C}_n=\begin{cases}0, & Z_n\ge 0\\ 1, & Z_n<0\end{cases}$
	end for
Parity check (PC)	* ***The parity check equation***
	if $H \cdot [\widehat{C}_1\dots\widehat{C}_N]^T$ is 0
	return success
	i++
	end while

3.2 Parallelization of LDPC Decoding

LDPC decoding can correct errors by repeatedly computing and exchanging messages. The amount of computation depends on the size of the H-matrix. However, recently published standards show that H-matrices are getting bigger as the amount of data transfer increases [10]. The huge size causes both decoding complexity and decoding time to increase. Therefore, it is crucial to distribute the workload to proper hardware accelerators and parallelize the computation efficiently.

For example, a recent study proposed techniques such as asynchronous data transfer and multi-stream concurrent kernel execution to utilize the GPU to run a WiMAX LDPC decoder [6]. In [7], a method for LDPC decoding using Compute Unified Device Architecture (CUDA) for the NVIDIA GPU is proposed. And a programming model for the low-power embedded CPUs is proposed in [8]. Falcão *et al*. proposed a portable LDPC decoder executed on a set of platforms ranging from multi-core CPUs to many-core GPUs [9]. Park *et al*. proposed a parallel software LDPC decoder using OpenMP for the CPU and CUDA for the GPU [10].

They showed that parallel LDPC decoding using multi-core processors could reduce decoding time dramatically. However, hardware resource utilization is not sufficiently high because they rely on a vendor-specific computing framework and different parallel programming models for the CPU and the GPU which make it hard to exploit hardware accelerators simultaneously.

This work proposes a novel parallelization technique for a software LDPC decoder to use both CPU and GPU in a unified programming framework. The platform, execution, memory, and programming models in the OpenCL framework are customized for LDPC decoding operations. Further, efficient computing resource management for heterogeneous devices is achieved using OpenCL objects in a single uniform context, as explained in the next section.

4. Parallel LDPC Decoder on a Heterogeneous Platform using OpenCL

The proposed OpenCL kernels for CMMB LDPC decoding were designed and customized to take full advantage of different target hardware architectures. Fig. 5 shows the execution flow of the proposed LDPC decoding kernels. An OpenCL task-parallel programming model was applied to the one-dimensional address generation kernel for execution by the CPU, and a data-parallel programming model was applied to the INIT, CNP, BNP, and PC kernels for execution by the GPU. Even if the CPU and the GPU are defined in a unified context, they employ separate command-queues. The CPU uses an out-of-order command-queue, whereas the GPU uses an in-order command-queue. Effective synchronization techniques and data transfer optimizations between defined OpenCL objects are applied to the host and kernel programs. These OpenCL models provide cross-platform programming environments that enable the proposed parallel LDPC decoder to improve performance by flexibly utilizing the heterogeneous platform.

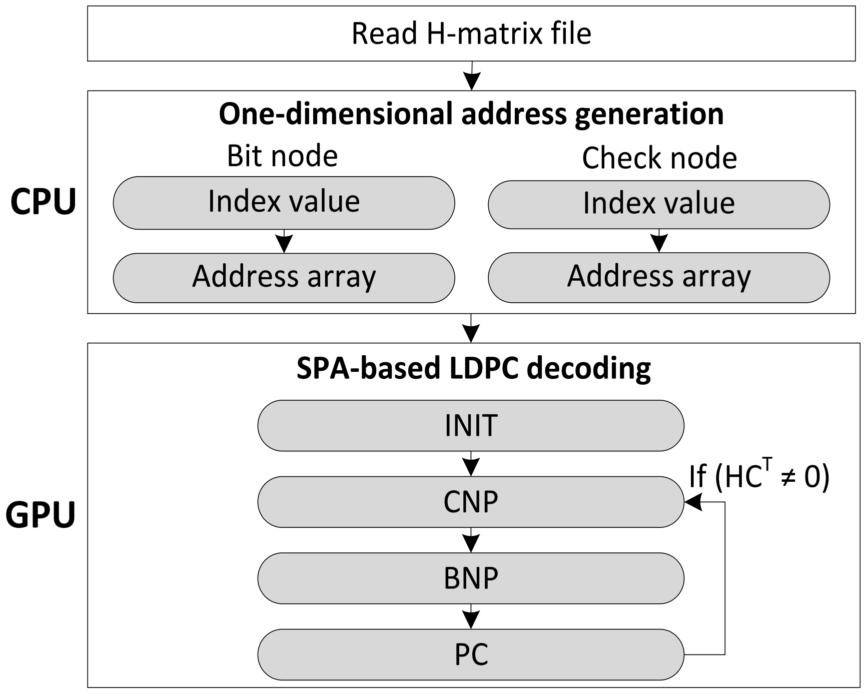
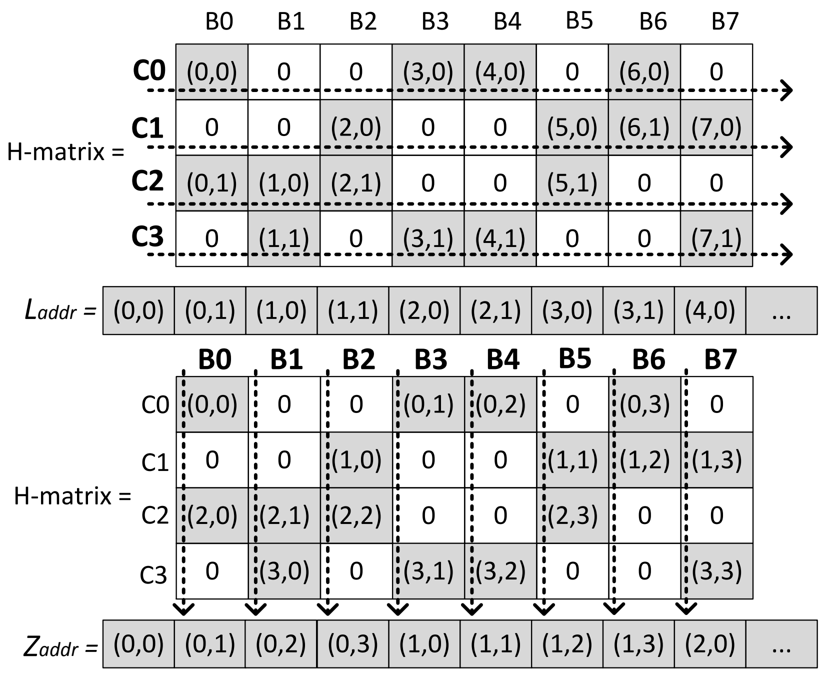


Fig. 5. Execution flow of the proposed LDPC decoding kernels

4.1 Design of One-dimensional Address Generation Kernels

In the LDPC decoding algorithm, LLR values are copied from the H-matrix at the INIT step, and updated values are stored back when the CNP and the BNP steps are conducted. The PC operation examines the termination condition using all check nodes connected to each bit node. To make it easier to parallelize the execution of LDPC decoding and minimize the number of memory accesses, the positions of the LLR values are rearranged as a one-dimensional array.



