An In-depth Comparison of Compilers for Deep Neural Networks on Hardware

Yu Xing[*†‡§¶], Jian Weng[†], Yushun Wang[†], Lingzhi Sui[†], Yi Shan[†], Yu Wang[*‡§‖]
[*]Department of Electronic Engineering, Tsinghua University, Beijing, China
[†]Xilinx, Beijing, China
[‡]Beijing National Research Center for Information Science and Technology, Beijing, China
[§]Center for Intelligent Connected Vehicles and Transportation, Tsinghua University, Beijing, China
[¶]xingy16@mails.tsinghua.edu.cn [‖]yu-wang@tsinghua.edu.cn

Abstract—**Deep neural networks (DNNs) are currently the foundation for many artificial intelligence tasks. The difficulty of mapping NN models to high-performance hardware implementations arises from factors ranging from the computation complexity of multiple operations to different hardware features such as memory hierarchy and parallelism. In this article, we present a generic compiler process flow and make an in-depth comparison of compiler frameworks regarding their domain-specific language (DSL), intermediate representations (IRs), optimization strategies and autoscheduling methods. We reimplement typical NN models based on these compiler frameworks and evaluate the resulting performance. We also review our previous work(Deep Neural Network Virtual Machine, DNNVM) on compiler frameworks and optimization for a custom FPGA-based accelerator to gain inspiration regarding the difference between compiler design for general-purpose processors and that of FPGA-based accelerators.**

Index Terms—**Compiler, deep neural network, optimization**

I. Introduction

The development of deep neural networks (DNNs) is driving an explosion in multiple artificial intelligence (AI) domains. DNNs currently achieve state-of-the-art performance in multiple AI applications, such as computer vision, robotics and natural language processing. To date, many well-designed, high-performance machine learning systems such as TensorFlow[1], Caffe[2], and PyTorch[3] exist and allow programmers to experiment with various DNN algorithms in a quick and elegant way.

Nevertheless, the appealing accuracy and ability of DNN comes at the cost of high computational complexity. In general, most implementations of DNNs are based on existing general-purpose computation engines, especially CPU and GPU platforms. When applications identify the needs for custom computation, improved efficiency in computation, lower power consumption/design cost, or physical system size, it is a tedious task to optimize the source code of algorithms targeting CPUs and GPUs. In recent years, there has been a significant trend in designing specialized processing units such as FPGA-based accelerators[4], [5] or ASICs[6], [7] to meet these aggressive platform requirements and accelerate DNNs. Despite the advantages of custom platforms, the intricacy of design flows remains a barrier to the adoption of custom accelerators.

To transform each segment of applications to an optimized version of implementation, compilers have been used for several decades[8]. Benefiting from the mature economy involving CPUs and GPUs, compilers are capable of generating platform-dependent code efficiently from high-level programming languages, while optimized implementations of DNNs are provided by linear algebra acceleration libraries such as Eigen[9], MKL[10] and OpenBLAS[11]. In addition, in the design of custom accelerators, computer-aided tools play a key role in mapping different DNNs into hardware blocks and generating efficient instructions executed by platforms.

Unfortunately, neural networks are computationally intensive and involve latency-critical tasks. It is a tedious process for users to write algorithms to fit the linear algebra acceleration libraries. It is also very hard for existing compilers to integrate newly introduced optimization methods to keep up with the pace of the rapid development of algorithms; they cannot readily provide a sufficient acceleration rate to bridge the gap between the written algorithms and target hardware. Building on these considerations, to improve the throughput of devices and enhance productivity, several compiler-inspired frameworks have appeared in recent years that intelligently simplify the realization of optimized neural network performance. In this paper, we leverage an in-depth comparison of the processing flow of these compiler frameworks and provide an analysis of their optimization methods. We also perform extensive experiments using several mainstream neural networks and present our practical experience. For this purpose, we use best-effort reimplementations based on the original papers and tutorials. Our main contributions can be summarized as follows:

- We present a generic compiler process flow and explain the challenges of compilers for deep neural networks on hardware.
- We analyze the difference of optimization strategies used in existing compiler frameworks[4], [5], [12]–[19].
- We fairly and empirically evaluate these compilers, targeting general-purpose processors (GPPs) or specialized accelerators on frequently used neural networks. We highlight the difference between the achieved throughput of GPPs and custom accelerators.

To the best of our knowledge, this paper is the first study to compare newly designed compilers for deep neural networks. The remainder of this paper is organized as follows. Section

978-1-7281-2437-7/19/$31.00 © 2019 IEEE

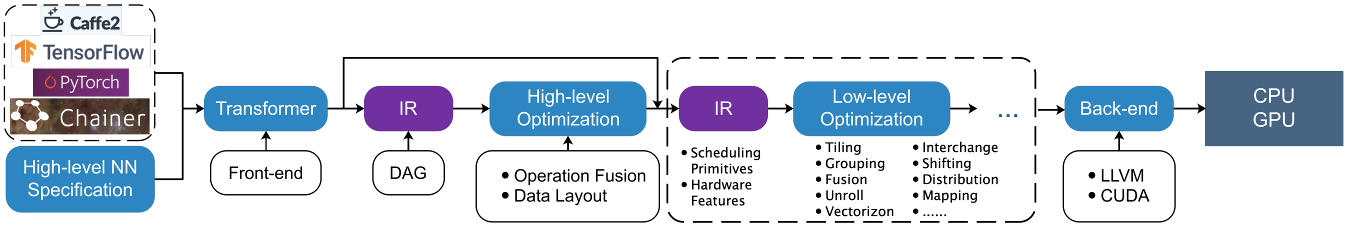

Fig. 1: Generic Compiler Frameworks for Deep Neural Networks

2 reviews the challenges and universal processing flow of compiler frameworks for DNNs. Section 3 presents the details of existing compilers and the difference between them. We introduce our experimental environments in Section 5 and present the analysis and results of the evaluation. Section 6 summarizes and concludes the paper.

