JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 16, No. 2, June 2022, pp. 105-112

Regularized Convolutional Neural Network for Highly Effective Parallel Processing

Sang-Soo Park and Ki-Seok Chung*
Department of Electronic Engineering, Hanyang University, Seoul, Korea
po092000@hanyang.ac.kr, kchung@hanyang.ac.kr

Abstract
Convolutional neural network (CNN) has been adopted in various areas. Using graphics processing unit (GPU), speed improvement can be achieved on CNN, and many studies have proposed such acceleration methods. However, parallelizing the CNN on GPU is not straightforward because there are irregular characteristics in generating output feature maps.in typical CNN models. In this paper, we propose a method that maximizes the utilization of GPU by modifying convolution combinations of a well-known CNN network, LeNet-5. Our regularized implementation on a heterogeneous system has achieved an improvement of up to 37.26 times in convolution and sub-sampling layers. Further, an energy consumption reduction of up to 26.40 times is achieved.

Category: Cloud Computing / High-Performance Computing

Keywords: Heterogenous system; GPGPU; Parallel processing; OCR; Diverse branch

I. INTRODUCTION

Convolutional neural network (CNN) is a biologically inspired model, which mimics the activity of neurons in the human brain. CNN has been adopted in various areas, which include image, speech, and natural language processing [1, 2].

To achieve high classification performance, CNN normally employs deep processing layers, and therefore, the computation amount of CNN is huge [3]. However, the convolution operation using floating-point numbers may be accelerated by graphics processing unit (GPU) parallel processing. Especially, parallel processing of CNN on a heterogeneous system where a CPU and a GPU are integrated into a single chip is advantageous because they share the physical memory space so that the data transfer overhead is much less than on a platform where a CPU and a GPU are externally combined.

In this paper, we modify LeNet-5 [4], which is widely used in the optical character recognition (OCR) field, to fit with a general-purpose computing on GPU (GPGPU) architecture. Convolution operations of convolution layer 2 in LeNet-5 have irregular characteristics, and it is almost impossible to assign a kernel to a work-group/thread-block with the same number of work-items/threads. Thread group scheduling and task allocation are critical for taking full advantage of the GPU resource with single instruction multiple threads (SIMTs). Therefore, we propose a new operation called "dummy operation" to properly distribute convolution operations to a GPU thread group. The performance of the proposed method is evaluated by two different programming environments: Open Computing Language (OpenCL) [5] and Compute Unified Device Architecture (CUDA) [6]. The proposed

Open Access http://dx.doi.org/10.5626/JCSE.2022.16.2.105 http://jcse.kiise.org

Received 28 April 2022; **Accepted** 08 June 2022
*Corresponding Author

 pISSN: 1976-4677 eISSN: 2093-8020

convolution structure achieves feature extraction with a speed-up of 37.26 times in OpenCL and 21.18 times in CUDA by efficiently utilizing the GPU resource with reasonable power dissipation.

The rest of this paper is organized as follows; Section II introduces CNN. Section III presents an introduction to GPU programming with OpenCL and CUDA. Section IV describes the proposed dummy operation and explains the effectiveness of the dummy operation on GPU. Section V shows our experiment results. Section VI concludes this paper.

II. CONVOLUTIONAL NEURAL NETWORK

Today, CNN-based object recognition systems can recognize objects with super-human accuracy [7]. CNN is composed of three different types of layers: convolution layers, sub-sampling layers, and fully-connected layers. These layers are arranged in a feed-forward structure. The group of convolution layers and sub-sampling layers is used for feature extraction and the group of fully-connected layers is used for classification. Using convolution, sub-sampling, and fully-connected layers, CNN can achieve highly accurate classification performance.

LeNet-5 is the CNN model that we modify to improve the inference performance on a CPU-GPU platform. It is configured with three convolution layers, two sub-sampling layers, and two fully-connected layers. LeNet-5 extracts output results called feature maps from one 32×32 input image through multiple convolutions and sub-sampling layers. Finally, the extracted 120 feature maps of the 1×1 resolution are passed through the fully-connected layers to get the final classification result, which is a number from 0 to 9.