B0 B1 B2 B3 B4 B5 B6 B7
H-matrix =
C0 (0,0) 0 0 (3,0) (4,0) 0 (6,0) 0
C1 0 0 (2,0) 0 0 (5,0) (6,1) (7,0)
C2 (0,1) (1,0) (2,1) 0 0 (5,1) 0 0
C3 0 (1,1) 0 (3,1) (4,1) 0 0 (7,1)
Laddr = (0,0) (0,1) (1,0) (1,1) (2,0) (2,1) (3,0) (3,1) (4,0) ...
B0 B1 B2 B3 B4 B5 B6 B7
H-matrix =
C0 (0,0) 0 0 (0,1) (0,2) 0 (0,3) 0
C1 0 0 (1,0) 0 0 (1,1) (1,2) (1,3)
C2 (2,0) (2,1) (2,2) 0 0 (2,3) 0 0
C3 0 (3,0) 0 (3,1) (3,2) 0 0 (3,3)
Zaddr = (0,0) (0,1) (0,2) (0,3) (1,0) (1,1) (1,2) (1,3) (2,0) ...

Fig. 6. One-dimensional address generation for the LLR values of each bit node and check node

Fig. 6 shows the generated address array for the bit node and the check node. The position of each LLR value for check nodes is stored with the form of (x, y), where x is the position of a bit node, and y indicates the order of the same bit node. This position information is rearranged as a one-dimensional array L_{addr} as in (1), where W_B is the degree of bit nodes.

$$L_{addr} = x \times W_B + y \quad (0 \leq y \leq W_B - 1) \tag{1}$$

Z_{addr}, the one-dimensional address array for bit nodes, is computed using a similar method to determine the address array for check nodes, as in (2) where W_C is the degree of check nodes.

$$Z_{addr} = x \times W_C + y \quad (0 \leq y \leq W_C - 1) \tag{2}$$

By using this address arrangement, the number of memory accesses to read the position of the LLR values is reduced in the INIT, CNP, BNP and PC operations [10].

In these address generation steps, positions of LLR values are simply read from the given H-matrix, and the one-dimensional array for each node is constructed independently. Thus, there is no memory access dependency between the two generation tasks. Therefore, the check node address generation (CAG) and the bit node address generation (BAG) can run concurrently on the target multi-core CPU to improve the decoding speed. **Fig. 7** shows the proposed OpenCL task-parallel programming model for the CAG and the BAG operations. In the programming model, PEs execute work-items for each kernel in parallel. In the target OpenCL platform model, a core of the CPU is defined as a PE, and each PE executes a thread independently of the others. The CPU uses an out-of-order command-queue to coordinate the parallel execution of the proposed LDPC kernels. Using the out-of-order queue, queued tasks are executed as soon as idle PEs are available regardless of the input command order.

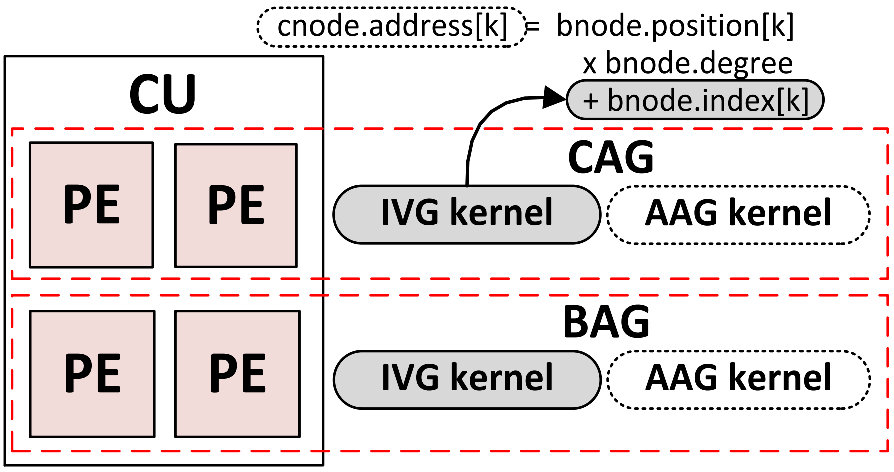


Fig. 7. Task parallel programming model for the check node address generation (CAG) and bit node address generation (BAG) kernels

Each address generation step consists of two mutually dependent kernels: the index value generation (IVG) kernel and the address array generation (AAG) kernel. The IVG kernel finds the index value *y* in the given H-matrix, which indicates the order of the corresponding nodes. The AAG kernel calculates the address array for each node using the index values calculated in the IVG kernel. Therefore, synchronization between IVG and AAG kernels is required for correct decoding.

