On Retargeting the AI Programming Framework to New Hardwares

Jiacheng Zhao[1,2], Yisong Chang[1,2], Denghui Li[1], Chunwei Xia[1,2], Huimin Cui[1,2(✉)], Ke Zhang[1,2], and Xiaobing Feng[1,2]

[1] SKL Computer Architecture, Institute of Computing Technology, CAS, Beijing, China
{zhaojiacheng,changyisong,lidenghui,xiachunwei,cuihm, zhangke,fxb}@ict.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China

Abstract. Nowadays, a large number of accelerators are proposed to increase the performance of AI applications, making it a big challenge to enhance existing AI programming frameworks to support these new accelerators. In this paper, we select TensorFlow to demonstrate how to port the AI programming framework to new hardwares, i.e., FPGA and Sunway TaihuLight here. FPGA and Sunway TaihuLight represent two distinct and significant hardware architectures for considering the retargeting process. We introduce our retargeting processes and experiences for these two platforms, from the source codes to the compilation processes. We compare the two retargeting approaches and demonstrate some preliminary experimental results.

Keywords: Retarget · AI programming framework · FPGA · Sunway

1 Introduction

In recent years, AI has moved from research labs to production, due to the encouraging results when applying it in a variety of applications, such as speech recognition and computer vision. As the widespread deployment of AI algorithms, a number of AI processors [1,2] and FPGA accelerators [3,4] are proposed to accelerate AI applications meanwhile reducing power consumption, including DianNao [1], EIE [2], ESE [4], etc. Therefore, it is a significant issue for retargeting AI programming frameworks to different hardware platforms.

Some popular AI programming frameworks, e.g., TensorFlow/MXNet, have enhanced the fundamental infrastructure for retargetability using compiler technologies. In particular, TensorFlow introduces XLA [5] to make it relatively easy to write a new backend for novel hardwares. It translates computation graphs into an IR called "HLO IR", then applies high-level target-independent optimizations, and generates optimized HLO IR. Finally, the optimized HLO IR is compiled into a compiler IR, i.e., LLVM IR, which is further translated to

© IFIP International Federation for Information Processing 2018
Published by Springer Nature Switzerland AG 2018
F. Zhang et al. (Eds.): NPC 2018, LNCS 11276, pp. 39–51, 2018.
https://doi.org/10.1007/978-3-030-05677-3_4

machine instructions of various architectures using the compiler of the platform. Similarly, MXNet introduces NNVM compiler as an end-to-end compiler [6].

The evolving compiler approach significantly enhances the retargetability of AI programming frameworks. However, it still has a number of challenges. First, the non-compiler version is of the essence since it guarantees performance via directly invoking underlying high performance libraries. Therefore, maintaining the TensorFlow non-XLA and MXNET non-NNVM versions are necessary when retargeting the frameworks to a new platform. Second, the existing compiler approaches rely on LLVM backend of the AI processors, since the final binary code generation is implemented by the backend compiler. But for emerging AI processors especially designed for inference, vendors typically provide only library APIs without compiler toolchains. Therefore, it requires us to consider retargetability of non-compiler approaches for AI programming frameworks.

In this paper, we select one representative AI programming framework, TensorFlow, to present our experience of retargeting it to FPGA and Sunway TaihuLight. For FPGA, the architecture is the X86 CPU equipped with FPGA as an accelerator, thus we discuss how to add a new accelerator in TensorFlow. Meanwhile, we also design a set of software APIs for controlling FPGA in high-level C/C++ languages. For Sunway TaihuLight, the processor is a many-core architecture which has 260 heterogeneous cores. All these cores are divided into 4 core groups (CG), with each CG including a big core and 64 little cores. Sunway can be regarded as a chip integrating CPUs (big cores) and accelerators (little cores), thus we discuss how to change the CPU type in TensorFlow. In this paper, we respectively discuss how to retarget TensorFlow to these two distinct architectures, and present some preliminary experimental results on FPGA and Sunway TaihuLight. We wish this paper can be helpful for programming framework developers to retarget TensorFlow to other newly designed hardwares.