A. Convolution Layer

Element-wise multiplications between an input feature map and a convolution kernel are carried out in the convolution layer.

First, the input feature map is convolved with a convolution kernel. Second, the sum of weighted results from the first stage and the bias is calculated. Finally, the sum result is filtered with an activation function such as sigmoid, tanh, and ReLu. If an $N \times N$ input feature map and an $m \times m$ convolution kernel are used in a convolution layer, the output feature map is determined as (N-m+1)×(N-m+1). The process in the overall convolution layer may be summarized as Eq. (1).

$$out_{t(x,y)} = f'\left(\sum_{i=0,\ j=0}^{m-1,\ n-1} W_{t(i,j)} * In_{(x+i,y+i)} + b\right) \quad (1)$$

where each component has a two-dimensional coordinate (x, y). *In*, *out*, and b are the input feature map, output feature map, and bias, respectively. W indicates a convolution kernel while i, j, and t denote the width, the height, and the type indices of the convolution kernel, respectively. The activation function is denoted by f'.

B. Sub-sampling Layer

Following the execution of a convolution layer, the sub-sampling layer reduces the size of feature maps from the previous layer. This layer is frequently used in CNN to gradually reduce the spatial size of features and the computational complexity of the network by reducing several adjacent neurons of feature maps in the previous layer.

C. Fully-Connected Layer

After the feature extraction obtained from processing multiple convolutions and sub-sampling layers, the fully-connected layer follows. The term "fully-connected" means that all neurons in the current layer are connected to all neurons in the next layer.

While the output feature map from convolutions and sub-sampling layers represents high-level features of the input image, the output of the fully-connected layer is the classification result.

III. OPENCL AND CUDA

Deep learning acceleration by using multiple GPUs is actively attempted. In general, GPUs have much more powerful computational capability for data-intensive applications than CPUs, and they can carry out massive data-parallel computations effectively.

NVIDIA's CUDA platform is widely used, and a number of deep learning tools utilize this platform for learning and inference acceleration [6, 8]. Also, OpenCL has gained a lot of attention as a programming framework for heterogeneous system architectures [5].

A. OpenCL

The OpenCL framework includes a programming language, application programming interfaces (APIs), and libraries to support software development. The OpenCL specification is defined in a hierarchy of models: platform model, execution model, and memory model.

1) Platform Model

The OpenCL platform consists of one CPU host and one or more accelerators called compute device (CD). Typically, CDs include field-programmable gate arrays (FPGA), GPUs, and digital signal processors (DSP). A CD is divided into several compute units (CUs) and a CU is further divided into one or more processing elements (PEs). At the lowest level, an OpenCL C/C++ function

called kernel is executed in parallel by each PE.

2) Execution Model

The execution model defines how an OpenCL kernel runs on PEs. At the lowest level, an OpenCL kernel is executed in parallel in a pre-defined computational index space called NDRange. In NDRange, each independent element is called a work-item. Work-items are grouped into a work-group. Work-groups are grouped and assigned to a CU that contains multiple PEs by which each work-item is executed.

B. CUDA

CUDA is a parallel computing framework developed by NVIDIA. CUDA is a general computing framework that runs only on the NVIDIA GPUs.

1) Hardware Architecture

In CUDA architecture, the GPU is called a device, and the CPU is called the host. A CUDA consists of a cluster of many-core processors. In CUDA, each many-core processor is called streaming multiprocessor (SM), and SM can contain multiple processors called streaming processors (SP).

2) Execution Model

The execution model is based on threads. A thread can be viewed as a function, called a kernel. The concept of the kernel is the same as in OpenCL. Threads are grouped into a block that executes a kernel. Each thread has a two-dimensional index that is unique within its block. Each block has a unique two-dimensional index.