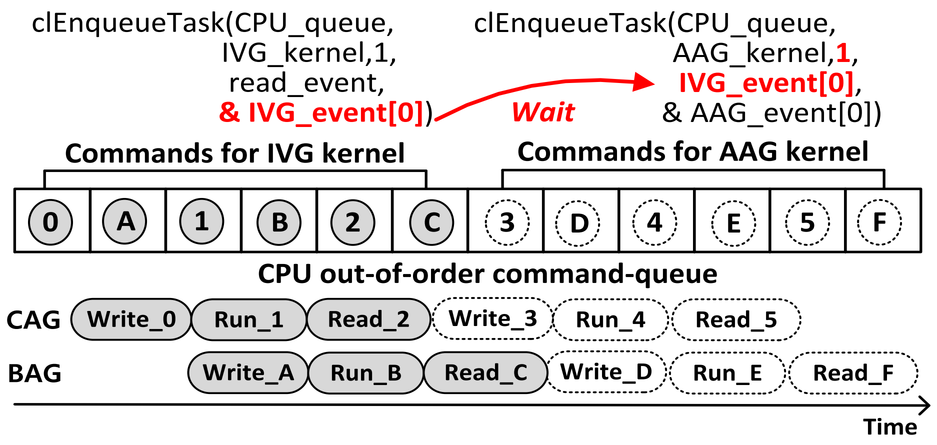


Fig. 8. Synchronization between the data dependent kernels

Command execution of the out-of-order queue depends only on the associated events in the wait list. As shown in **Fig. 8**, each AAG kernel registers unique event objects for an IVG kernel in an event wait list for the *clEnqueueTask* API and waits until all input events have completed the execution. This provides fine-grained control for the out-of-order commands that preserves the execution order between the IVG kernel and the AAG kernel. Therefore, task-level parallel execution of BAG and CAG operations can be conducted successfully.

4.2 Design of SPA-based LDPC Decoding Kernels

Table 2. Pseudo-code for the proposed SPA-based LDPC decoding kernels

Decoding Step	Algorithm
INIT	//BS_B **: Block size of bit node**
	$xIndex = work\_group\_ID \times W_B \times BS_B + local\_ID \times W_B$
	$Index = work\_group\_ID \times BS_B + local\_ID$
	for $i < W_B$ //**For each bit node**
	$Z[Z_{addr}[xIndex + i]] = F[Index]$
	end for
CNP	//BS_C **: Block size of check node**
	$xIndex = work\_group\_ID \times W_C \times BS_C + local\_ID \times W_C$
	for $i < W_C$ //**For each check node**
	$L[L_{addr}[xIndex + i]] = Message[xIndex+i]$
	end for
BNP	$xIndex = work\_group\_ID \times W_B \times BS_B + local\_ID \times W_B$
	$Index = work\_group\_ID \times BS_B + local\_ID$
	for $i < W_B$//**For each bit node**
	$Z[Z_{addr}[xIndex + i]] = Message[Z_{addr}[xIndex + i]]$
	if $Z \geq 0$
	$Decode[Index] = 0$
	else $Decode[Index] = 1$
	endif
	end for
PC	$xIndex = work\_group\_ID \times W_C \times BS_C + local\_ID \times W_C$
	$Index = work\_group\_ID \times BS_C + local\_ID$
	for $i < W_C$ //**For each check node**
	$Check \mathrel{+}= Decode[\text{int}(L_{addr}[xIndex + i]/W_B)]$
	end for
	$Check = int\ (Check\%2)$

Table 2 shows a pseudo-code for the proposed SPA-based LDPC decoding kernels that parallelized on the target GPU. The INIT step is carried out with a pre-generated Z_{addr} that indicates the destination of copied LLR values. The CNP operation then reads the degree of check nodes from the memory and updates the decision values based on the delivered messages. The results are stored in the same region using L_{addr}. The BNP step conducts similar operations for bit nodes using Z_{addr}. Finally, the PC operation examines the check nodes using L_{addr} to determine whether all the decoded values are 0.

In the proposed kernels, the numbers of work-items executing the same instruction for each cycle are denoted as BS_B for bit node operations and BS_C for check node operations, and they are configurable using the index size argument of the *clEnqueueNDRangeKernel* runtime API.

As discussed above, the OpenCL data-parallel programming model operates by concurrently executing work-items on multiple PEs. Even though all work-items execute the same kernel code, the data to be processed can be assigned differently to each work-item by changing the allocated index. Work-items are uniquely identified with the combination of the work-group index and the local index. The local index is the position of the work-item inside of a work-group [2].

As shown in **Fig. 9**, explicit vectorization was applied to GPU kernels. The OpenCL compiler generates SIMD instructions when built-in vector types are used in the kernels. In general, GPUs are optimized for 128-bit data transfer operations per SIMD lane to support four 32-bit pixel vectors [12]. Therefore, vectorization with the four-wide vector type which combines four scalar data into a single vector enables work-items to perform decoding operations on multiple data simultaneously to improve performance. Furthermore, explicit use of vector data types enables more coalesced memory operations and data transfers with a higher bandwidth.

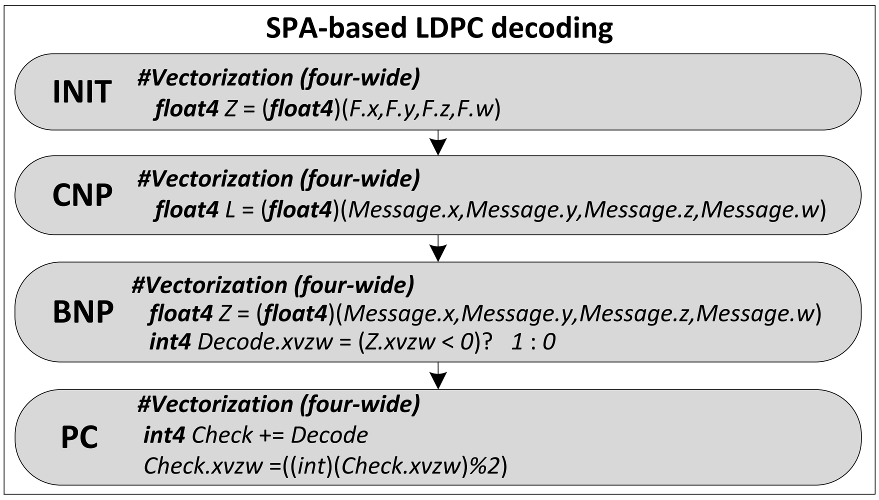

SPA-based LDPC decoding
INIT
#Vectorization (four-wide)
float4 Z = (float4)(F.x,F.y,F.z,F.w)
CNP
#Vectorization (four-wide)
float4 L = (float4)(Message.x,Message.y,Message.z,Message.w)
BNP
#Vectorization (four-wide)
float4 Z = (float4)(Message.x,Message.y,Message.z,Message.w)
int4 Decode.xvzw = (Z.xvzw < 0)? 1 : 0
PC
#Vectorization (four-wide)
int4 Check += Decode
Check.xvzw =((int)(Check.xvzw)%2)

Fig. 9. Explicit vectorization for SPA-based LDPC decoding kernels

As mentioned above, GPUs have a local memory region to store local data for a CU, and its access time is much faster than that of the global memory. Therefore, efficient use of the local memory region can improve performance of the GPU kernels by providing fast data access and efficient data sharing among hundreds of work-items in a work-group. In the proposed GPU kernels, the one-dimensional address array for each node is used by every work-item without having to repeatedly compute a new address. Therefore, the proposed GPU kernels allocate the address arrays in the local memory to hold position values that can be shared by all work-items that belong to the same work-group.

