An Automated Compiler for RISC-V Based DNN Accelerator

Zheng Wu[1], Wuzhen Xie[1], Xiaoling Yi[1], Haitao Yang[1], Ruiyao Pu[1], Xiankui Xiong[3,4], Haidong Yao[3,4], Chixiao Chen[1,2], Jun Tao[1] and Fan Yang[1*]
[1]*State Key Lab of ASIC & System, School of Microelectronics, Fudan University, China*
[2]*Frontier Institute of Chip and Systems, Fudan University, China*
[3]*ZTE Corporation, China*
[4]*State Key Laboratory of Mobile Network and Mobile Multimedia Technology, China*

2022 IEEE International Symposium on Circuits and Systems (ISCAS) | 978-1-6654-8485-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/ISCAS48785.2022.9937645

***Abstract*—Multifarious hardware accelerators are developed for the widely used Deep neural networks (DNN). Nowadays the SoCs composed of a general processor and a coupled accelerator are becoming prevalent. Compared to the specialized DNN accelerator for one specific DNN, this kind of coupled architecture is programmable and supports diverse DNNs. However, for the low-level programming interface of the co-processor-like accelerator and the multi-hierarchy memory structure, programming for the DNN accelerator is not easy work. Meanwhile, there are a couple of tensor compilers that deploy the DNN on various hardware. In this work, we combine the flexibility of the tensor compiler and the high efficiency of the hardware accelerator by proposing an automated compiler that can compile tensor programs and generate high-performance programs for programmable DNN accelerators. Our compiler is based on TVM [1] and target at Rocket Chip Coprocessor (RoCC) [2]. The compiler is flexible and supports many kinds of RISC-V instructions. The programmer can define the hardware constraints in the proposed compiler which makes the generated code more efficient. Our compiler can lower the program with the ping-pong strategy and the generated code can achieve 26% speed up compared to the baseline.**

***Index Terms*—RISC-V, Accelerator, Automated Compiler, TVM, Ping-pong buffering.**

I. Introduction

With the demand of the industry, machine learning is required in more and more circumstances. In the past, big workloads like deep neural networks has to be implemented on the server. The long-distance data transformation leads to power consumption and high latency. However, with the progress of IC technology, some computation tasks can be deployed on the edge side. Nowadays, there are more and more accelerators on FPGA for deep learning interfere workloads being developed [3] [4] [5]. These accelerators provide good performance and acceptable power consumption.

Rocket Chip [2] is an SoC generator with high extension. It can generate configurable RISC-V CPU coupled with an accelerator through the Rocket Custom Coprocessor interface, also known as RoCC [2]. Many novel designs of DNN accelerators have been made such as Gemmini [6] and vector extensions [7].

TVM [1], a deep learning compiler, which can deploy neural networks on multiple targets efficiently, is a good tool for us to customize the execution of neural networks on the targets. TVM can generate low-level code targeted at CPU, GPU, and AI accelerators. However, TVM now doesn't have an official backend for RISC-V and its extensions. But the researchers have developed an interface between TVM and Spike [8], a simulator of RISC-V. This software toolchain names DLR [9], which uses the LLVM backend of TVM. DLR can generate executable programs to run on Spike. The codegen in [10] proposed a code generation flow for Gemmini, a simple RoCC accelerator. It provides a way of using TVM RPC to evaluate the performance of the generated low-level code on Gemmini.

The accelerator using the ping-pong buffering strategy is becoming prevalent, which creates a pipeline of instruction execution. When the computing unit is calculating the previous data block, the next data block can be prepared and sent to another input buffer. The programmable accelerator with ping-pong strategy requires the low-level program efficient enough to cross send data to the ping-pong buffer. The accelerator in [11] used a ping-pong buffer to store the result of the computation unit and reduce the execution time at most 1.5%.

In this paper, we present a general RISC-V based accelerator with a specific ping-pong buffering strategy. We propose an automated compiler based on the C backend of TVM for this accelerator. The compiler can generate high-performance low-level code using ping-pong strategy which can accelerate convolution operators at most 26%.

