A Compiler for Deep Neural Network Accelerators to Generate Optimized Code for a Wide Range of Data Parameters from a Hand-crafted Computation Kernel

Eri Ogawa†, Kazuaki Ishizaki†, Hiroshi Inoue†, Swagath Venkataramani‡, Jungwook Choi‡, Wei Wang‡, Vijayalakshmi Srinivasan‡, Moriyoshi Ohara†, and Kailash Gopalakrishnan‡

†IBM Research – Tokyo, 19-21, Nihonbashi Hakozaki-cho Chuo-ku, Tokyo 103-8510 Japan
‡IBM TJ Watson Research Center, Yorktown Heights, NY
†{erio, ishizaki, inouehrs, ohara}@jp.ibm.com
‡swagath.venkataramani@ibm.com, {choij, weiwang, viji, kailash}@us.ibm.com

Abstract – This paper presents the design and implementation of a compiler for a deep neural network accelerator that provides high performance and energy efficiency. The compiler allows deep learning frameworks, such as TensorFlow, to exploit the accelerator hardware by automatically creating data transfer code and outer loops around highly-tuned hand-crafted inner-loops for a wide range of neural network parameters. In other words, our compiler significantly reduces the development effort for deep learning libraries without sacrificing their performance. We have evaluated our prototype compiler to show that it can generate code for five most-critical deep learning operators with a comparative performance obtained from hand-tuned code.
(Keywords: Domain-Specific Accelerator, DNN and Compiler)

1 Introduction: Driven by recent advancements in deep learning techniques, accelerating deep neural networks (DNN acceleration) is becoming increasingly important. In fact, many domain-specific accelerators have been developed for this purpose. They provide high processing performance with high energy efficiency by increasing the data and thread level parallelism in a single chip, typically in exchange for reduced functionalities, such as hardware-controlled cache memory for ease of programming. In production-level AI systems, however, these accelerators must be easily programmable by software developers while providing significant performance advantages over general-purpose CPUs and GPUs. Since most software developers in AI fields are expected to use popular deep learning frameworks such as TensorFlow [1] and PyTorch [2] without knowing the details of the underlying accelerator hardware, a compiler that can generate code from high-level DNN descriptions in those frameworks, is a key building block in an eco-system for accelerators.

Top-notch programmers can write highly-optimized computation kernels for DNN accelerators. It is, however, unrealistic for them to develop a complete set of libraries to run all DNN operators for all data sizes to be supported. For example, some DNN accelerators [3][4] are equipped with hierarchical software-controlled small scratchpads instead of hardware-controlled caches. This means that additional software coding is necessary to partition and transfer a large amount of input data to fit it into the hierarchical scratchpads. In addition, it is critical to hide a latency to fetch data into those scratchpads using a double-buffering and pre-loading strategy to overlap computation and data communication. A scheduler [5] for example can generate data partitioning information to specify the temporal and spatial control and to determine optimal data transfer locations for a compiler to generate data transfer code.

In this work, we present the design and implementation of our compiler for a DNN accelerator [3]. It takes two inputs. One is a computation kernel for the computation units and scratchpad controllers with an annotation which defines the category of the operation in the kernel (e.g. an activation function, a pooling function, and a convolution function). The other is data partitioning information (e.g. the input data size obtained from a DNN computation graph, optimized loop blocking sizes, and data transfer positions), generated by a scheduler [5]. By using these two inputs, the compiler automatically generates loops with data transfer instructions to handle temporal and spatial data staging for the hierarchical scratchpads. The compiler further enables performance features, such as double buffering and pre-loading to hide the data transfer latency. Our contribution is a compiler infrastructure to generate high-performance code for a wide range of parameters from hand-crafted computation kernels. We show that our generated code for Convolution, ReLU, Sigmoid, Average pooling and Softmax, which are five most-critical deep learning operators, is comparable with hand-crafted code in terms of the performance and energy efficiency.