As shown in **Fig. 10**, the INIT and BNP kernels use Z_{addr} and the CNP and PC kernels use L_{addr} in the local memory. However, the local memory state is not guaranteed to be consistent across work-items inside the group because the data transmission time of each work-item is different. Therefore, synchronization using a barrier is applied to the GPU kernels. A barrier is a built-in function that prevents work-items from crossing it until all work-items in a work-group have reached it [12]. Global synchronization among kernels is managed by event objects for non-blocking commands in the in-order GPU command-queue. Detailed experimental results are addressed in Section 5.

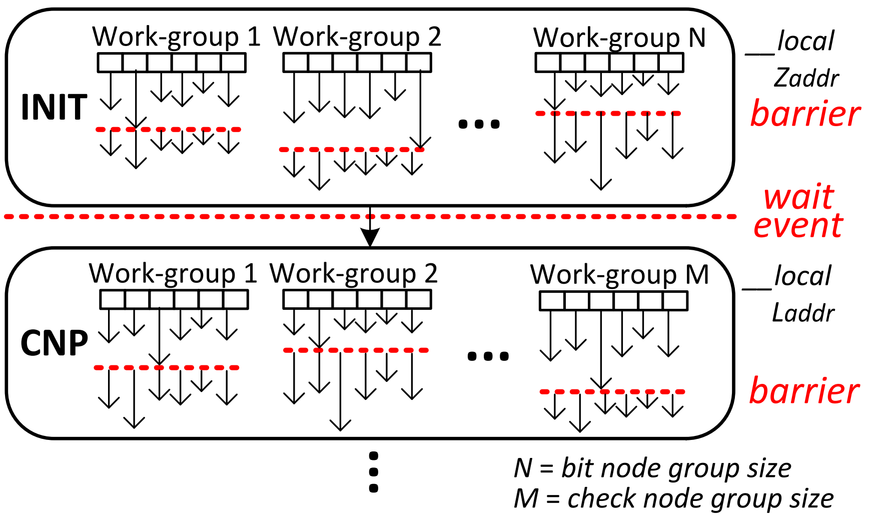


Fig. 10. Local memory allocation and synchronization for proposed GPU kernels

4.3 Zero-copy Technique for the Proposed LDPC Decoder

The host program asks a device to execute the assigned kernel using OpenCL runtime APIs. To execute a kernel, the data to be processed must be copied to the device memory because the host cannot directly access the memory system of the discrete CDs. After processing the data on the kernel, the computation results are transferred back to the host memory. Generally, OpenCL devices carry these operations out using the *clEnqueueRead* and *clEnqueueWrite* APIs. These explicit data transfers may suffer from large latency because the request must be handled by relatively slow inter-device busses.

However, as mentioned in Section 2, the global memory of the target mobile platform is shared by the CPU and the GPU. A significant benefit of this heterogeneous system is that no memory object copy is necessary since the physical memory is unified. Furthermore, the CPU in the proposed platform model is defined as a CD as well as the host, and therefore, any explicit data transfer is unnecessary. Thus, the zero-copy technique imposing little execution overhead is applied to the entire CPU and GPU kernels.

Fig. 11 shows how the zero-copy is carried out by the *clEnqueueMapBuffer* and *clEnqueueUnmapMemObject* APIs in place of explicit data transfer functions. These APIs just map a buffer region into the shared address space and return a pointer of the mapped region. The memory objects created with the *CL_MEM_ALLOC_HOST_PTR* flag can be accessed directly by both the host and a CD. These memory objects are called pinned zero-copy buffers, and they can be used by OpenCL applications as function arguments without any data copying. Using pinned zero-copy buffers prevents the allocated memory from being paged out accidentally by the operating system and provides an improved transfer speed by achieving near the peak interconnect bandwidth [13].

As shown in **Fig. 12**, the zero-copy technique significantly reduces the execution time of the entire LDPC decoding process by eliminating runtime data transfers during kernel execution. When the host finishes a logical memory transfer to the zero-copy buffer with the *map* function, the *unmap* function makes this memory object available to the target devices. After processing the data by the GPU kernels, the host can directly access results using the zero-copy technique without any additional copying. Furthermore, this is done without any additional memory allocation because both CDs are defined in the same context, and memory objects are associated with the context rather than the device. As the context is shared among CDs, global synchronization between the CPU and the GPU kernels becomes feasible using event objects. The completion of events for CPU commands is guaranteed that the generated address arrays for each node can be accessed directly by GPU kernels. The following section reports performance comparison results when the proposed zero-copy techniques were applied to the decoding kernels.