Also, the execution model of CUDA reflects the specific hardware architecture of GPU. A block of threads can be executed by only one SM. But SM can execute multiple blocks simultaneously by time slicing. A group of blocks that executes simultaneously is called Warp. Warps are concurrently executed in a time-slicing manner.

The execution model defines how an OpenCL kernel runs on PEs. At the lowest level, an OpenCL kernel is executed in parallel in a pre-defined computational index space called NDRange. In NDRange, each independent element is called a work-item. Work-items are grouped into a work-group. Work-groups are grouped and assigned to a CU that contains multiple PEs by which each work-item is executed.

C. Programming on a Heterogeneous System

A heterogeneous system that combines CPUs with GPUs has become standard in most embedded systems such as smartphones. In the heterogeneous system, the CPU and GPU share physically the same memory. Since CPUs and GPUs have different architectures, optimization methods for CPUs should be different from those for GPUs. In this paper, we mainly focus on accelerating the inference speed on GPUs.

In our implementation, we used unified memory in CUDA and zero-copy in OpenCL [7, 9]. Both methods eliminate unnecessary copy processes between CPU and GPU using memory pointers. Therefore, there is no need for a memory copy. Especially, in a heterogeneous system that has limited memory size, efficient memory utilization is very important. By preventing keeping multiple copies of the same data, the memory utilization is significantly improved leading to an overall performance improvement.

IV. DUMMY OPERATION FOR REGULAR GPU PARALLELISM

A GPU can execute thousands of threads in parallel. However, LeNet-5's convolution layer 2 (C2) is not adequate for parallelization by GPU as it is. In C2, 16 output feature maps are generated from 6 input feature maps. In generating an output feature map, only a subset of the input feature maps is used. The subset selection is not regular, and therefore, it is not straightforward to parallelize the computation for this layer using multi-threading. Also, the thread scheduling of C2 may suffer from considerable overhead when thread blocks/work-groups are scheduled with different numbers of threads/work-items.

A. Convolution Layers in LeNet-5

Unlike other convolution layers, only a few selected input feature maps are used to generate an output feature map in C2. To generate 16 output feature maps of the resolution of 10×10, C2 uses input feature maps and convolution kernels of variable depths. Fig. 1 shows the subsets of input feature maps to each output feature map. Each column indicates which input feature maps are used for each output feature map. Each 10×10 output feature map is used to generate a 5×5 feature map by sub-sampling layer 1.

As shown in Fig. 1, to generate an output feature map, 3–6 input feature maps are involved. For example, the

Input feature index \ Output feature index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

Fig. 1. Input and output feature map connections of C2.

14th output feature map is the convolution result of the 0th, 2nd, 3rd, and 5th input feature maps with four convolution kernels of the resolution of 5×5. Therefore, 5×5×4 operations will be needed. Also, the 0th output feature map is the convolution result of the 0th, 1st, and 2nd input feature maps with three convolution kernels of the resolution of 5×5 and thus, 5×5×3 operations should be carried out. These irregularities in generating an output feature map make it difficult to be executed in parallel by the SIMT architectures such as GPU. The SIMT architecture can achieve high throughput performance by sacrificing the per-thread logic for control flow to increase the number of execution units [10]. That is, the control flow logic is shared by all the PEs in a CU/SM, and the control divergence per PE in a GPU may impose a significant burden on parallel executions. Specifically, when a subset of threads is executed by one control logic and the other subset of threads is executed by another control logic, the execution of the two subsets should be serialized because their control logics are different. This may significantly degrade the GPU performance due to the low utilization of execution units.