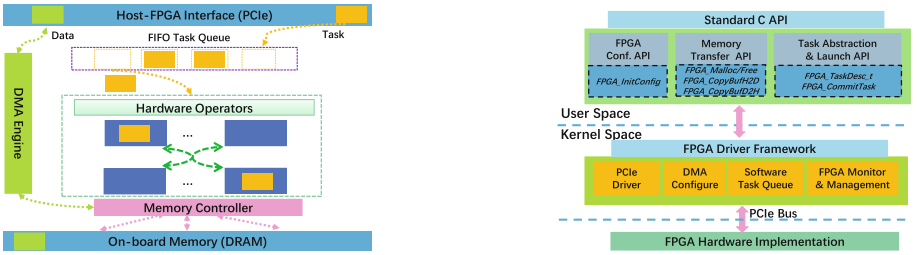
The rest of this paper is organized as follows: Sects. 2 and 3 discuss how to retarget TensorFlow to FPGA and Sunway TaihuLight respectively. Section 4 demonstrates experimental results. Section 5 discusses differences of retargeting to FPGA and Sunway. Section 6 discusses the related work. Section 7 concludes.

2 Retargeting TensorFlow to FPGA

2.1 FPGA Execution Model

A representative approach to utilizing FPGA is Amazon EC2 F1 [3], which is a compute instance with FPGA that users can program to create custom accelerators for their applications. The user-designed FPGA can further be registered as an Amazon FPGA Image (AFI), and be deployed to an F1 instance.

We also follow the design rule for leveraging FPGA in AI programming frameworks. In particular, we create an abstract execution model for FPGA, and provide a set of APIs for the developers to register an FPGA into the system. In this paper, we use the naive first-in-first-out policy (FIFO) to model the FPGA execution, shown in Fig. 1a. Furthermore, the task execution on our FPGA is



(a) Execution model of FPGA. (b) APIs of FPGA.

Fig. 1. Abstract execution model and APIs of target FPGA accelerators.

non-preemptive. Our current execution model is similar with GPU kernel execution (without streams). Certainly designers can create different execution models for FPGA, and TensorFlow runtime shall be adjusted correspondingly.

2.2 FPGA APIs and Implementation

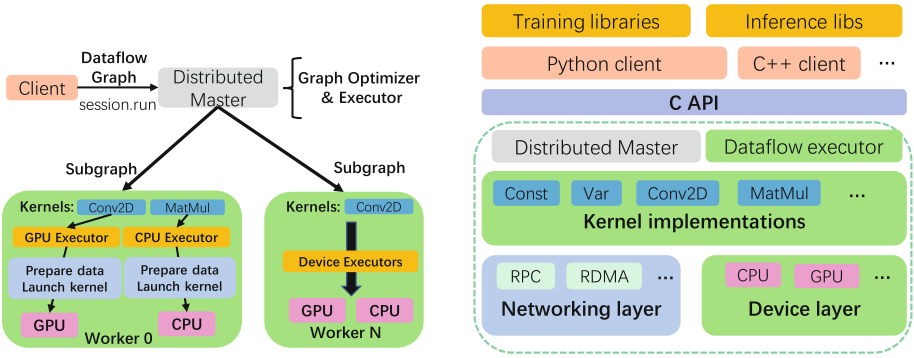
Furthermore, we also provide a set of abstract APIs for accessing FPGA accelerators. The abstract APIs are designed to be standard C functions and data structures, as shown in the top part of Fig. 1b. The APIs are:

- *FPGA_InitConfig*. FPGA resource initialization and configuration.
- *FPGA_Malloc/Free*. FPGA memory management.
- *FPGA_CopyBufH2D*. Copy data from host to device, using DMA.
- *FPGA_CopyBufD2H*. Copy data from device to host, using DMA.
- *FPGA_TaskDesc_t*. Data structure for FPGA task description.
- *FPGA_CommitTask*. Commit a task to FPGA.