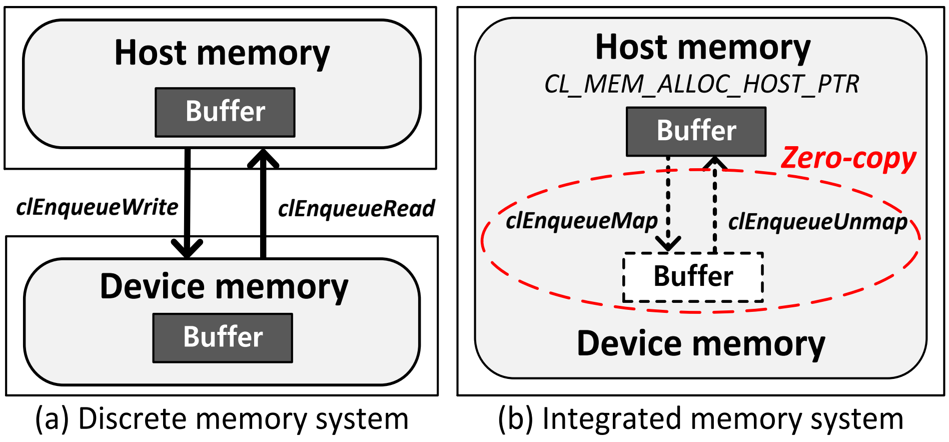


Fig. 11. Zero-copy technique for the integrated memory system

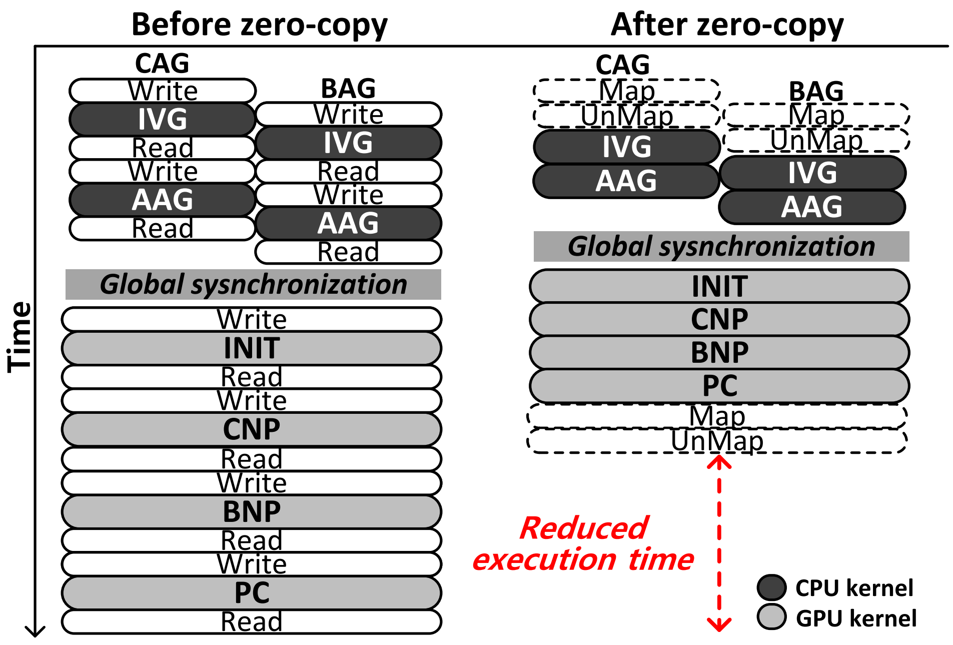


Fig. 12. Reduced execution time after applying the zero-copy technique

5. Experimental Results

This article proposes a novel decoder for (3, 6) regular LDPC codes with a word length of 9216 bits to satisfy the throughput requirement of the CMMB, China's designated mobile TV standard. Its H-matrix is known to have high error-correcting capability [5]. The CMMB standard supports two code rates, 1/2 and 3/4; specifications for each code rate are summarized in **Table 3**.

Table 3. LDPC coding configuration

Code rate R	**Information K**	**Code word N**	**Row weight W_C**	**Column weight W_B**
1/2	4608 bits	9216 bits	6	3
3/4	6912 bits	9216 bits	12	3

To evaluate the performance of the proposed LDPC decoder on a heterogeneous platform, kernels were compiled with an Intel software development kit (SDK) for OpenCL version 1.2 which supports the out-of-order command-queue for the target CPU. Microsoft Visual Studio 2010 with 64-bit Windows 7 Service pack 1 was used as the host C program compiler. The target heterogeneous mobile platform consists of an Intel i7-4720HQ quad-core CPU and an Intel HD 4600 integrated GPU with 8GB of DDR3 random access memory (RAM). The CPU was defined as a single CU including four PEs, and the GPU consisted of 7 CUs where each CU contained 32 PEs. A unified memory hierarchy for both CPU and GPU cores was defined as the OpenCL global memory. Performance of the proposed LDPC decoder was evaluated with respect to various signal-to-noise ratio (SNR) values on the target platform.

5.1 Performance Evaluation for Proposed CPU Kernels

The aforementioned task-level parallelization (TP) and the zero-copy technique were applied to the one-dimensional address generation and they were executed on the CPU. In this experiment, execution times for the proposed CPU kernels were compared with three other decoders: a decoder written in the C language, a decoder parallelized with OpenMP, and an OpenCL implementation that was not based on the proposed techniques (hereafter called as the base OpenCL implementation). OpenMP is a pragma-based parallelization API for multi-core CPUs [14]. In this experiment, OpenMP pragmas were inserted to parallelize the address generation on a quad-core CPU using four threads.

Table 4. Average address array generation processing time/frame

	C	**OpenMP**	**OpenCL (CPU Kernel)**		
			Normal	**After TP**	**After Zero-copy**
CMMB, code rate: 1/2, frame: 100,000, SNR: 1					
(ms)	2.564	1.563	3.041	1.641	1.295
Speedup	1.97	1.21	2.34	1.26	-
CMMB, code rate: 3/4, frame: 100,000, SNR: 1					
(ms)	4.245	2.522	4.890	2.577	2.047
Speedup	2.12	1.23	2.40	1.25	-

Table 4 summarizes processing times for the one-dimensional address array generation step for each CMMB code rate. Due to runtime overhead caused by OpenCL object management, the base OpenCL implementation takes longer than the standard C and the OpenMP implementations. However, when the TP technique was applied, the performance gap between the OpenCL and the OpenMP implementations was reduced. Performance of the entire address generation using the OpenCL framework was much improved when the zero-copy

technique was applied.

As shown in Fig. 13, the average data transfer time of the OpenCL implementations was much longer than the memory allocation time of the C and the OpenMP designs. To reduce the runtime of data transfers and unnecessary memory duplications caused by the mismatches between the platform model and the CPU architecture, the zero-copy technique was applied to the CPU kernels, and the data transfer time was decreased by almost 60%. As a result, the address generation processing achieved a speed-up of up to 2.12 over the case in which no parallelization was applied to the application written in C.