II. Compiler Optimization Concept

A. Generic Compiler Frameworks for Deep Neural Networks

Compiler frameworks for deep neural networks work in the context of high-level DNN specifications, especially models from deep learning frameworks such as Caffe[2], TensorFlow[1], MXNet[20], and PyTorch[3]. The optimization steps are applied at different stages of the compilation process, and the processing flow of these compiler frameworks can be categorized into five layers: 1) *front end*, 2) *intermediate representation* (IR), 3) *high-level optimization*, 4) *low-level optimization* and 5) *back end*.

First, *front ends* transform high-level specifications of deep neural networks into compiler-specific IRs. These IRs are usually in the form of directed acyclic graphs[1], [3], in which each node represents a computation operation and each edge denotes the data dependency between operations. As a result, graph-level algorithms[13], [19] can be used upon these IRs to fuse operations and optimize data layouts. Apart from the high-level IRs, multiple extensive IRs[21]–[24] are adopted in the optimization process of compilers. Deep neural network workloads can be decomposed into tensor operations, such as matrix-vector and matrix-matrix multiplication. Low-level optimization methods[12], [13] are used to optimize the schedule for enhancing data locality and making full utilization of the parallelism of hardware platforms. This problem turns into what optimization to use and which parameters to choose from (e.g., tiling size, fusion strategies, and vectorization). Hundreds of low-level optimization steps may be applied during the compilation phases. Finally, the back end[24], [25] is responsible for mapping optimized implementations into machine-dependent executable instructions.

B. Challenges

1) Intermediate Representations (IRs): Well-known deep learning frameworks offer high-level abstraction for deep neural networks expressed as a computing graph. TensorFlow employs a static dataflow graph of operators and offers highly optimized implementations, in which GPUs and other specialized accelerators are transparent to the users. However, a static computing graph cannot support computations that are not explicitly specified. For example, the size of the input and output needs to be specified, and all required data should be loaded on-chip before execution of an operation. Dynamic frameworks such as PyTorch and Chainer adopt a define-by-run computing graph to alleviate this problem, but the control flow is lost due to the dynamic computing graphs.

In addition, these deep learning frameworks lack efficiency in the instances where researchers need to develop a custom operator. Hundreds of lines of codes need to be written manually to express the algorithm. The computing graphs in these frameworks lack features and information from the hardware as well. The computation efficiency decreases dramatically when the operators do not fit the preoptimized version of the library functions. At the very least, we need to make $O(N_f \cdot N_p)$ efforts to optimize operations in N_f deep learning frameworks targeting N_p hardware platforms.

To solve these problems, an effective design mentality is to decouple the algorithm description with deep learning frameworks and hardware platforms. IRs should not only provide concise, portable and expressive syntax to represent the NN models and control flow but also provide a powerful abstraction containing both the features of algorithms and hardware platforms for the following analysis and optimizations.

2) Scheduling Pipelines: Given the expression of algorithms, schedulers contain the rules to map computation descriptions to implementations for different hardware platforms. Preoptimized libraries such as Eigen[9] and cuDNN[25] provide various reliable and fast implementations for linear algebra, but these libraries lack optimization across operators, and the execution of each operation varies dramatically for different data sizes, data layouts, configurations for operators, memory hierarchies and specific hardware features. Determining when the functions should be computed, where the data should be stored, and how long they should be cached, in addition to configuring the trade-off between recomputation and data locality, are the main challenges for the scheduler. The combination of fusion and tiling are the most common methods to enhance producer-consumer locality and make full utilization of parallelism, but the optimization space is too large to be explored.

3) Autotuning: Let the optimization sequence of the scheduler in a compiler contain n optimization passes. If we focus on whether to apply the optimization, then we have 2^n optimization options to select from. Furthermore, if each optimization pass has a many-choice option with m variants, then the total optimization space becomes $\prod_{i=0}^{n} m_i$. If we take the ordering of optimization steps into account, the optimization search possibilities become $n!$ due to the permutations. Hence, difficulties often arise from the combinatorial explosion of optimization choices. If all optimization steps and scheduling are manually specified, it would incur a high engineering cost to achieve an ideal performance even for the most experienced engineers. Autotuning refers to a methodology incorporating a model with which users can traverse the entire optimization space efficiently. For autotuning, it is challenging to develop an approach that is able to traverse all potentially profitable optimization choices incorporating a precise execution cost with finite time complexity.

4) Back Ends and Code Generation: The back end is responsible for emitting machine code for the optimized implementations. In general, the IRs and programs are transformed into LLVM[24] or CUDA[25]/OpenCL[26] source code. Unfortunately, several patterns of the implementations may generate poor code when passed directly to LLVM. Additionally, due to the custom instruction set architectures for specialized accelerators, back ends for custom accelerators need to be designed explicitly from scratch. In some extreme cases, the end implementations might not target an individual CPU or GPU kernel, and the hybrid execution of CPU/GPU/FPGA/ASIC platforms introduces new challenges. A small change in the implementation can affect the memory management, communication between devices, synchronization and optimization choices.

III. Implementations of Compiler Frameworks

In this section, we analyze 4 full-stack compiler frameworks for deep neural networks to generate optimized implementations on CPUs and GPUs. We focus on the domain-specific representations, high-level transformations and data scheduling optimization. In addition, we present 5 compiler architectures for specialized FPGA-based accelerators. Other compilers, such as Intel's nGraph[27] and TensorFlow XLA[16], are still experimental and in active development, and we do not discuss them in this paper. There is not much literature or source code for compilers targeting ASICs such as TPU[7] or Diannao[28] for reference, so we do not include them either.