The APIs are implemented in the operating system (middle part of Fig. 1b) and user-space libraries (top part of Fig. 1b) coordinately. User-space libraries encapsulate the FPGA accelerators into APIs, based on interfaces provided by the FPGA driver framework. The FPGA driver framework interacts with FPGA hardware via PCIe bus, and consists of four functional components: "PCIe Driver" for handling PCIe device registration and interrupts, "DMA Configure" for DMA memory transfer requests, "Software Task Queue" for FIFO execution model and"FPGA Monitor & Management" for monitoring and managing FPGA devices, such as querying FPGA states and task status.

2.3 TensorFlow Architecture for Supporting Retargetability

Figure 2a illustrates how TensorFlow executes user-defined dataflow graphs. When the session manager receives a message of *session.run()*, it starts the "computational graph optimization and execution" module, which automatically partitions the dataflow graph into a set of subgraphs, and then assigns the subgraphs to a set of worker nodes.



(a) TensorFlow dataflow graph execution. (b) Architecture of TensorFlow [7].

Fig. 2. TensorFlow architecture and its execution of user-defined dataflow graphs.

The execution of subgraphs is managed by "dataflow executor", which is local to one worker node where the subgraphs are assigned to. The dataflow executor schedules operations in subgraphs to the underlying devices. Dataflow executor prepares the input and output data for each kernel invocation, launches the specific kernel via device executor (e.g. CPU/GPU Executor in Fig. 2a).

Figure 2b further depicts the overall architecture of TensorFlow framework. The modules related to retargeting are: "Device Layer", "Dataflow Executor" and "Kernel Implementation". "Device Layer" aims to provide proper abstraction of FPGA resources and launching FPGA tasks. "Dataflow Executor" should be aware of the FPGA devices and be able to assign operations to them, and "Kernel Implementation" is the fundamental operation kernels on the FPGA.

2.4 Supporting FPGA in TensorFlow

Step 1. FPGA Device Abstraction. First, we add the FPGA device into the device layer. Two important issues are addressed here:

Memory Management: FPGA accelerators are commonly equipped with DDR memory to hold input/output features and/or weights. This memory is treated as a memory pool in our work and C-style memory management scheme is provided. Thus, four critical routines: *memcpyDeviceToHost*, *memcpyHostToDevice*, *malloc*, and *free* are implemented using APIs provided in Sect. 2.2.

Execution Model: Execution model determines how TensorFlow runtime interacts with underlying devices and must match the nature of corresponding devices. The abstracted FPGA in this paper is a synchronous FIFO device. An FPGA executor is implemented using APIs defined in Sect. 2.2.

Step 2. FPGA Device Runtime. Second, runtime support for the new FPGA device will be implemented, including the kernel launching and high level memory management wrapper.

Kernel Launching. In TensorFlow, the dataflow executor assigns operations to specific device by invoking the *Compute* method of corresponding device, which is set to launch the *Compute* function of the given kernel.

High-Level Memory Management Wrapper. The device abstraction provides low-level C-style memory management API. And TensorFlow runtime requires high-level APIs to deal with tensor data. In particular, a 'best-fit with coalescing' memory allocator, *FPGABFCAllocator*, is provided to serve the tensor data allocation/free of TensorFlow runtime. Furthermore, two high-level APIs, *CopyCPUTensorToDevice* and *CopyDeviceTensorToCPU*, are implemented to manipulate tensor data, instead of raw data.

Besides, a factory class, namely "FPGADeviceFactory" is provided to create and instantiate instances of "FPGADevice".