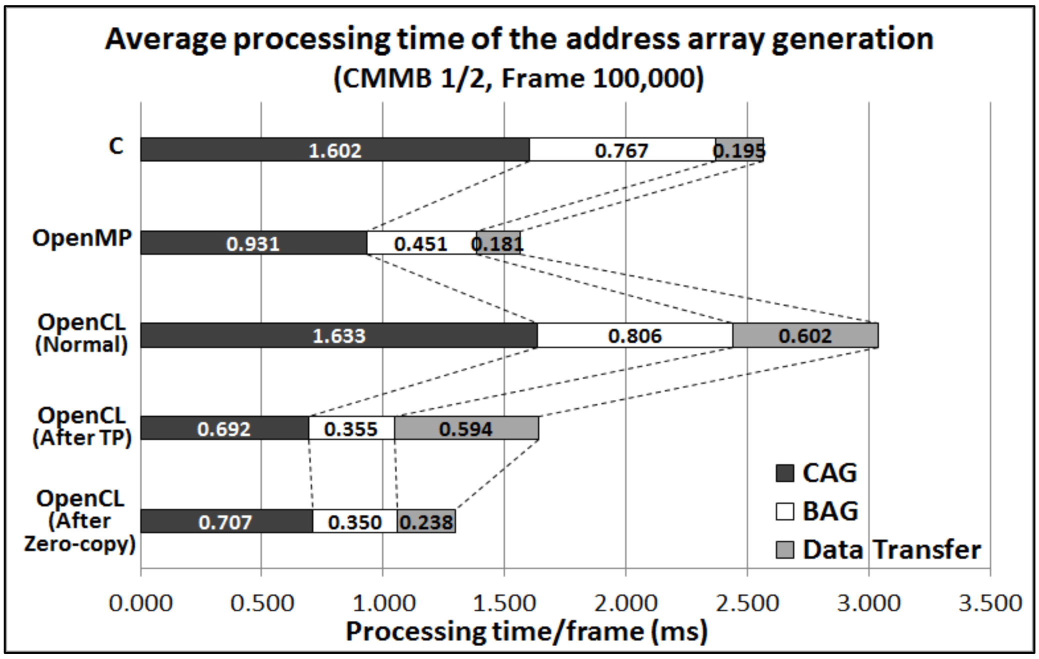


Fig. 13. Average processing time of one-dimensional address generation on a CPU

5.2 Performance Evaluation for Proposed GPU Kernels

As described in Section 2.2, the number of work-items specified by a programmer is the same as the number of active threads that execute the proposed LDPC kernels in parallel. The best size of the thread block composing a work-group was determined to hide memory access latencies and use the GPU hardware resources wisely. Fig. 14 shows the average decoding time of the proposed GPU kernels for 100,000 frames with a code rate of 1/2. The execution time was most improved when each kernel was processed in parallel with 32 threads, largely because the number of PEs defined in the proposed execution model is 32, which was determined by considering the maximum number of the target GPU's hardware threads.

Table 5 compares the performance of the proposed OpenCL design with those of the other implementations for each code rate. Every OpenCL design ran the same GPU kernels on 32 PEs to compare the CPU performance under the same conditions. In this article, explicit vectorization using four-wide vector types were applied to each GPU kernel because the internal memory paths on the target GPU support 128-bit data transfers. The effective data transfer techniques including zero-copy and local memory allocation were applied as well to maximize hardware resource utilization. When the utilization techniques were applied, the proposed LDPC decoder achieved a speedup of up to 17.6 over the C design and 2.14 over the base OpenCL design. To satisfy the CMMB performance requirements, decoding for one frame must be processed within 3.3 milliseconds [10]. The proposed decoder satisfies this requirement with signal reception of at least 1 dB for a 1/2 code rate and 3 dB for a 3/4 code rate.

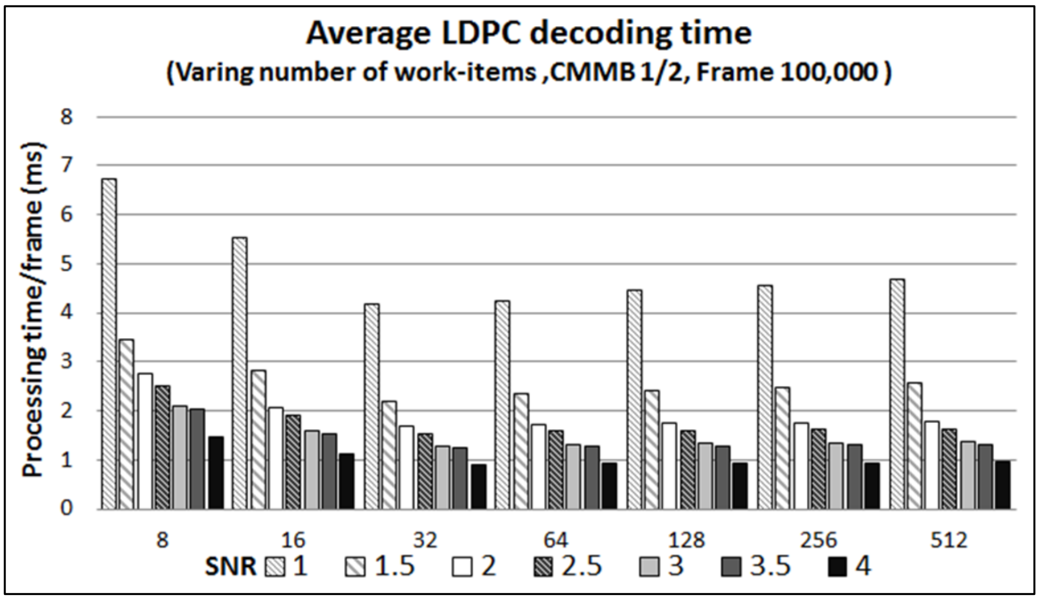


Fig. 14. Average processing time of LDPC decoding kernels on a GPU with varying numbers of work-items

Table 5. Average LDPC decoding time per frame versus SNR value

SNR	C	OpenMP	OpenCL (GPU, Work-item: 32)	
			Normal	After Utilization
CMMB, code rate: 1/2, frame: 100,000, millisecond				
1	37.62	9.40	5.06	2.32
1.5	15.60	6.30	2.10	1.04
2	10.06	3.08	1.19	0.69
2.5	7.39	1.84	1.00	0.56
3	3.93	1.25	0.92	0.38
3.5	3.01	0.90	0.86	0.33
4	2.21	0.78	0.67	0.29
CMMB, code rate: 3/4, frame: 100,000, millisecond				
2.5	140.84	39.91	17.15	7.99
3	45.29	15.66	5.96	2.26
3.5	38.85	13.29	5.26	1.75
4	25.78	12.05	4.44	1.39
4.5	21.69	8.03	2.97	1.21
5	16.62	5.87	2.37	0.88
5.5	14.50	5.16	2.24	0.75

5.3 Performance Evaluation of the Entire LDPC Decoder

As described in Section 4, the proposed design is a cost effective and flexible software LDPC decoder that can support multiple standards. To support multiple standards, multiple H-matrices are stored as files and the host CPU reads the H-matrix for a given standard and generates an address table of the LLR values. To show that our method is satisfactory for multiple standards, an LDPC decoder for IEEE 802.11n with a code rate of 1/2 is implemented. 802.11n is a local area network (LAN) standard, and the H-matrix has 1944 bits of code words and 972 bits of an information word. In general, the entire decoding time of the 802.11n takes longer than CMMB because it is an irregular code that the parity check matrix contains a different number of 1's in each row and each column [15].