A. Halide

The Halide compiler[12] was originally designed for image processing, and neural networks have many similarities with image processing. They are both composed of a long computation sequence of many operations, and they both combine the challenges of stencil computation and stream programs. Thousands of lines of code for the optimized pipeline must be written manually in C, CUDA or assembly for each complex operation to achieve the peak performance, even by an experienced engineer. The optimized pipeline cannot be ported to other architectures. In consideration of these aspects, Halide applies high-level abstraction and efficient schedule methods to improve portability and composability.

1) Domain-specific Language (DSL) and Representations: Halide‘s DSL decouples the algorithm definition from the execution strategy. Instead of providing specific values to describe a function, algorithms are defined as pure functions over an infinite integer domain. The DSL avoids higher-order functions, dynamic recursion and complex data structure. To express operations such as convolution and pooling in neural networks, a recursive reduction function is applied by defining the minimum and maximum value for the targeting dimension. The scheduling representations provide efficient descriptions for the implementations of parallel, vectorized, tiled, fused and reordered functions. Halides DSL is simpler than most functional languages and is sufficient to describe most operations and scheduling methods for neural networks.

Listing 1: Algorithm description of convolution written manually in Halide DSL. By setting the potential range of shapes of the input $Buffer$ and $Variables$, the autoscheduler can search for valid CPU execution strategies automatically.

```
//Data are stored in an on-chip buffer.
Input<Buffer<float>> input,wtB,bias;
Input<float> pad_l,pad_t,stride_w,stride_h;
//Set the boundary condition for functions.
Func in = BoundaryConditions::constant_exterior(
    input, 0);
Func in_w = BoundaryConditions::repeat_edge(wtB);
Func in_b = BoundaryConditions::repeat_edge(bias);
//Pre-declare the variables and functions.
Var w,h,c,oc,kw,kh;
Func fw,fh,conv2d,convsum;
//Algorithm description of convolution.
fw(w) = w*stride_w - pad_l;
fh(h) = h*stride_h - pad_t;
RDom kernel(0,wtB.width(),0,wtB.height());
conv2d(w,h,c,oc) += in(fw(w)+kernel.x,fh(h)+kernel
    .y,c) * in_w(kernel.x,kernel.y,c,oc);
//Reduction of the dimension of the channel.
RDom channel(0, input.channels());
convsum(w,h,oc) += conv2d(w,h,channel,oc);
//Add the bias.
output(w,h,oc) = convsum(w,h,oc) + in_b(oc);
//Set bounds_estimate for pre-defined buffers.
//Halide optimizes the schedule under such
    constraints.
input.dim(i).set_bounds_estimate(0,value_{i}); ...
output.estimate(w,0,16).estimate(h,0,16).estimate(
    oc,0,4096);
```

2) Autoscheduler: At the very beginning, the optimization in Halide is semiautomatic, and the schedule is specified by the user. The schedule search space in Halide is enormous due to the combination of choices of marking the implementations of loop parallel, reorder, vectorized or unrolled, caching behaviors and hybrid code generation for various devices. The primary autotuner[12] in Halide applies stochastic search and genetic algorithms to optimize schedulers for pipelines. It starts from seeding potentially profitable schedules to initial populations of a fixed size (128) and constructs a new generation with crossover elitism, mutated and random individuals. In recent work, Mullapudi et al. proposed a model-driven autoscheduler[29] for Halide. The heuristic autoscheduler first determines the best tile size for each group to maximize input data reuse. Then, the scheduler enumerates all valid grouping opportunities with a direct producer-consumer relationship to reduce memory communications. However, different tile sizes for the grouping operations introduce additional recomputation, so the authors set some rules for candidate tiling: 1) the minimum of the innermost dimension of tiling should be more than `VECTOR_WIDTH` so that the loop nests can be efficiently vectorized, 2) the number of tiles should be larger than the number of CPU cores to enhance parallelism, and 3) the memory footprint is dependent on the last-level `CACHE_SIZE`. Then, the autoscheduler provides a cost function to estimate the performance improvement of grouping and tiling by adding the arithmetic cost of implementations ARITH_COST to the total number of loads LOAD_COST. The performance benefit of inlining a function to consumers is also considered by the scheduler. Finally, the autotuner selects the optimal schedule strategies with the minimal cost.

3) Portability to Different Architectures: The Halide autoscheduler provides the ability to generate an optimized pipeline targeting various hardware architectures. By altering the values of PARALLELISM_THRESHOLD, VECTOR_WIDTH, CACHE_SIZE and LOAD_COST as mentioned above, the autoscheduler can be adapted to CPUs such as the Xeon and ARM CPUs. Alone, the autoscheduler can generate schedules for GPUs in a similar way, and the number of threads per GPU thread block is constrained by MAX_THREADS_PER_BLOCK to avoid generating invalid strategies.

B. TVM

TVM is an end-to-end full-stack compiler framework that maps high-level specifications of deep neural networks from multiple deep learning frameworks to low-level optimized code for a diverse set of hardware back ends. In recent months, TVM has become a community that attracts many developers to optimize their models based on TVM stacks.

1) Relay[22]: Relay is the front end of TVM. The computing graphs in TensorFlow are static graphs with a fixed topology. It is easy to optimize each operation, but users can construct their own operations only in a deeply embedded DSL. Dynamic computing graphs as adopted by PyTorch and Chainer[30] provide the convenience to describe the operators, but it would be very hard to leverage optimization across the operation and hardware platforms. As a result, Relay presents a new high-level IR to provide expressiveness and efficient compilation from the perspective of a programming language instead of the previous dataflow representations.