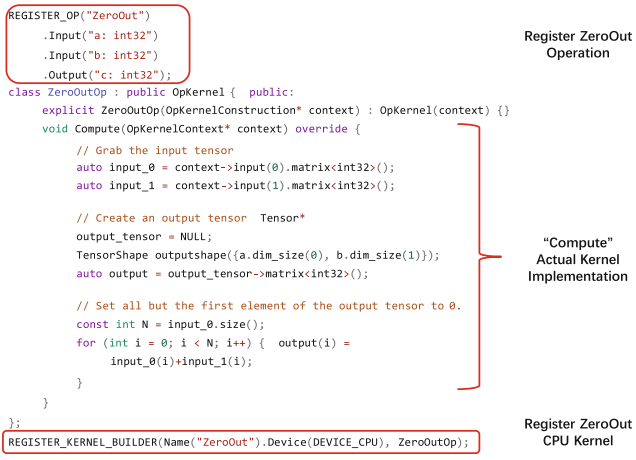


Fig. 3. An example of implementing an operation in TensorFlow.

Step 3. FPGA Kernel Implementation. Figure 3 shows an example operation in TensorFlow, where the `Compute` function takes the input tensor parameters, the target device, and the context. All the input parameters are encapsulated into the data structure of `OpKernelContext`.

When defining an operation, its specific implementation on a device is called a *kernel*, which is typically implemented as libraries. For example, most CPU kernels are implemented via Eigen libraries [8], and most GPU kernels are implemented via CUBLAS or CUDNN libraries. Therefore, when we introduce FPGA for acceleration, we first define the implementation of operations on FPGA, which translates to function calls to *FPGA_CommitTask* defined in FPGA APIs. After implementing an operation on a new device, we should register the new implementation into TensorFlow, using the `REGISTER_OP` and `REGISTER_KERNEL_BUILDER`.

Figure 3 shows an example for registering a new operation `ZeroOut`, which has two input tensor parameters `a` and `b`, and generates one output tensor `c`. We specify these information in `REGISTER_OP` and implement the operation in `OpKernel`. Finally, `REGISTER_KERNEL_BUILDER` is used for registering the kernel.

3 Retargeting TensorFlow to Sunway

In this section, we first briefly introduce the architecture of Sunway processor, and then present our retargeting process.

3.1 Sunway Architecture

Sunway 26010 processor [9] is composed of 4 core groups (CGs) connected via an NoC. Each CG includes a Management Processing Element (MPE) and 64 Computing Processing Elements (CPEs) arranged in an 8 by 8 grid. MPE and CPE cluster in one CG share same memory space. All the MPEs and CPEs run at the frequency of 1.45 GHz.

On the software side, Sunway uses a customized 64-bit Linux with a set of compilation tools, including native C/C++ compiler and cross compiler.

Aiming at Sunway processor, we regard MPEs as CPUs and leverage CPEs for acceleration. However, the MPEs and CPEs share same memory space, making it pointless to transfer data between them. Thus, we firstly retarget the TensorFlow framework which runs on CPUs to the Sunway MPEs, and then CPEs for acceleration in the retargeted TensorFlow.

3.2 Compiling TensorFlow for Sunway MPEs

We have two ways to compile TensorFlow for Sunway. The first is to use the native compiler of Sunway nodes by submitting compilation process as a job for Sunway. The second is to cross-compiler TensorFlow on an X86 server. We select cross-complication, since the native compiler is too restricted to compile the large-scale complex TensorFlow source codes. We met a series of obstacles during the retargeting process, and we discuss them here for providing some experience of porting a large scale software package to Sunway TaihuLight.

Static Linked Library. First, Sunway TaihuLight does not support dynamic linked library when CPEs are expected to be used. Therefore, we choose to cross-compile TensorFlow into a static linked library, i.e., libtensorflow.a.

The Bazel Compilation Tool. TensorFlow is configured to use Bazel as its default compilation tool, which can generate dynamic linked library, but does not work well for generating static linked library. Meanwhile, a number of unexpected problems raised when using the cross compiler `swgcc` in Bazel. Therefore, we switch to use Makefile as our compilation tool.

The Python Support. TensorFlow is tightly coupled with the language of Python, which is not supported on Sunway TaihuLight. A number of modules utilize

Python-based tools, such as *tf.train* and *tf.timeline*. Therefore, we decouple these modules from the TensorFlow framework. As a result, our retargeted TensorFlow on Sunway TaihuLight only supports C++ programming interface, without support for the Python binding.