2 Target Hardware Architecture: The target deep learning acceleration core [3], being developed by IBM, is equipped with systolic array processing units (PEs), a row of vector processors (SFUs) and two-level scratchpads (L0 and Lx) as shown in Fig. 1. Each scratchpad has a pair of control units, a load unit (LU) and a store unit (SU). One processor chip consists of many cores that share one external device memory. Each core has a device memory controller (DMC) with a load unit (LU) and a store unit (SU) to transfer data between the Lx scratchpad and the device memory. Each processor core has many computation and data transfer units, and each unit has its own instruction buffer and program counter. The compiler needs to generate code for all computation and data-transfer units (PE, SFU, L0 LU/SU, Lx LU/SU, and DMC LU/SU) separately. In addition, each processor core uses two hierarchical scratchpads. All data transfers between the device memory and the Lx scratchpad and those between the Lx and L0 scratchpads are software controlled; the compiler must determine the load/store address for each data transfer instruction.

Each computation and data-transfer unit is architected with a carefully curated instruction set and an instruction buffer with limited capacity to achieve high-performance and energy-efficient deep learning processing. The compiler generates and optimizes instructions to fit them in the instruction buffer.

3 Software Stacks for the Target Processor: Fig. 2 shows our overall software stack for TensorFlow, which is our initial target framework. Software developers create their DNN on TensorFlow as a computation graph to run it on this accelerator.

We use a scheduler [5] at the second layer in our software stack to partition the computations among the processing elements and to partition the data into multiple stages to be transferred to each level of the memory hierarchy. It takes a description of the DNN from TensorFlow and the parameters of the hardware as its input to optimize the mapping of the DNN onto the architecture. It outputs the mapping information as a data partitioning parameter for each operator in the DNN.

The compiler, the third layer in our software stack, generates optimized code using the data partitioning parameter from the scheduler and an annotated computation kernel for each deep learning operator, as described in the next section.

4 The Compiler Overview and Advantages: Fig. 3 illustrates the structure of the compiler. It takes a data partitioning parameter from the scheduler and an annotated computation kernel for a deep learning operator to generate optimized code in the following four steps: a) to generate code for controlling additional loops to repeat executing the kernel, b) to generate code for transferring data between the device memory and a scratchpad, c) to enable performance features, such as double buffering and pre-loading, and d) to generate code for zero-padding, where the DMC aligns the input data for convolution operators by filling additional zeros when it transfers data from the device memory to the Lx scratchpad.

The annotated computation kernel represents a deep learning operator, including the operation category and kernel computation instructions for each hardware unit, hand-crafted by top-notch library developers. Using the kernel instructions and the operation category from the kernel, as well as the data partitioning parameter from the scheduler, the compiler automatically generates outer loops for the kernel for each computation and data-transfer unit. For the DMC, the compiler generates its code to transfer data between the device memory and the Lx scratchpad. The library developers need to write instructions for all computation and data-transfer units except for the DMC only on one core; the compiler automatically generates data transfer code for multiple cores by determining the actual address of data for each core.

The compiler has a hierarchical intermediate representation (IR) that represents code structures, such as loops, conditional branches, and address updates. First, the compiler generates a 'KernelBlock' IR which stores the kernel computation instructions. This IR is already hand-optimized, not to be optimized by the compiler. Second, the compiler generates other blocks such as a loop and an address update based on the operator category and the data partitioning information. Third, the compiler optimizes the code. It eliminates unnecessary instructions (e.g. add zero to a register) and fuses certain nested loops to reduce the number of instructions. Finally, it generates assembly and binary code for the target hardware.

Currently, the compiler supports nine operation categories to generate code for major DNN operators listed in Table. I. One operation category can represent multiple deep learning operators with the same data access pattern. For example, the 'ActFnFwd' category represents activation functions, such as ReLU. Operators in the category performs a forward computation for each element in its input data array independently. Thus, it can represent other activation functions, such as Sigmoid. If a programmer wants to add a new activation function, he or she can simply write its computation kernel by using the same operation category without modifying the compiler. For example, one can implement LeaklyReLU using the 'ActFnFwd' category.