2) Graph-level Optimization: Graph-level optimization of the computing graphs of deep neural networks has been demonstrated to be effective by many off-the-shelf tools[31]. TVM focuses on operation fusion and data layout transformation. Operator fusion in TVM combines adjacent operators: 1) those that can be precomputed, 2) reduction functions, and 3) pointwise operations to reduce data transfer between the on-chip buffer and off-chip memory. Data layout can be converted for better execution on the target hardware. Based on TVM, Amazon[32] has presented a new data layout method to accelerate an NN on a CPU.

3) Automating Operator-level Optimization: Unlike the high-level representations, optimized implementations of each operator are opaque to users. Internally, TVM reuses helpful schedule primitives in Halide and extends the primitives to optimize the GPU and specialized accelerator performance. For example, TVM provides schedule primitives that assign data into the shared memory in GPU so that groups of threads can fetch the data they need cooperatively. Additionally, TVM decouples the schedule primitives with the hardware intrinsically, so the compiler is capable of matching schedule patterns with hardware implementations.

Given the schedule primitives, users could leverage optimization for scheduling either manually or based on the experience provided by TVM developers. Such an approach is inefficient, so the remaining problem is to find an automatic method for scheduling optimization. By comparing random search and blackbox genetic algorithms, which are similar to the methods in the previous edition of Halide with ML-based models[33], [34], users can find an ML-based model: a gradient tree boosting model that provides high speedup quality and requires little training cost.

4) Hybrid Execution: TVM supports multiple types of hardware platforms, including a server-class CPU/GPU and an embedded-class CPU/GPU. In particular, the TVM group provides a codesign of the hardware and software tool VTA[35], which can generate hardware architecture based on FPGA through high-level synthesis (HLS) and generate hybrid implementations for both the ARM CPU and FPGA.

C. DLVM

The IR in DLVM is a graph-based, modular representation that has a set of hierarchy of abstractions, including module, function, basic block and instruction. In particular, each module contains type definitions and functions, each function contains a control flow graph formed by basic blocks, and each basic block contains instructions with data dependencies in the form of a DAG. The virtual instructions in DLVM include basic fine-grained math operators, which can be categorized into 1) elementwise operators such as add and tanh and 2) complex operators such as dot and convolve. Optimization steps in DLVM include algebra simplification, linear algebra fusion, matrix multiplication reordering and some traditional compiler optimization steps. Ultimately, DLVM IR exists at a lower level than implementations of BLAS and computation kernels of LLVM for code generation.

D. Tensor Comprehension (TC)

On GPUs, the achieved performance of parallel execution of operations through the preoptimized library functions depends on the data size, data layout and various hardware features of the GPUs. However, creating a library that covers all transformations and optimizations for a combination of these features and leverages optimizations across operators is feasible. All scheduling transformations of algorithms for GPUs should be written manually in Halide. More recent deep neural network compilers such as XLA and Latte[36] cannot readily achieve the ideal throughput even though they take the data size and cross-layer optimization into consideration. Building on these considerations, TC presents an expressive DSL that can efficiently describe the algorithms. Based on the DSL, the compilation flow maps the high-level representation into the polyhedral model to explore scheduling of the optimization space and generates highly optimized GPU code automatically.

1) High-level Representations and DSL: Instead of designing an embedded DLS such as Halide (embedded in C++), TC avoids a verbose process when addressing debugging and warnings of the embedded DSL. The presentation of computing and multiarrays in TC is inspired by OoLaLa[37], TACO[38] and Einstein notation[39], which prevents users from predeclaring variables and functions, simplifies the reduction operations and eliminates the influence of evaluation sequence of points on the output. To integrate TC into deep learning frameworks, they provide APIs to transform the NN model from deep learning frameworks into TC. A single TC corresponds to a node in the computing graph in the ML framework. When adding a new operator, users can write their own TC implementation instead of using back-end implementation.

Listing 2: Algorithm description of convolution written manually in TC DSL. The autotuner in TC traverses scheduling opportunities based on hardware features and leverages a generic algorithm to find the optimal execution strategies.

```
// No predeclaration of variables like B,C,H or W
def conv_relu(float(B,C,H,W) Input, float(CO,C,KH,KW) Weights, float(CO) Bias, float(1) kernel_w, float(1) kernel_h)->(Output): {
//Indices that appear on the right but not on the left are assumed to be reduced dimensions.
Output(b,co,h,w) += Input(b,r_c,h*stride_h[0]+r_kh, w*stride_w[0]+r_kw) * Weights(co,r_c,r_kh,r_kw)
Output(b,co,h,w) = fmax(0.0,Output(b,co,h,w)+Bias(co))}
//TC autotunes the schedule strategies for a specific shape of the data each time.
 H,W,C,B,F,CO,KH,KW = 224,224,3,1,32,3,3
 tc.autotune_and_compile(conv_relu, input, weights, bias, tuner_config)
```

2) Polyhedral Compilation: Halide explores the interaction of grouping and slicing based on Halide IR to optimize the scheduling[29]; more recent work[40] presents a fusion model with potentially profitable fusion strategies that are not covered by the previous Halide approach. Instead, the TC compiler transforms the TC representations into Halide IR and then lowers Halides IR into a polyhedral IR. This process is realized by PENCIL[41] and pet libraries, but the PENCIL IR is ultimately bypassed to bridge the mismatch between high-level operations and polyhedral code. The core polyhedral scheduling in TC is provided by isl[42], and a data dependency graph is generated internally by analyzing the algorithm and loops. A combination of affine transformations such as tiling, mapping, shifting, fusion, distribution and interchange upon the loops without changing the data dependency exposes more opportunities for scheduling optimization. TC extends the isl by providing more fine-grained control, and additional custom constraints can be inserted into the program. All affine maps can be integrated into a schedule tree, and the schedule tree can be used to present loop tiling, mapping to blocks and threads, and mapping tensors into shared or private memory and registers. The mapping algorithms are borrowed from PPCG[43], but more complex scheduling methods and optimization opportunities regarding imperfectly nested structures are taken into consideration by TC.