Processing Protobuf. The Protobuf tool `protoc` is used both during the compilation of TensorFlow (on X86 platform), and during the execution of TensorFlow (on Sunway TaihuLight platform). For such purpose, `protoc` is required to be compiled on x86 platform using X86 native `gcc` and cross compiler `swgcc`.

Two-Phase Compilation. The compilation of TensorFlow is a two-phase compilation. In the first phase, the X86 `gcc` compiler is used to generate some tools for X86 platform, e.g., the X86 `protoc`, which reads the *.pb files in TensorFlow source code and generates the corresponding C++ files. In the second phase, the cross compiler `swgcc` is used to generate the final libtensorflow.a. During this phase, all dependent libraries should be switched to the static linked versions, e.g., protobuf, libstdc++, libm, etc.

After TensorFlow is cross-compiled successfully, it can run on the MPEs of Sunway TaihuLight. Since Python module like *tf.train* is disabled, the ported TensorFlow does not support training.

Now we have had a baseline TensorFlow which completely runs on the MPEs of Sunway. The operations can be implemented following steps in Sect. 2.4. Next we add CPEs for acceleration. Specially, MPEs are responsible for graph creation and optimization, together with task creation and scheduling. Meanwhile CPEs can execute the computation-intensive kernels, e.g. convolutions.

3.3 Using CPEs for Acceleration

We have two approaches for using CPEs. First, we can force the CPU kernel implementation to invoke CPE libraries, which means MPEs and CPEs are considered together as one device. Alternatively, we can consider CPEs as individual accelerators, similar with GPUs and FPGAs. In this paper, we select first approach as the second approach has been discussed in Sect. 2.

To use CPEs in an operation, consider the steps described in Fig. 3. Take `matmul` for example, the original implementation will use Eigen as the math library in *Compute* part. We will change the math library from Eigen to SWCBLAS library, i.e. from Eigen call *MatMul<CPUDevice>* to *sgemm/dgemm* call in SWCBLAS. As SWDNN library is being developed, we only use SWCBLAS for implementing the operations in this work. When SWDNN is released, we can use the same approach to change the library from SWCBLAS to SWDNN.

4 Evaluation

We select four DNN models, i.e., Cifarnet [10], Lenet [11], Inception-V3 [12] and Resnet-50 [13], to evaluate our retargeted TensorFlow on FPGA and Sunway

TaihuLight. The trained models are obtained from TensorFlow model zoo. We only focus on inference phase. Our experimental results demonstrate that our retargeted TensorFlow can run correctly on FPGA and Sunway platforms.

The functionality of retargeted TensorFlow relies on underlying operation kernels. For the aforementioned four DNN models, CPU and Sunway MPE support all seven main operations: *Conv2D*, *BiasAdd*, *Pooling*, *Relu*, *Softmax*, *Matmul*, *FusedBatchNorm*. Our FPGA doesn't implement *FusedBatchNorm*, which means it can't support Inception-V3 and Restnet-50. Sunway CPE supports only *Conv2D*, *Softmax* and *Matmul*. Other operations can be easily supported once SWDNN is deployed.

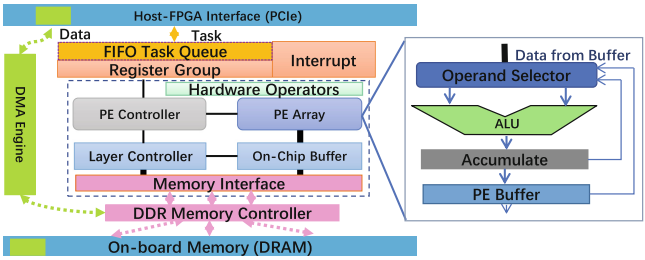


Fig. 4. Evaluated hardware of target FPGA accelerator.