In OpenCL and CUDA, GPU executes a kernel in parallel by assigning the same number of threads/work-items to each thread work-group/block. Each PE/SP in a CU/SM processes one or more work-items/threads, and a work-item/thread in a thread work-group/block is processed by only one PE/SP. The irregularities of generating an output feature map make an overhead of irregularities and cause a divergence branch problem [10]. If threads are branched by conditional statements, the work-item is not run in parallel on PE. It is executed sequentially in each PE and there are many idle state PEs. Therefore, it makes compute kernel to run in serial rather than in parallel thus increasing the number of idle state PEs. Also, if the sizes of thread work-groups/blocks are all different, the overhead to handle different sizes may be significant. Further, irregularly-sized thread work-groups/blocks make it difficult to determine the optimal thread block/work-group size for a specific CD/SM. Therefore, the convolution method in Fig. 1 is not suitable for GPU to execute in parallel.

B. Proposed Dummy Operation Method

The key idea of the proposed method is to introduce a special operation called a dummy operation. The dummy operation performs no meaningful work. However, by inserting dummy operations, it is possible to make the amount of computation even regardless of the type of the output feature map. Thus, C2 can be parallelized with each work-group/thread block having the same number of threads/work-item. The proposed method is shown in Fig. 2.

In this proposed method, the convolution layer uses uniformly 6 14×14 input feature maps and 6 5×5 convolution kernels. The total number of operations is 5×5×6. The 6 convolution kernels are divided into two groups: the original and the extra parts. The original part has values that correspond to the convolution kernels in Fig. 1 and the extra part has all 0s. For example, to make the 0th output feature map, the original part includes the three convolution kernels that have an "X" mark in Fig. 1 and the extra part does the three convolution kernels that have no "X." The added dummy operation that occurs in the extra part does not affect the result and the same result will be obtained. The method of generating the output feature map with dummy operations may be summarized as Eq. (2).

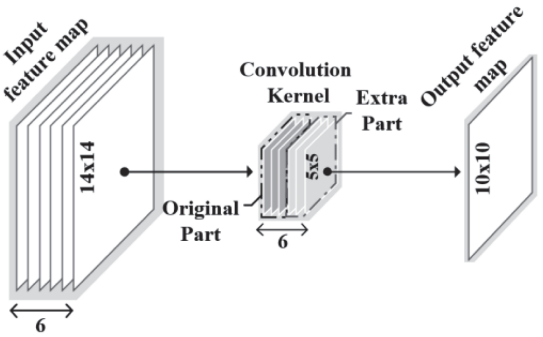

Fig. 2. Convolution combination with dummy operation.

$$out_{idx} = \sum_{idx=0}^{15} In * Kernel_{idx} \quad (2)$$

where *out*, *In*, and *Kernel* denote the output feature map, 6 10×10 input feature maps, and 6 5×5 convolution kernels with 16 types, respectively. $Kernel_{idx}$ indicates a convolution kernel that is grouped into 6 5×5 convolution kernels.

By adding dummy operations, the amount of computation in C2 increases significantly. The operation amount for C2 with the dummy operation (C2 Dummy) is 1.55 times more than that with the conventional operation (C2 Conventional) and the number of parameters is 1.59 times bigger. Especially, the more input images are used, the larger the difference between the conventional operation and the dummy operation will become. The difference between the conventional operation and the dummy operation is about 1,000M operations when 10,000 inputs are executed.

However, the proposed dummy operation method considerably reduces control divergence. It enables the allocation of a kernel to work-group/thread-block with the same number of work-items/threads. Therefore, the utilization of GPU can be maximized. In OpenCL/CUDA, each active work-group/thread block is split into a group of 64/32 threads called wave-fronts/warps. This means that all work-items/threads within a wave-front/warp must execute the same instruction at any given time. It is advantageous performance-wise when the size

of a work-group/thread-block is a multiple of 64/32. From our experiments, it turns out that the best performing work-group/thread block size is 256. The proposed method with dummy operations shows better throughput by up to 3.84 times than the conventional method without the dummy operations.

IV. EXPERIMENTS AND DISCUSSIONS

A. Experimental Setup

We carried out experiments using two different GPU development environments, OpenCL and CUDA. We executed our proposed method on two different heterogeneous (CPU+GPU) platforms: an NVIDIA Jetson TX2 platform for CUDA and an AMD APU A10 7870K for OpenCL.