3) Autotuning Methods: In Halide, the value of the data size is not specified, and the algorithm is described over an infinite integer domain. However, in TC, the generated code depends on the specific input shapes and other options such as hardware features, including the shared memory size and register size. After setting these options as an entry key, the autotuner in TC sets up candidate configurations about the tile size, block mapping, fusion strategies and shared memory usage randomly, and then each tuning is compiled and profiled on the GPUs. After obtaining the implementation costs of each tuning, TC leverages a genetic search to find the optimal execution strategy. Finally, the compilation cache stores the generated CUDA code, which holds the fastest known implementations for each entry key to enable reuse.

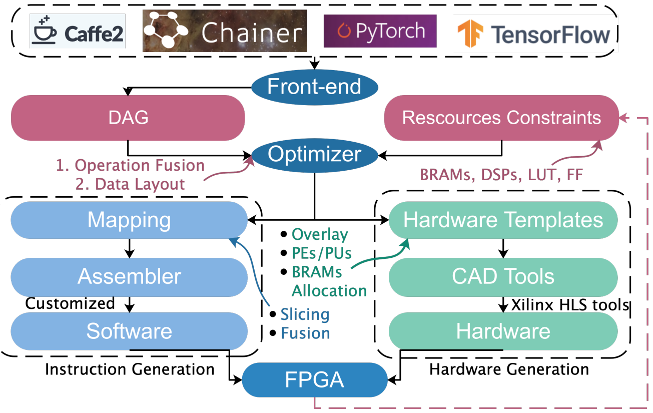

Fig. 2: Generic Compiler Frameworks for FPGA-based Accelerators

E. Overview of fpgaConvNet, DNNWeaver, DLA[18], xfDNN, and DNNVM

In Figure.2, the function of compilers for the FPGA-based specialized accelerators can be divided into two categories, which are to map neural networks into 1) hardware blocks[4], [5], [18] and 2) instructions executed on hardware[17]–[19].

The compiler framework of fpgaConvNet is based on the synchronous dataflow (SDF) paradigm graph[44], which is able to obtain a predictable amount of required on-chip memory and provide a static execution strategy. By analyzing the interaction between the DAG of the DNN model and the platform-specific resource constraints, the compiler transforms the computing graph into an SDF hardware IR, in which each node represents a hardware block. The compiler tiles the DNN model into segments and generates an optimized hardware block for each subgraph separately. DNNWeavers compiler adopts a dataflow graph generated from Caffe. When given a specific NN model, DNNWeaver leverages an automatic resource optimization algorithm to maximize the performance by varying 1) the number of PEs per PU and 2) the output slice. Additionally, DNNWeaver provides a custom ISA to decouple accelerators with different FPGA platforms.

DLA presents a compiler and FPGA overlay for deep neural network acceleration. On the hardware side, DLA proposes a very long instruction word (VLIW) that introduces negligible overhead. By continuously fetching the VLIW from external memory, the VLIW is decomposed into segments and sent to modular hardware kernels connected by Xbar. DLAs hardware supports vectorized and parallel implementations. By changing the parallelism in the width `Q_VEC`, height `P_VEC`, input channel `C_VEC`, and output channel `K_VEC` dimensions, the overlay achieves the optimal implementation efficiency. On the software side, the DLA compiler slices the feature

	Halide	TF(XLA)		TVM		TC	DNNVM	xfDNN	fpgaConvNet*
Benchmark	CPU	CPU	GPU	CPU	GPU	GPU	FPGA	FPGA	FPGA
VGG	751.6	205	3	235	9	-	20.1	24.33	249.4
ResNet50	600.8	102	5	130	11	18.6	13.5	12.42	-
ResNet152	1683	287	13.6	343	30	48.3	36.4	34.87	153.84
Inception_v3	826.1	137	10.5	156	19	-	13.6	24.62	-
MobileNet_v1	197.3	15	6.5	50	2.9	3.17	3.4	1.5	-
SqueezeNet	139.1	10.6	7.6	20.8	4	4.4	2.5	-	-

Fig. 3: Performance (ms) comparison of the optimized implementations generated by different compilers (autoscheduler of Halide for CPU, TensorFlow (TF) with XLA and cuDNN, TVM, autotuner of Tensor Comprehension for GPU, DNNVM for Deephi DPU on ZCU102@330 MHz, xfDNN of Xilinx on VU9P@450 MHz, and fpgaConvNet on ZC706@125 MHz) on our benchmarks (all batch = 1). * denotes that the result is achieved in the literature; others are tested by us onboard.