4.1 Hardware Platforms

FPGA Implementation: We implement a custom PCIe-attached acceleration card based on a Xilinx Virtex-7 690T FPGA chip as shown in Fig. 4. The card communicates with host CPU via the standard PCIe Gen 3 × 8 interconnect. We leverage dual off-chip DDR3-1600 SODIMMs with total capacity of 8 GB as device memory. Xilinx Vivado 2016.4 toolset is used and the synthesized core accelerator logic and DMA engine operate at the frequency as high as 200 MHz.

Figure 4 further illustrates the design of our FPGA accelerator. For details, we implement a unified hardware template of DNN accelerator with a configurable number of processing elements (PEs) for per layer specific operations, like convolution and full-connection. The processing element is composed of a 1-D array of multiply-and-accumulation (MACC) units, loop tiling and unrolling are leveraged to partition computation into specific PEs. An on-chip buffer is also implemented to hold tiled input feature map. To reduce the external memory bandwidth, temporary results are pushed into the PE buffer. Data movements between PE array and on-chip buffer is elaborately controlled by the PE controller according to the loop unrolling and tiling strategies.

Sunway TaihuLight: The Sunway TaihuLight is described in Sect. 3, and we use one node for evaluation. As we focus on the inference, the number of nodes does not matter.

Baseline Platforms: For comparison, we also run these models on a CPU and nVIDIA GPU. In particular, the CPU is Intel Xeon E5-2620 which runs at 2.0 GHz and has a main memory of 32 GB. The nVIDIA GPU is Tesla K40c which has the frequency of 745 MHz, and the global memory is 12 GB.

4.2 Results on FPGA Platform

With our retargeted TensorFlow, programmers can use the "with tf.device ("fgpa:0")" statements to use the FPGA, with no modifications in their source codes.

Figure 5 shows the overall execution time (data transfer time included) of Cifarnet and Lenet on FPGA, CPU and GPU. In this paper, we focus on retargeting process, thus the underlying FPGA implementation is not optimized.

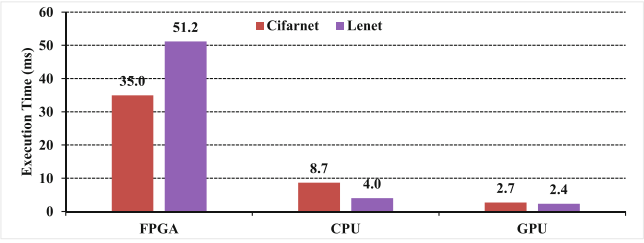


Fig. 5. Overall execution time of Cifarnet and Lenet on FPGA, CPU and GPU.

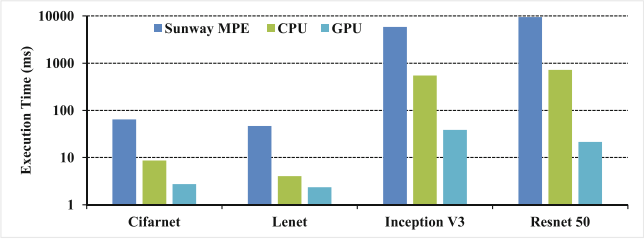


Fig. 6. The overall execution time on Sunway MPE, CPU and GPU.

4.3 Results on Sunway TaihuLight Platform

As we treat Sunway MPE and CPEs as a CPU, the source codes needs no modification and the models can be directly executed on the ported TensorFlow.

Figure 6 shows the overall execution time when using only MPE, in comparison with CPU and GPU. Note that the vertical axis is in log scale. Besides, we use only one core of Sunway and CPU, frequency of which are 1.45 GHz and 2.0GHz respectively. Therefore, Sunway MPE performs worse than CPU.

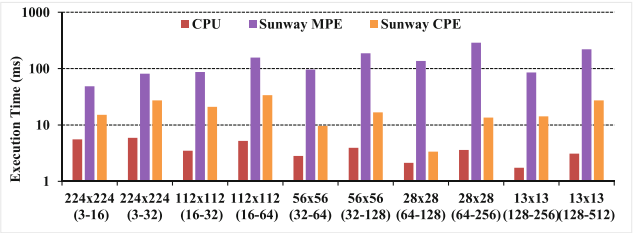


Fig. 7. Performance of convolution with 3×3 filter size.