Jetson TX2 integrates a quad-core ARM A57 CPU, a customized dual-core Denver CPU, and a Pascal-based GPU in a system-on-chip (SoC). The Jetson TX2 platform provides not only a low-power solution but also a comparable computing performance to the desktop GPU.

AMD APU A10 7870k for OpenCL consists of a quad-core CPU and an R7 GPU with 16 GB DDR3 RAM. The APU, which integrates an x86-based CPU and a GCN 1.1-based GPU in a single die, delivers better performance than a platform where a CPU and a GPU are externally linked through the PCI Express bus. The platform specifications of Jetson TX2 and AMD APU A10 are summarized in Table 1.

To evaluate the performance of the proposed method, the modified National Institute of Standards and Technology (MNIST) benchmark [11] was used. MNIST is one of the most widely used benchmarks in OCR. It consists of images that contain a single handwritten digit from 0 to 9 and has 55,000 training sets and 10,000 test sets. Using the weights of pre-trained LeNet-5, execution time and energy dissipation for inference were measured. A PMIC sensor in Jetson TX2 and a profiler called CodeXL in APU were used for measurement [12].

B. Experimental Result

1) Execution Time

To compare the execution time, six different designs were implemented in our experiment. First, as the conventional method, three different implementations were compared: a C/C++ code without any optimization (NONE), an OpenMP-optimized code with 4 threads (OMP), and a GPU implementation with work-groups/thread blocks having different numbers of threads (OCL/CUDA). Second, like in the implementations with dummy operations, three different implementations have been compared: a basic implementation with dummy operations ($NONE_D$), an OpenMP implementation with dummy operations (OMP_D), and a GPU implementation with dummy operations with work-groups/thread blocks having the same number of threads (OCL_D/$CUDA_D$).

As shown in Fig. 3, among the OpenCL implementations, the best speedup of OCL over NONE is 25.72, and that of the proposed OCL_D over $NONE_D$ is 37.26. Among the CUDA implementations, the best speedup of CUDA over NONE is 17.81, and that of the proposed $CUDA_D$ over $NONE_D$ is 21.18.

2) Energy Dissipation

Table 2 shows the amount of power dissipated at the PCI bus, the main memory, the GPU, and the CPU in each implementation. Overall, the implementations with the dummy operation show better energy efficiencies than the implementations without dummy operations. Among the OpenCL implementations, OCL_D dissipates lesser energy than $NONE_D$ by 26.4 times while OCL

Table 1. Specification of target platform

	CUDA (Jetson Tx2)		OpenCL (APU 7870K)	
	CPU	**GPU**	**CPU**	**GPU**
Architecture processor clock	Denvor/Cortex-A57 2 GHz	Pascal 1.3 GHz	X86 3.9 GHz	GCN 1.1 R7 0.866 GHz
Memory bandwidth	59.7 GB/s	59.7 GB/s	25.6 GB/s	25.6 GB/s
Number of cores	Dual/Quad	256 (SP)	Quad	512 (PE)
Performance (GFLOPS)	64/55	750	91.109	665.15
Memory size	8 GB 128 bit LPDDR4		16 GB DDR3	
TDP	15 W		95 W	
Floating point	Single precision (32 bit)		Single precision (32 bit)	
Optimization level	O3		Ox (OpenMP), O3 (OpenCL)	
Operating system	Ubuntu 16.04 64 bit		Windows 10 64 bit	
Library	OpenMP/CUDA 8.0		OpenMP 2.0/OpenCL 2.0	