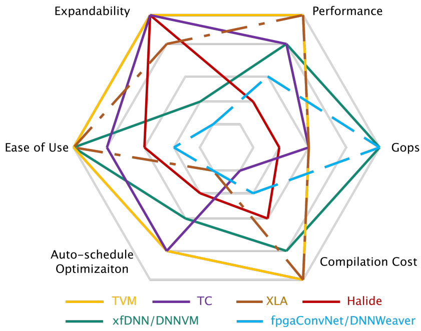

Fig. 4: Comparison of compilers according to our practical experience. The score increases (1 to 5) from the inside out. A higher score is better.

maps and weights to fit the stream buffer and filter caches to enhance data locality. Then, the compiler fuses adjacent operations to reduce the communication between the on-chip buffer and external memory. The sequence of implementations with many branches may affect the data layout and on-chip memory consumption. In DLA, the optimized schedule in DLA uses a priority queue to reduce on-chip buffer usage. In particular, DLA introduces some neural-network-dependent optimizations for ResNet[45].

xfDNN[17] and DNNVM[19] share similar framework design ideas, and they both construct an NN-independent hardware architecture. Their quantization tools transform the data into an 8-bit fixed number from a 32-bit float. The compilation tools compile and optimize the networks for efficient inference deployment. Due to relatively sufficient on-chip resources on Xilinx VU9P, xfDNN fuses many layers in the DNN model, and the entire network, schedule and weights need to be loaded only once onto the FPGA. This one-shot inference eliminates the CPU calls. DNNVM transforms the optimization problems of computing the graph generation, pipeline and data layout into graph-level problems. DNNVM transforms NN models from different deep learning frameworks into a uniform DAG named XGraph, then leverages heuristic subgraph isomorphism and shortest-path algorithms to traverse fusion opportunities and find the optimal execution strategies. Despite the frequent data communication between the on-chip BRAMs and off-chip DRAM, concurrent computation and data communication greatly alleviate the bandwidth saturation and improve the overall throughput of the system.

IV. Evaluation

As far as we know, no one has presented a fair comparison among the aforementioned compiler frameworks for deep neural networks. In this section, we analyze the performance of the optimized implementations generated by these compilers. According to our evaluation and practical experience, we hope to help developers choose from hardware platforms and compilation technologies for deep neural networks.

A. Experiment Environment

All our experiments are conducted on an Intel(R) Xeon(R) CPU E5-2687 W v4 (CPU) and a GeForce GTX TITAN X (GPU). Xilinx xfDNN is evaluated on VU9P (FPGA), and this service is provided by AWS Cloud. DNNVM is tested on ZCU102 (FPGA), which is provided by Deephi, now part of Xilinx. The results obtained from the literature are highlighted. We reimplement multiple operations of neural networks in the DSL of Halide, TC, TVM, etc. We set several mainstream DNNs[45]–[49] as the benchmarks, which include many typical operators.

B. Comparison of Different Compiler Frameworks

As shown in Figure 3 and Figure 4, we compare these compiler frameworks from different angles.

1) Performance, Gops/W: Performance means the throughput achieved on our benchmarks. Halide provides a very convenient DSL and an autoscheduler for CPU implementation optimization, while the GPU implementations can only be optimized manually. We find that Halide is 3-13x slower than TF and TVM on the CPU because Halide was originally designed for image processing, and the autoscheduler considers only a combination of parallelism, grouping and tiling. In addition, Halide's optimization algorithms are greedy, so some potentially valid opportunities may be missed, and the tile sizes are limited to several given numbers for narrowing the optimization space.

Obviously, CPU implementations generated by Halide, TF and TVM require 10-200x the time of GPU implementations because of the lack of parallelism. Due to multiple customized codes in TF, optimization across operators by XLA and assembly-level optimization in cuDNN, TensorFlow with XLA and cuDNN show the optimal performance compared with that of other implementations on the GPU. TVM integrates all of the optimization experience, provides favorable performance and supports most deep learning frameworks and hardware platforms at the same time. Compared with manual implementations in TF and the optimization experience needed by TVM, TC requires almost no manual experience or intervention for scheduling optimization. However, the scheduling strategies achieved through TC are dependent on the start point choice of the autotuner and the number of seeds and iterations in its genetic algorithm. In Figure 3, each operator in TC runs 1000 times with the best scheduling strategies that we trained, and the value we select represents the minimal time consumption.

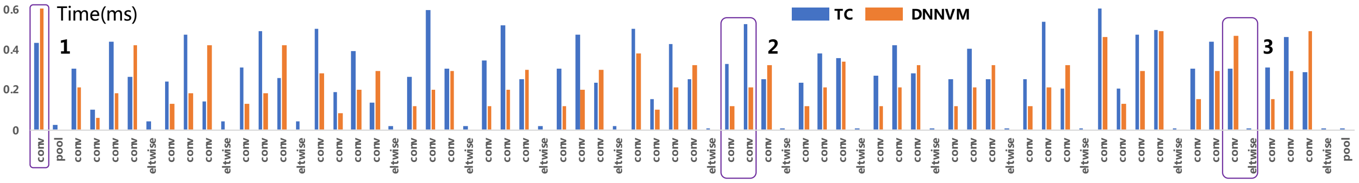

Fig. 5: Evaluation of optimized implementations generated by the TC autotuner for the GPU and DNNVM for the FPGA (ZCU102@ 330MHz) on ResNet50. We select the best autotuning results, and each operation is run 1000 times on the GPU for TC. The value we choose is the minimal time consumption of each operator.

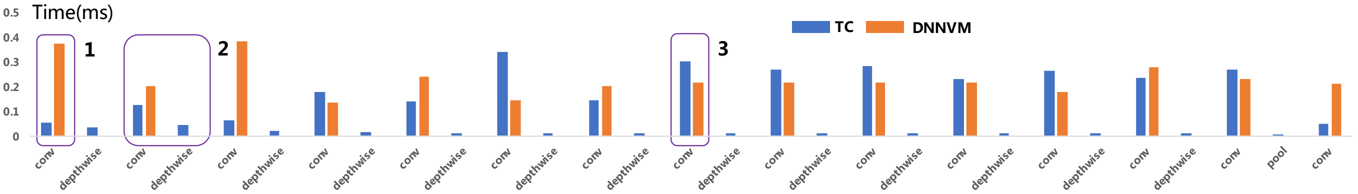

Fig. 6: Evaluation of optimized implementations generated by the TC autotuner for the GPU and DNNVM for the FPGA (ZCU102@ 330MHz) on MobileNet. In DNNVM, each depthwise convolution and pooling step is embedded into the previous convolution so that the single depthwise convolution is skipped.