Table 6. Processing time of LDPC decoding versus SNR value

SNR	CPU only			CPU+GPU (PCI-e)		CPU+GPU (SoC)
	C	OpenMP	Cilk Plus	OpenMP+CUDA	OpenCL	OpenCL
CMMB, code rate: 1/2, frame: 100,000, second						
1	312.73	101.24	96.68	39.14	34.21	44.36
1.5	157.21	55.62	51.21	20.35	17.79	23.07
2	103.21	33.78	31.02	13.03	11.39	14.76
2.5	52.46	18.63	17.24	6.77	5.92	7.68
3	28.17	9.12	8.40	3.52	3.08	4.09
3.5	13.58	4.93	4.25	1.83	1.6	2.12
4	6.52	2.57	2.11	0.95	0.83	1.18
802.11n, code rate: 1/2, frame: 100,000, second						
1	15.90	9.21	8.89	3.12	2.87	4.21
1.2	28.00	19.80	17.12	8.22	7.32	8.90
1.4	69.80	29.32	26.87	15.31	14.87	16.81
1.6	272.60	123.59	114.94	54.12	52.41	61.12
1.8	1114.10	612.22	589.77	385.13	370.41	391.91
2.0	956.2	502.92	484.42	244.12	238.41	257.20
2.2	838.9	452.34	440.06	210.89	190.54	227.52
2.4	752.4	426.01	404.41	181.31	174.32	207.51
2.6	684.3	387.41	368.67	165.41	150.32	191.28

Table 6 shows the entire LDPC decoding time of six different implementations for CMMB and 802.11n standards with respect to various SNR values. The first three are C, OpenMP, and Intel Cilk Plus implementations that run only on the CPU. The fourth is a recently proposed design that was mentioned in Section 3.2 using OpenMP and CUDA [10]. The fifth and sixth approaches are the proposed OpenCL implementations on a heterogeneous mobile platform. The difference of the two is whether an external GPU is employed or an integrated one is employed. A NVIDIA GeForce GTX 860M, which was used for the external GPU of the third and the fourth cases, consisted of 16 CUs where each CU contained 40 PEs. And the external GPU is connected to the CPU via the peripheral component interconnect-express (PCI-e) bus.

Intel Cilk Plus is a simple language extension to C and C++ to express multi-threaded parallelism for the Intel multi-core CPUs [16]. It introduces three new keywords in the programming language: *cilk_for*, *clik_spawn*, and *cilk_sync*, while OpenMP uses directive based programming models. In this section, all CPU versions have been compiled with Intel C++ compiler 16.0 that offers stable support for both OpenMP and Cilk Plus. As shown in

Table 6, both CMMB and 802.11n software LDPC decoders parallelized with Cilk Plus show the best performance among the CPU only versions because Cilk Plus supports optimized thread scheduling and vectorization using streaming SIMD extensions (SSE) for modern Intel processors.

However, performance gain of the CPU only implementations is limited compared to the parallel LDPC decoder using OpenCL due to they rely on the vendor-specific computing framework. The parallel LDPC decoding on the heterogeneous platform using OpenCL shows better performance than the other implementations. Table 6 shows that for SNR of 1, we have achieved a speedup of 2.18 for CMMB and that of 2.11 for 802.11n compared to the Cilk Plus implementations. These results show that performance of the proposed decoder is excellent, mainly due to efficient hardware resource utilization with a unified computing framework for the CPU and the GPU.

Table 7. Performance per watt of LDPC decoding

	CPU (OpenMP)	**CPU (Cilk Plus)**	**CPU+GPU (OpenCL, PCI-e)**	**CPU+GPU (OpenCL, SoC)**
CMMB, code rate: 1/2, frame:100,000, SNR: 1				
Power consumption (watt)	11.97	14.86	55.26	22.63
Frame per second/watt	81.52	69.61	52.89	99.75
802.11n, code rate: 1/2, frame: 100,000, SNR: 1				
Power consumption (watt)	16.12	19.86	59.23	28.23
Frame per second/watt	786.25	566.39	588.27	843.41

Although the PCI-e bus significantly limits the data transfer rate between the host and CDs, decoding time of the external GPU is faster than the integrated GPU because the external GPU has much more PEs that execute more work-items in parallel. However, speed and energy efficiency are equally important for consumer electronic devices. Thus, performance per watt of the LDPC decoding was evaluated. Table 7 compares the performance per watt of the CPU only implementations and the OpenCL implementations for CMMB and 802.11n standards with SNR of 1. Power consumption of each implementation has been measured using real-time power monitoring tools for the mobile CPU and the mobile GPU [17]. These results verify that performance per watt of the proposed decoder on a mobile platform is excellent, mainly due to the proposed hardware utilization techniques for the shared memory architecture.

Hardware decoder implementations of LDPC codes show better performance per watt compared to the proposed OpenCL implementations because power consumptions of hardware decoders have ranged merely from 87 mW to 173.5 mW [18]. However, it is very challenging to design a hardware LDPC decoder that supports various standards and multiple data rates. The proposed parallel software LDPC decoder on a mobile system using an OpenCL framework satisfies the performance requirement of the multiple LDPC standards while providing scalability, high portability, and flexibility, which may not be possible with hardware implementations.