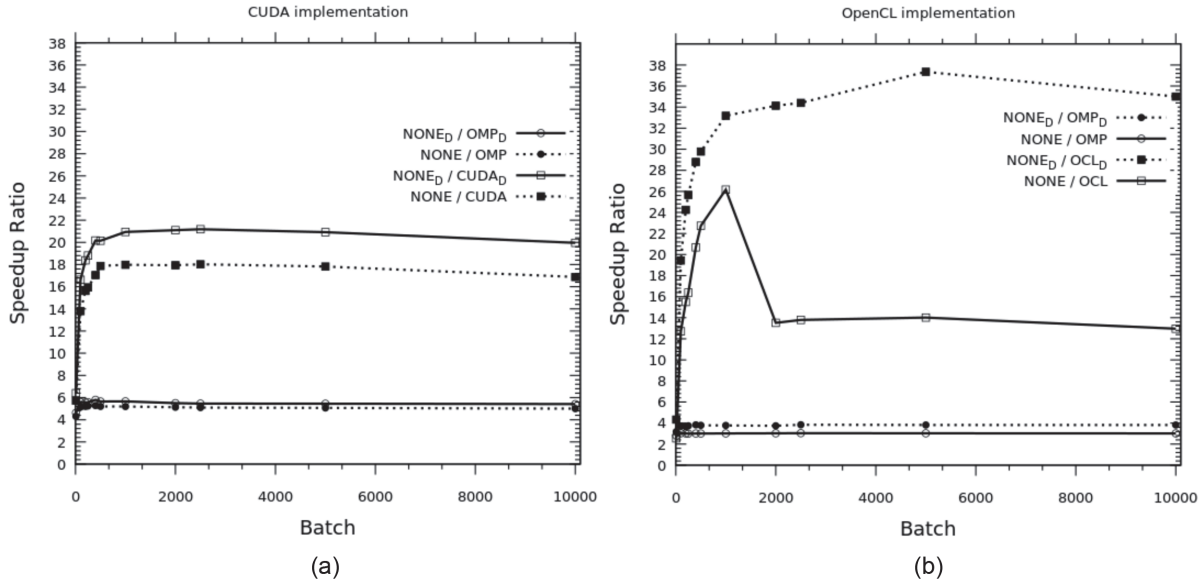

Fig. 3. Speedup of GPGPU implementation performing various batch sizes of a LeNet-5 (10,000 images): (a) CUDA and (b) OpenCL.

Table 2. Energy consumption comparison

		Power (W)				Time (ms)	Energy (J)
		PCI	MEM	GPU	CPU		
CUDA	NONE	-	376	0.76	5.34	13,089	129.01
	OMP	-	4.28	0.83	11.9	2,507	42.61
	CUDA	-	6.54	6.87	4.76	735	13.36
	$NONE_D$	-	3.42	0.73	4.91	15,241	138.12
	OMP_D	-	4.53	0.88	12.99	2,626	48.30
	$CUDA_D$	-	5.0	6.7	4.24	719	11.47
OpenCL	NONE	2.0	5.7	13.7	72.9	25,232	2,279.60
	OMP	2.0	5.5	13.3	74.1	7,536	711.80
	OCL	2.0	6.0	24.6	73.1	981	105.60
	$NONE_D$	2.0	5.5	13.5	69.0	26,083	2,204.00
	OMP_D	2.0	5.6	14.3	75.2	6,807	660.00
	OCL_D	2.0	5.9	22.2	74.3	700	83.50

dissipates less than NONE by 21.6 times. Among the CUDA implementations, $CUDA_D$ dissipates less energy than $NONE_D$ by 12.04 times while CUDA dissipates less than NONE by 9.65 times.

3) Performance Efficiency

Table 3 shows the comparison results in terms of throughput. In this paper, the throughput is defined as the number of processed images per second. In the proposed implementations, CUDAD achieves 13,908.21 and OCL_D achieves 14,285.71. Also, "Throughput/Joule" has been measured, and the proposed implementations with dummy operations show a much better number than the implementation without dummy operations.