Instructions generated by DNNVM and xfDNN (D&x) executed on FPGA platforms show comparable performance but 10x Gops/W relative to GPU implementations on the Nvidia GeForce Titan X. The hardware architectures of D&x are relatively fixed so that the compiler plays a key role in optimizing instructions. The hardware architecture of fpgaConvNet or DNNWeaver (f&D) is flexible and depends on the structure of the NNs. We find that D&x achieves 10x throughput over f&D. This performance gap comes from the following: first, the separate optimization for hardware design and compiler can be much easier than that of NN-dependent architectures. Second, the streaming architecture of f&D decomposes an NN into subgraphs and generates a hardware block for each subgraph, which means that the hardware resources are distributed for each subgraph so that each computation engine is constructed by fewer resources. Additionally, reconfiguration overhead may be introduced.

2) Compilation Cost: The autotuner of TC searches for optimized implementations for each operator with specific parameters and data shapes. Multiple iterations of the genetic algorithm and profiling process of all seeds cost hours or even days to train an optimized implementation for an NN. Compilers of f&D need to realize performance gains and generate hardware architecture through long compilations even with HLS tools. It takes several minutes for Halide to finish the autoschedule process for an NN. D&x spends dozens of seconds to generate optimized instructions.

3) Ease of Use: reflects the difficulty of running a specific NN on a processor. XLA, TVM and D&x are end-to-end compilers used to map an NN into instructions. Users need to transform NN models into TC and Halide manually, which may take some effort. The autoscheduler should also be initialized by the user in Halide. f&D requires basic knowledge for FPGA and HLS tools and can be more difficult to use.

4) Expandability: Expandability here means the difficulty of adding a new operation or an algorithm. It is extremely convenient to describe an operation using DSL provided by Halide, TVM and TC and map the operation into machine code. For f&D, a new algorithm necessitates a new combination of hardware blocks and optimization strategies. It takes a great deal of time to design a new computation engine for a new operator. D&x can reuse computation engines by different instructions to provide slightly easier use than that of f&D.

5) Autoschedule Optimization: We propose this property to describe the difficulty for users to optimize the schedule when given a newly designed operation or algorithm. TVM applies a machine learning algorithm to autotune the schedule. TC adopts polyhedral representations and explores the optimization space by an automatic random search and genetic algorithm. However, TVM and TC cannot achieve throughput equivalent to that of manually optimized implementations. The schedule of f&D is optimized on the hardware side. Designing and optimizing hardware is more difficult than optimizing instructions in D&x design.

C. GPU or FPGA?

As shown in Figure 5 and Figure 6, we used ResNet50 and MobileNet as a case study. We evaluated TC on the GPU and DNNVM on the FPGA. In Figure 5, DNNVM shows a 1.38x system throughput relative to TC. The performance improvement mainly comes from parts marked as 2 due to the highly optimized logic of the specialized accelerators. Unfortunately, we find two typical conditions marked as part 1 and part 3 in Figure 5, in which specialized accelerators achieve poor performance. To accelerate NNs on specialized accelerators, developers usually apply parallelism to several dimensions of data, such as the width, height, channel and kernel. In part 1, the channel of the input feature map of the first layer in the NN is 3, which cannot satisfy the parallelism degree. As a result, insufficient utilization of parallelism contributes to the inefficiency of the computation. In part 3, the operation of elementwise adding needs to load results from the previous 2 convolutions and leverages the dot-add function upon these data, which contributes to the bandwidth saturation. Although DNNVM embeds elementwise adding into the previous convolution to reduce data communication between the off-chip DRAM and on-chip buffers, the much smaller bandwidth of the FPGA relative to that of the GPU is still the main reason for the performance gap. Similarly, in Figure 6, part 1 and part 3 reveal the same condition as described above. In subgraphs such as that of part 2, the operation of depthwise convolution reduces the amount of weights and computation, which shows appealing performance on GPUs. However, this operation contributes to less data reuse. Although depthwise convolution is embedded into the previous operator in DNNVM, frequent data transportation and an unbalanced data load limit the overall performance.

V. Conclusion

In this article, we draw an in-depth comparison of compiler frameworks for DNNs from different angles. These compilers provide convenient DSLs and expressive IRs to represent NNs and pass information to the following optimization steps. Although the auto-optimization in compilers such as Hailde, TVM and TC may require substantial time to schedule pipelines and cannot achieve the acceleration rate of manual optimization in TF(XLA), they still present a high baseline performance when adding a new operation. It is still appealing to explore more efficient and effective algorithms for auto-schedule. Additionally, FPGA-based accelerators with DNNVM show comparable system performance to NNs and 10x Gops/s/W relative to GPUs. FPGA-based accelerators are more suitable for computation-intensive tasks, while GPUs are more suitable for bandwidth-bound operations.

VI. Acknowledgement

The authors gratefully acknowledge the support from TOYOTA, Xilinx and Beijing Innovation Center for Future Chips. This work was supported by National Key R&D Program of China 2018YFB0105005, National Natural Science Foundation of China(No. 61622403, 61621091), the project of Tsinghua University and Toyota Joint Research Center for AI Technology of Automated Vehicle(TT2018-01).