The rest of this paper is organized as follows. Section II reviews the background of RISC-V and TVM. In section III, we introduce our DNN accelerator and the proposed compiler. The experiment results are presented to specify how the speed up of the ping-pong buffering strategy in section IV. At last, we conclude the paper in section V.

II. Background

A. RISC-V

RISC-V is an open-sourced Instruction Set Architecture (ISA) designed by the researchers from UC Berkeley and designed from the ground up to be clean, microarchitecture-agnostic, and highly extensible. RISC-V has SIMD instructions that make full use of the registers. Based on RISC-V, the Rocket Chip [2], which is a design generator, is developed. Rocket can produce many design instances for an SoC design. Variable functional digital circuits generated by Rocket have been manufactured. Rocket Chip is programmed with Chisel [12], which is also an open-source hardware description language developed by researchers in Berkeley. It is a domain-specific language (DSL) built on the Scala language and supports highly parameterized hardware generators.

The Rocket Chip Coprocessor (RoCC) is a special feature of the Rocket Chip SoC generator to integrate custom coprocessors into SoC. RoCC defines the interface between Rocket Chip and the accelerator. It provides a decoupled communication plan for Rocket core and the coprocessor. The Rocket core can send custom instructions to the interface to drive the coprocessor. The coprocessor can access the Float Point Unit and other peripherals of the SoC. Also, the coprocessor can interrupt the core through the interface. At last, the memory access for the accelerator can be fast and coherent via TileLink [13] on-chip bus or another simple interface to the CPU L1 cache.

*Corresponding author: {yangfan}@fudan.edu.cn

B. TVM

Nowadays neural networks are needed for more and more devices. But most of the deep learning frameworks are aimed at GPU or CPU. To bring neural networks to various devices conveniently, TVM was developed and brought us a tool to map DNN workloads on CPU, GPU, and specialized accelerators.

TVM is an automated end-to-end optimizing compiler for deep learning. It supports common deep learning frameworks. Users can import their trained model in TVM and use TVM to optimize the execution of the neural network. The compilation flow of TVM is illustrate in Fig.1. Relay [14] is an intermediate representation(IR) for expressing deep learning models and it is integrated into TVM. A neural network in terms of relay IR can be represented as a computational graph. TVM offers plenty of configurable passes to optimize computation graphs, such as operator fusion and data layout transformation. The relay model can then be compiled into an executable library file if TVM supports this backend.

To deploy a neural network on a new backend, programmers need to define the execution of operators for this new backend. Understanding the hardware structure is a burden for programmers. Halide [15] brought a way to separate the computation and schedule. The computation process of operators can be defined using the *lambda expression*, which can be lowered to loop nests. Then the programmers can use schedule structure to optimize the loop nests. TVM takes this paradigm to ease the burden of programmers and support more backends. As for scheduling, TVM provides many primitives. There are primitives like *tile*, *fuse* and *split* to change the structure of the loop nests. Meanwhile, TVM intrinsic like *tensorize* or *vectorize* can substitute loop nests with handcrafted kernels. Also, TVM provides *pragma* Application Programming Interface(API) for developers to customize their primitives. Programmers can use *pragma* to insert more accelerator instruction in the program.

With TVM Pass Infrastructure, programmers can define their optimized pass to improve the performance metrics of models for specific devices. TVM provides an approach to run the TVM model on remote devices via RPC. In addition, AutoTVM offers a way to tune models and operators by providing a template schedule, and searching the parameter space defined by the template [16].

III. Automated Compiler for RoCC Accelerator

In this section, we will first introduce the architecture of the accelerator and its instruction set. Then, we will focus on the automated compiler for the presented RoCC accelerator and the speed-up of the ping-pong buffering strategy.