Fig. 4 shows an example of an annotated computation kernel for a ReLU operator (a) and an IR constructed from the kernel for LxLU (b) and DMC (c). The ReLU function returns zero if an input is zero or negative, and returns the input value if it is positive. Thus, the kernel for the SFU consists of only one MAX instruction to take a max value between a FIFO input from the L0 (y) scratchpad and zero. The kernel for the L0 LU and SU consists of a LOAD and a STORE instruction respectively to represent two data movements: one from the L0 scratchpad to the SFU and the other from the SFU back to the L0 scratchpad. The kernel for the Lx LU and SU is the same as for the L0 LU and SU, respectively, as these two units simply transfer each input data from the Lx scratchpad to the L0 (y) scratchpad and also from the L0 (y) scratchpad to the Lx scratchpad, respectively. By using this annotated kernel and data partitioning parameters from the scheduler, the compiler constructs one IR for each execution unit, like one shown in Fig. 4 (b). The scheduler generates data partitioning parameters of many patterns to specify the order of loop tiling, the data-transfer position in the tiled loop, and the data size stored for each scratchpad by using hardware parameters and given data size parameters. The compiler generates instructions for loops and address updates for all of their valid patterns. The compiler furthermore generates data transfer instructions for the DMC LU/SU automatically from the operation category and the data partitioning parameters, as shown in Fig.4 (c).

5 Evaluation: We evaluated the performance of the compiler generated code on a cycle-accurate simulator of the target accelerator written in SystemC. Fig. 5 shows the execution time of the compiler generated code for five most-critical DNN operators; ReLU, Sigmoid, Average pooling, Softmax, and Convolution. The results are normalized by the time of the highly-tuned hand-crafted code. As shown in the graph, the compiler generated code for all of the five operators can achieve almost the same performance as the hand-crafted code. Especially for Softmax and Convolution, compiler generated code can achieve higher performance than the hand-crafted code because of an extra compiler optimization, which the library developer did not conduct due to the complexity in the operator code. While it is not shown in this extended abstract due to the space limitation, for all configuration parameters the scheduler generated in our experiments, our compiler was able to generate high performance code as equally tuned as top-notch library developers could have written.

6 Conclusion: Our devised compiler was designed for DNN accelerators that have hierarchical software-controlled scratchpads instead of hardware-controlled caches. It takes highly-optimized computation kernels for high performance and energy efficiency and data partitioning parameters from a scheduler that analyzes a computation graph input. This approach allows us to generate optimized code to execute a DNN operator for a wide range of input data parameters, such as the data size, from one highly-tuned hand-crafted kernel without sacrificing the performance. As a future work, we will implement more optimization techniques to increase the performance and energy efficiency and will support more DNN operators. In addition, we will add software stacks for other DL frameworks such as PyTorch.

[1] M. Abadi *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*

[2] P. Adam *et al.*, "Automatic differentiation in PyTorch," *31st Conference on Neural Information Processing Systems (NIPS)*, 2017

[3] B. Fleischer *et al.*, "A Scalable Multi-TeraOPS Deep Learning Processor Core for AI Training and Inference," *2018 IEEE Symposium on VLSI Circuits*, Honolulu, HI, 2018

[4] Y. Chen, T. Krishna, J. Emer and V. Sze, "14.5 Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, San Francisco, CA, 2016

[5] S. Venkataramani, J. Choi, V. Srinivasan, K. Gopalakrishnan and L. Chang, "POSTER: Design Space Exploration for Performance Optimization of Deep Neural Networks on Shared Memory Accelerators," *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), Portland, OR, 2017*