References

[1] *Tensorflow*, https://www.tensorflow.org/.
[2] *Caffe*, http://caffe.berkeleyvision.org.
[3] *Pytorch*, https://pytorch.org.
[4] Sharma and et al., “From high-level deep neural models to FPGAs,” in *MICRO*, 2016.
[5] Venieris and et al., “fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs,” in *FCCM*, 2016.
[6] S. Liu and et al., “Cambricon: An instruction set architecture for neural networks,” in *ISCA*, 2016.
[7] Jouppi and et al., “In-datacenter performance analysis of a tensor processing unit,” in *ISCA*, 2017, pp. 1–12.
[8] Ashouri and et al., “A survey on compiler autotuning using machine learning,” *CSUR*, vol. 51, no. 5, pp. 1–42, 2018.
[9] *Eigen*, http://eigen.tuxfamily.org/.
[10] *MKL*, https://software.intel.com/en-us/mkl.
[11] *Openblas*, http://www.openblas.net.
[12] Ragan-Kelley and et al., “Halide:a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” *Sigplan Notices*, 2013.
[13] T. Chen and et al., “TVM: An automated end-to-end optimizing compiler for deep learning,” in *USENIX & OSDI*, 2018.
[14] R. Wei and et al., “DLVM: A modern compiler infrastructure for deep learning systems,” *arXiv:1711.03016*, 2017.
[15] Vasilache and et al., “Tensor comprehensions: Framework agnostic high-performance machine learning abstractions,” 2018, https://arxiv.org/pdf/1802.04730.pdf.
[16] *XLA*, https://tensorflow.google.cn/xla/.
[17] *xfDNN*, https://github.com/Xilinx/ml-suite.
[18] M. S. Abdelfattah and et al., “Dla: Compiler and fpga overlay for neural network inference acceleration,” in *FPL*, 2018.
[19] Y. Xing and et al., “DNNVM : End-to-end compiler leveraging heterogeneous optimizations on FPGA-based cnn accelerators,” https://arxiv.org/pdf/1902.07463.pdf, 2019.
[20] *MXNet*, http://mxnet.incubator.apache.org.
[21] S. Girbal and et al., “Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies,” *IJPP*, vol. 34, no. 3, pp. 261–317, 2006.
[22] J. Roesch and et al., “Relay: A new ir for machine learning frameworks,” *MAPL*, 2018.
[23] L. N. Pouchet and et al., “Polly - polyhedral optimization in LLVM,” in *IMPACT*, 2011.
[24] *The LLVM compiler infrastructure*, https://llvm.org.
[25] *Cuda*, https://developer.nvidia.com/cuda-downloads.
[26] Munshi and et al., “Opencl,” 2011.
[27] Cyphers and et al., “Intel nGraph: An intermediate representation, compiler, and executor for deep learning,” 2018, https://arxiv.org/pdf/1801.08058.pdf.
[28] L. Tao and et al., “DaDianNao: A neural network supercomputer,” *TOC*, vol. 66, no. 1, pp. 1–1, 2016.
[29] Mullapudi and et al., “Automatically scheduling halide image processing pipelines,” *Acm Transactions on Graphics*, 2016.
[30] *Chainer*, https://chainer.org.
[31] *NVIDIA TensorRT*, https://developer.nvidia.com/tensorrt.
[32] L. Yizhi and et al., “Optimizing CNN model inference on CPUs,” 2018, https://arxiv.org/pdf/1809.02697.pdf.
[33] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” in *SIGKDD*, 2016.
[34] K. S. Tai and et al., “Improved semantic representations from tree-structured long short-term memory networks,” *arXiv:1503.00075*, 2015.
[35] T. Moreau and et al., “Vta: An open hardware-software stack for deep learning,” 2018, https://arxiv.org/pdf/1807.04188.pdf.
[36] L. Truong and et al., “Latte: A language, compiler, and runtime for elegant and efficient deep neural networks,” 2016.
[37] M. L. An and et al., “OoLaLa: An object oriented analysis and design of numerical linear algebra,” *Acm Sigplan Notices*, vol. 35, no. 10, pp. 229–252, 2000.
[38] F. Kjolstad and et al., “Taco: A tool to generate tensor algebra kernels,” in *ASE*, 2017.
[39] A. P. Harrison and D. Joseph, “Numeric tensor framework: Exploiting and extending Einstein notation.,” *Journal of Computational Science*, vol. 16, pp. 128–139, 2016.
[40] A. Jangda and U. Bondhugula, “An effective fusion and tile size model for optimizing image processing pipelines,” in *PPoPP*, 2018.
[41] Baghdadi and et al., “Pencil:a platform-neutral compute intermediate language for accelerator programming,” in *PACT’15*.
[42] S. Verdoolaege and G. Janssens, *Scheduling for PPCG*, Report CW 706, June 2017.
[43] S. Verdoolaege and et al., “Polyhedral parallel code generation for CUDA,” *TACO*, vol. 9, no. 4, pp. 1–23, 2013.
[44] E. Lee and et al., “Synchronous data flow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.
[45] K. He and et al., “Deep residual learning for image recognition,” in *CVPR*, 2016.
[46] K. Simonyan and et al., “Very deep convolutional networks for large-scale image recognition,” *Computer Science*, 2014.
[47] C. Szegedy and et al., “Rethinking the inception architecture for computer vision,” in *CVPR*, 2016.
[48] A. G. Howard and et al., “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *arXiv:1704.04861*, 2017.
[49] F. N. Iandola and et al., “Squeezenet: Alexnet-level accuracy with 50x fewer parameters and $<$ 0.5 MB model size,” *arXiv:1602.07360*, 2016.