6. Conclusion

When digital signal processing and communication applications are parallelized on an OpenCL framework, they can be executed on heterogeneous devices such as CPUs and GPUs to achieve both high portability and high performance. This article proposed parallelization methods for the LDPC decoder on an embedded SoC platform which included both a multi-core CPU and a multi-core GPU. Efficient computing resource management for heterogeneous devices was achieved using OpenCL objects defined under a unified single context. To improve performance of the proposed decoding kernels, explicit thread scheduling, vectorization, and effective data transfer techniques were applied. The proposed LDPC decoder satisfied the performance requirements of the CMMB standard and showed high performance per watt by intelligently utilizing both the CPU and the GPU with OpenCL. Future works are going on to parallelize the LDPC decoder on various heterogeneous computing platforms, including one with field-programmable gate arrays (FPGAs), using OpenCL.

References

[1] J.-H. Hong, Y.-H. Ahn, B.-J. Kim, and K.-S. Chung, “Design of OpenCL Framework for Embedded Multi-core Processors,” *IEEE Trans. Consumer Electron*., vol. 60, no. 2, pp. 233-241, May, 2014. Article (CrossRef Link).

[2] Khronos OpenCL Working Group, *The OpenCL Specification Version 1.2*, Document Revision 19, 2012. Article (CrossRef Link).

[3] R.G. Gallager, “Low-density parity check codes,” *IRE Trans. Information Theory*, vol. 8, no. 1, pp. 21-28, Jan. 1962. Article (CrossRef Link).

[4] E.-S. Jeon *et al*., "Iterative detection and ICI cancellation for MISO-mode DVB-T2 system with dual carrier frequency offsets," *KSII Transactions on Internet and Information Systems*, vol. 6, no. 2, pp. 702-721, Feb. 2012. Article (CrossRef Link).

[5] K. Zhang, X. Huang, and Z. Wang, “A Dual-Rate LDPC Decoder for China Multimedia Mobile Broadcasting Systems,” *IEEE Trans. Consumer Electron*., vol. 56, no. 2, pp. 399-407, May, 2010. Article (CrossRef Link).

[6] G. Wang, M. Wu, B. Yin, and J. R. Cavallaro, “High throughput low latency LDPC decoding on GPU for SDR systems,” in *Proc. of 2013 IEEE Global Conference on Signal Processing*, pp.1258-1261, Dec. 2013. Article (CrossRef Link).

[7] S. Wang, S. Cheng, and Q. Wu, “A parallel decoding algorithm of LDPC codes using CUDA,” in *Proc. of 2008 42nd Asilomar Conference on Signals, Systems and Computers*, pp. 171-175, Oct. 2008. Article (CrossRef Link).

[8] B. Gal and C. Jego, “High-throughput LDPC decoder on low-power embedded processors,” *IEEE Communication Letters*, vol. 1, no. 99, pp. 1-4, Sep. 2015. Article (CrossRef Link).

[9] G. Falcão, V. Silva , L. Sousa, and J. Andrade, “Portable LDPC decoding on multicores using OpenCL,” *IEEE Signal Processing Magazine*, vol. 29, no. 4, pp. 81-109, July, 2012. Article (CrossRef Link).

[10] J.-Y. Park and K.-S. Chung, “Parallel LDPC decoding using CUDA and OpenMP,” *EURASIP Journal on Wireless Communications and Networking*, vol. 2011, no. 1, Dec. 2011. Article (CrossRef Link).

[11] J.-H. Hong, W.-J. Kim, and K.-S. Chung, “A Parallelization Technique with Integrated Multi-Threading for Video Decoding on Multi-core Systems,” *KSII Transactions on Internet and Information Systems*, vol. 7, no. 10, pp. 2479-2496, Oct. 2013. Article (CrossRef Link).

[12] B. R. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schadd, *Heterogeneous computing with OpenCL: Revised OpenCL 1.2 Edition*, Morgan Kaufmann, 2012. Article (CrossRef Link).

[13] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu, "Performance Traps in OpenCL for CPUs," in *Proc. of 21st Euro Micro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 38-45, Feb. 2013. Article (CrossRef Link).
[14] D. Leonardo and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science & Engineering*, vol. 5, no.1, pp. 46-55, Jan. 1998. Article (CrossRef Link).
[15] K. Gunnam, G. Choi, W. Wang, and M. Yeary, "Multi-Rate Layered Decoder Architecture for Block LDPC Codes of the IEEE 802.11n Wireless Standard," in *Proc. of 2007 IEEE International Symposium on Circuits and Systems*, pp. 1645-1648, May, 2007. Article (CrossRef Link).
[16] Intel Corporation, *Intel Cilk Plus Language Specification*, Document Revision 1.0, 2010. Article (CrossRef Link).
[17] G. Sheng, Z. Yong, S. Zhukai, and S. Fan, "Optimize power for portable games on Ultrabook," in *Proc. of 2012 IEEE International Conference on Energy Aware Computing*, pp. 1-6, Dec. 2012. Article (CrossRef Link).
[18] J. Li, J. Ma, and G. He, "A memory efficient parallel layered QC-LDPC decoder for CMMB systems," *Integration, the VLSI Journal*, vol. 46, no. 4, pp. 359-368, Sep. 2013. Article (CrossRef Link).

Jung-Hyun Hong received his B.S. degree in Media Communication Engineering from Hanyang University, Seoul, Korea, in 2011. Since 2011, he has undertaken a unified M.S. and Ph.D. course at Hanyang University, Seoul, Korea. His research interests include software parallelization, heterogeneous computing, and embedded multi-core architecture.

Joo-Yul Park received his B.S. in Electronic Engineering from Ajou University, Suwon, Korea, in 2004, and his Ph.D. in Electronics, Computer, and Communication from Hanyang University, Seoul, Korea, in 2013. He was an engineer at LG Electronics Corp. in Seoul from 2005 to 2007. Since 2013, he has been an Adjunct Professor at Hanyang University, Seoul, Korea. His research interests include reconfigurable processor and DSP design, channel coding, and system software for MPSoC.

Ki-Seok Chung received his B.E. degree in Computer Engineering from Seoul National University, Seoul, Korea, in 1989 and his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 1998. He was a Senior R&D Engineer at Synopsys, Inc. in Mountain View, CA, from 1998 to 2000 and was a Staff Engineer at Intel Corp. in Santa Clara, CA, from 2000 to 2001. He also worked as an Assistant Professor at Hongik University, Seoul, Korea, from 2001 to 2004. Currently, he is a Professor at Hanyang University, Seoul, Korea. His research interests include low-power embedded system design, multi-core architecture, image processing, reconfigurable processor and DSP design, SoC-platform-based verification, and system software for MPSoC.