Figure 7 demonstrates the execution time of one convolution operation (with the filter size of 3×3) on Sunway MPEs and CPEs, in comparison with CPU. We don't evaluate the overall execution as some operations are not supported on CPEs. The horizontal axis marks different scales of input feature sizes and input/output channel numbers, e.g. $224 * 224 * (3 - 16)$ means input feature size is $224 * 224$ while input channel is 3 and output channel is 16. The vertical axis is execution time in log scale. The results show that CPEs can obtain significant performance improvement, up to 45 times than MPE. Furthermore, in our experiments, only one core group is leveraged (the reason is that the SWCBLAS interface is designed for one core group). The performance is expected to be improved when all core groups are utilized and SWDNN is released.

5 Discussion and Future Work

We have discussed two types of TensorFlow retargeting processes, i.e., FPGA and Sunway TaihuLight. In particular, FPGA represents the approach of introducing a new accelerator into TensorFlow while Sunway TaihuLight represents the approach of changing the CPU architecture in TensorFlow.

Retargeting to a New AI Accelerator. Most of emerging AI processors will be deployed as accelerators. Thus, our experience of retargeting to FPGA can apply for such scenarios. The modification for the device layer is the same with the process for FPGA. The runtime support shall be designed by vendors of AI processors, in corresponding to their execution model. Furthermore, amount of work is needed for implementing hundreds of operation kernels. Even if most AI processors will provide machine learning libraries, porting these operation kernels are still time-consuming. We will further explore automatic kernel generation.

Exploiting the Computation Ability of Sunway TaihuLight. Sunway TaihuLight exhibits performance potential for machine learning, e.g., some preliminary work on SWDNN [14] has been released. To enable more machine learning programs, especially model training, to run on Sunway TaihuLight, a more robust TensorFlow is necessary. Thus, we will further consider following issues, i.e., Bazel compilation tool, Python support, and stable SWDNN library.

Data Layout Issue. Moreover, the data layout is a significant issue for the framework developers. For example, TensorFlow stores the tensor with the default format of NHWC. But NCHW is the default format for GPU libraries, e.g. cudnn [15], making it the framework's burden to transform between them. Sunway TaihuLight has not finally determined its data layout in SWDNN. When TensorFlow is retargeted to a new platform, data format shall be designed by taking hardware and/or library into consideration.

6 Related Work

In recent years, AI has drawn many interest from both researchers and industry, especially DNNs (Deep Neural Networks [12,13,16,17]). Despite the enormous advance in AI algorithms, researchers have also done extensive work to meet the performance/energy/programming requirements of DNN applications.

First, from the aspect of software, a huge number of software tools are proposed to enable flexible programming of DNN applications, such as TensorFlow [18], Caffe [19], and MXNet [20]. All these tools support general purpose CPU and high performance nVIDIA GPU, both of which have mature compiler toolchains [21] and highly optimized libraries [15].

Second, from the aspect of hardware, a series of domain specific accelerators [1,2,22,23] are explored. DianNao [1] leverages loop tiling to efficiently reuse data and supports both DNNs and CNNs. EIE [2] focus on inference for compressed DNN models. Furthermore, researchers also explore FPGA as accelerators [4,24,25] for DNN applications. And to the best of our knowledge, all these accelerators lack mature compiler toolchains, for example, a C compiler.

At last, it is becoming a big challenge to utilize these diverse hardware accelerators in software tools. TensorFlow proposes XLA [5], which leverages compiler technology to transform high-level dataflow graph to compiler intermediate representation, i.e. LLVM IR, relies on hardware-specific backend to generate binary code, e.g., NVPTX for nVIDIA GPU. Similarly, MXNET introduces NNVM [6], which also makes use of compiler backend. However, these compiler-based approaches require a mature compiler backend, which is rarely seen in AI processors. Thus, this work explores non-compiler approach of retargeting software frameworks to diverse AI hardwares. Besides, [26] proposes a NN compiler to transform a trained NN model to an equivalent network that can run on specific hardwares, which sheds some light on automatic retargeting of AI frameworks.