Our experimental results confirm that the overall performance of CUDA is better than OpenCL because CUDA provides highly optimized libraries to the NVIDIA GPUs. However, OpenCL is an open-source programming environment and provides better portability to other computing devices such as FPGA. Our proposed method has been verified to be effective in both OpenCL and CUDA.

Table 3. Performance efficiency

		Throughput	Energy consumption (J)	Throughout/Joule
CUDA	NONE	764.00	129.01	5.92
	CUDA	13,605.44	13.36	1,018.37
	$NONE_D$	656.12	138.12	4.75
	$CUDA_D$	13,908.21	11.47	1,212.57
OpenCL	NONE	396.32	2,279.60	0.17
	OCL	10,193.68	105.60	96.53
	NONE	383.39	2,204.00	0.17
	OCL	14,285.71	83.50	171.09

IV. CONCLUSION

In this paper, a novel LeNet-5 implementation with CUDA and OpenCL was presented. A new method to effectively parallelize C2 in LeNet-5 has been proposed. By adding dummy operations, parallel processing on a GPU can effectively be carried out. Further, the optimal size of the thread-block/work-group was empirically determined to achieve a significant speed-up.

A key contribution of this paper is that the proposed method is efficient in terms of both performance and energy consumption in both CUDA and OpenCL. Our experimental results showed an improvement of up to 37.26 times in execution time and a reduction of up to 26.40 times in energy consumption.

ACKNOWLEDGMENTS

This work was supported by the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2022-0-01304, Development of Self-learnable Mobile Recursive Neural Network Processor Technology).

REFERENCES

1. C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Boston, MA, 2015, pp. 1-9.
2. P. Swietojanski, A. Ghoshal, and S. Renals, "Convolutional neural networks for distant speech recognition," *IEEE Signal Processing Letters*, vol. 21, no. 9, pp. 1120-1124, 2014.
3. J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *Artificial Neural Networks and Machine Learning – ICANN 2014*. Cham, Switzerland: Springer, 2014, pp. 281-290.
4. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278-2324, 1998.
5. A. Munshi, "The openCL specification," in *Proceedings of 2009 IEEE Hot Chips 21 Symposium (HCS)*, Stanford, CA, 2009, pp. 1-314.
6. D. B. Kirk, "NVIDIA CUDA software and GPU parallel computing architecture," in *Proceedings of the 6th International Symposium on Memory Management (ISMM)*, Montreal, Canada, 2007, pp. 103-104.
7. J. Shen, J. Fang, H. Sips, and A. L. Varbanescu, "Performance traps in OpenCL for CPUs," in *Proceedings of 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Belfast, UK, 2013, pp. 38-45.
8. S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: efficient primitives for deep learning," 2014 [Online]. Available: https://arxiv.org/abs/1410.0759.
9. R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in CUDA," in *Proceedings of 2014 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, 2014, pp. 1-6.
10. J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications," *IEEE Transactions on Multimedia*, vol. 15, no. 2, pp. 279-290, 2013.
11. L. Deng, "The MNIST database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141-142, 2012.
12. Advanced Micro Devices Inc., "Legacy CodeXL (archived content)," 2002 [Online]. Available: https://gpuopen.com/archived/legacy-codexl/.

Sang-Soo Park https://orcid.org/0000-0002-5916-7687

Sang-Soo Park received his B.S. degree in Electrical and Electronic Engineering, Dongguk University, Seoul, Korea in 2016. Since 2016, he is been taking a unified M.S and Ph.D course at Hanyang University, Seoul, Korea. His research interests include deep learning accelerator, heterogeneous computing for deep learning acceleration.

Ki-Seok Chung https://orcid.org/0000-0002-2908-8443

Ki-Seok Chung received his B.S. degree in Computer Engineering from Seoul National University, Seoul, Korea in 1989 and Ph.D degree 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. Since 2004, he has been a Professor at Hanyang University, Seoul, Korea. His research interests include low power embedded system design, architecture for deep learning accelerator, platform-based SoC verification and system software for heterogeneous many-core systems.