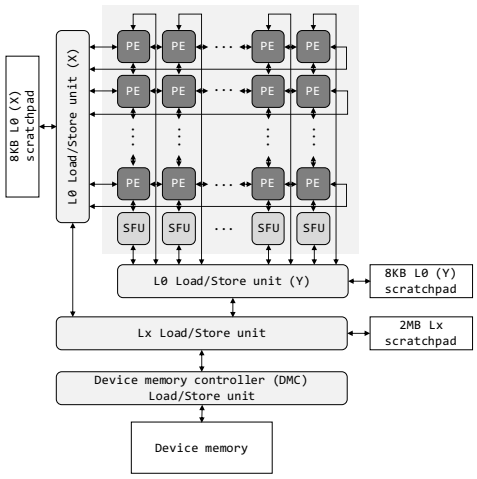


Fig. 1 Hardware Overview

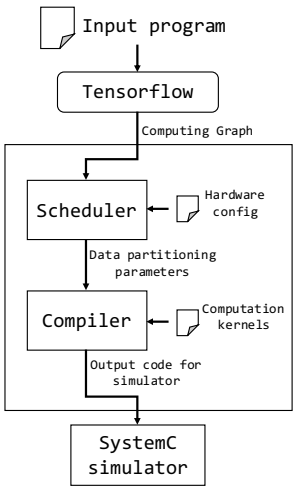


Fig. 2 Software Stack

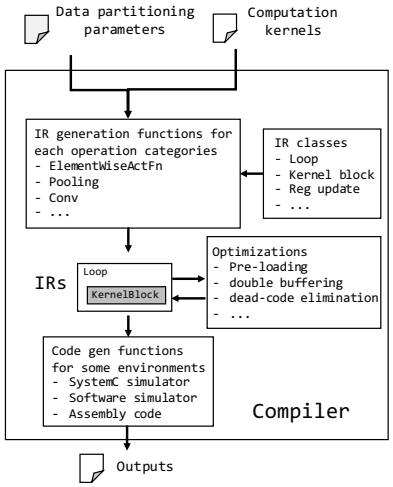
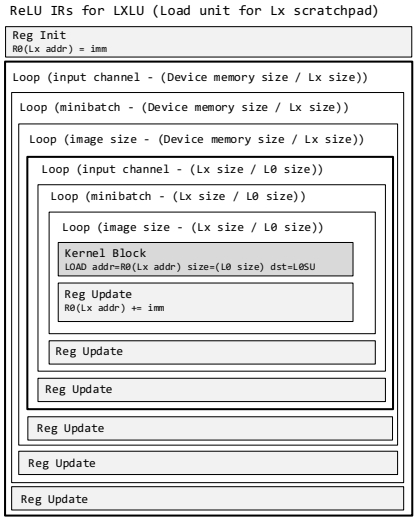


Fig. 3 Compiler Overview

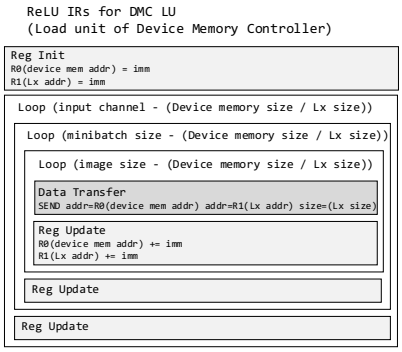
```
/// @type:ActFnFwd

/// @unit:LXLU
LOAD addr=R0 size=0x80 dst=L0SU
/// @unit:L0SU_Y
STORE addr=R1 size=0x80 src=LXLU
/// @unit:L0LU_Y
LOAD addr=R2 size=0x80 dst=SFU
/// @unit:SFU
MAX src0=L0_FIFO src1=0 dst=L0SU
/// @unit:L0SU_Y
STORE addr=R3 size=0x80 src=SFU
/// @unit:L0LU_Y
LOAD addr=R4 size=0x80 dst=LXSU
/// @unit:LXSU
STORE addr=R5 size=0x80 src=L0LU
```

(a) A computation kernel for ReLU



(b) The IRs for LxLU



(c) The IRs for DMC

Fig. 4 Example: (a) annotated computation kernel for ReLU operator, (b) IR to represent the computation for LxLU, and (c) IR to represent the computation for DMC

Table. I DNN operation categories and corresponding DNN operators supported by compiler

Operation Category	Operator
Conv	conv2d
ActFn Fwd/Bwd	relu
	sigmoid
	tanh
	exp
	reciprocal (1/x)
	rsqrt (1/sqrt(x))
Pooling Fwd/Bwd	avg_pool
PoolingReqIndex Fwd/Bwd	max_pool
LinearFn	axpy
	hadamard product
MultiLoopsPrograms	softmax
	batch normalization

Fig. 5 Execution time of compiler generated code for five DNN operators, normalized by that of highly-tuned hand-crafted version