7 Conclusion

We have presented our experience of retargeting TensorFlow to different hardwares, e.g. FPGA and Sunway, together with some preliminary evaluation results using popular DNN models. We have investigated the differences between FPGA and Sunway with respect to retargeting.

Acknowledgments. This work is supported in part by the National Key R&D Program of China (2016YFB1000402), the National Natural Science Foundation of China (61802368, 61521092, 61432016, 61432018, 61332009, 61702485). The authors would like to thank all the anonymous reviewers for their valuable comments and helpful suggestions.

References

1. Chen, T., et al.: DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In: ASPLOS 2014, NY, USA. ACM, New York (2014)
2. Han, S., et al.: EIE: efficient inference engine on compressed deep neural network. In: ISCA 2016 (2016)
3. Amazon EC2 F1. https://aws.amazon.com/cn/ec2/instance-types/f1/
4. Han, S., et al.: ESE: efficient speech recognition engine with sparse LSTM on FPGA. In: FPGA 2017 (2017)
5. Tensorflow XLA. https://www.tensorflow.org/performance/xla/
6. Li, M.: Introducing NNVM compiler: a new open end-to-end compiler for AI frameworks (2017)
7. Tensorflow architecture. https://www.tensorflow.org/extend/architecture
8. Guennebaud, G., Jacob, B., et al.: Eigen v3 (2010). http://eigen.tuxfamily.org
9. Lin, H., et al.: Scalable graph traversal on sunway taihulight with ten million cores. In: IPDPS 2017 (2017)
10. Krizhevsky, A.: Learning multiple layers of features from tiny images (2009)
11. Lécun, Y., Bottou, L., Bengio, Y., Haner, P.: Gradient-based learning applied to document recognition. In: Proceedings of the IEEE (1998)
12. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the inception architecture for computer vision. CoRR vol. abs/1512.00567 (2015)
13. He, K., Zhang, X., Ren, S., Sun, J.: Identity mappings in deep residual networks. CoRR vol. abs/1603.05027 (2016)
14. Fang, J., Fu, H., Zhao, W., Chen, B., Zheng, W., Yang, G.: swDNN: a library for accelerating deep learning applications on sunway taihulight. In: IPDPS 2017 (2017)
15. Chetlur, S., et al.: cuDNN: efficient primitives for deep learning. CoRR vol. abs/1410.0759 (2014)
16. Lecun, Y., Bottou, L., Bengio, Y., Haner, P.: Gradient-based learning applied to document recognition. In: Proceedings of the IEEE, pp. 2278–2324, November 1998
17. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. In: NIPS 2012 (2012)
18. Abadi, M., et al.: TensorFlow: a system for large-scale machine learning. In: OSDI 2016 (2016)
19. Jia, Y., et al.: Caffe: convolutional architecture for fast feature embedding. In: MM 2014, pp. 675–678 (2014)
20. Chen, T., et al.:, MXNet: a flexible and efficient machine learning library for heterogeneous distributed systems. CoRR vol. abs/1512.01274 (2015)
21. Nvidia Corporation: Nvidia cuda C programming guide. Nvidia Corporation (2011)
22. Chen, Y.-H., Emer, J., Sze, V.: Eyeriss: a spatial architecture for energy-efficient dataflow for convolutional neural networks. In: ISCA 2016 (2016)
23. Parashar, A., et al.: SCNN: an accelerator for compressed-sparse convolutional neural networks. In: ISCA 2017 (2017)

24. Suda, N., et al.: Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In: FPGA 2016 (2016)
25. Qiu, J., et al.: Going deeper with embedded FPGA platform for convolutional neural network. In: FPGA 2016 (2016)
26. Ji, Y., Zhang, Y., Chen, W., Xie, Y.: Bridge the gap between neural networks and neuromorphic hardware with a neural network compiler. In: ASPLOS 2018 (2018)