A. Accelerator Architecture

We first briefly describe the RoCC accelerator we will deploy our program on. The accelerator implements General Matrix Multiplication(GEMM) with a systolic array structure. This GEMM unit can perform the multiplication of two 8*8 matrices in one clock cycle. With the extension of macro instruction, it can perform the multiplication of two matrices of any size. The accelerator can deal with injective computation with Vector Engine (VE). The VE can perform several kinds of computation, including addition, subtraction, multiplication, left and right shift, maximum, minimum, and some logical operator. As for the storage, the accelerator includes many buffers around the computing units. The Ping-pong buffering strategy is used on both the input side and the output side of the GEMM unit. There are also large buffers on the RoCC accelerator to store intermediate results. The Direct Memory Access (DMA) engine handles the data movement on the accelerator and the data communication between the RoCC accelerator and the Rocket core. The overview of the RoCC accelerator is illustrated in Fig. 2.

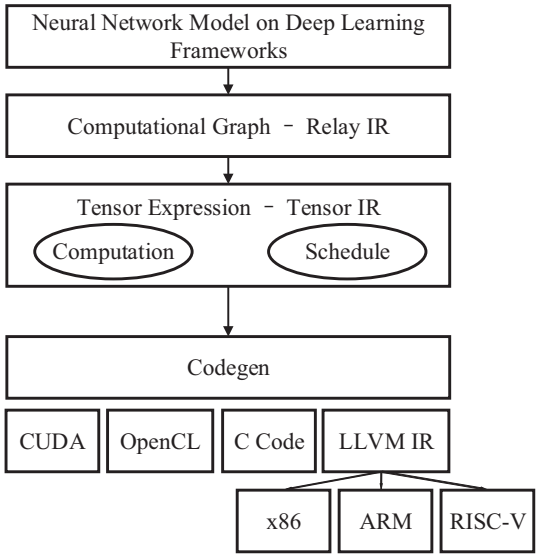


Fig. 1: The system overview of TVM. It provides an end to end compiler stack from trained DNN model to executable program.

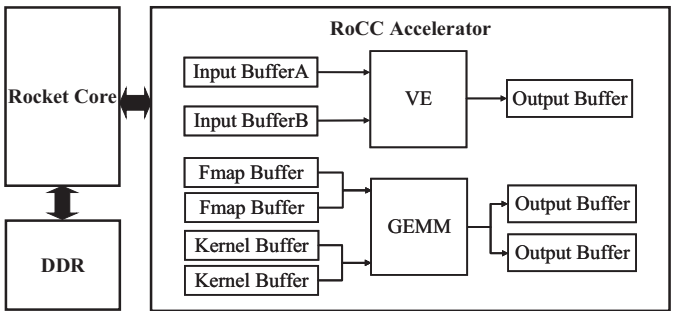


Fig. 2: The overview of the RoCC accelerator consisted of the Rocket Core, the RoCC accelerator, and the DDR memory.

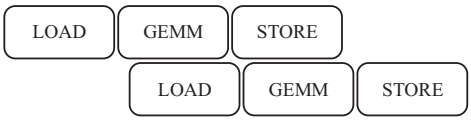
The RoCC accelerator uses normal RISC-V and custom RoCC instructions. The RISC-V handles the data layout transformation and float-point calculation, while the RoCC accelerator performs DMA operations and integer calculation. The accelerator employs a decoupled-load-access memory access pattern, provides mainly data moving instructions and computing instructions.

B. Ping-pong Buffering Strategy and Memory Model of Compiler

Convolution occupies a large proportion of calculations in convolutional neural networks. The acceleration of convolution calculation is the main point of neural network acceleration. The DNN accelerator usually uses the GEMM unit to accelerate the convolution. The feature map and weight kernel need to be transformed into matrices through the Im2Col [17] algorithm.

The computing unit of the accelerator has to be reused for big workloads like neural networks. General computation flow

(a) The serial execution flow of computation workload.



(b) The pipeline parallelism with pingpong buffer

Fig. 3: The comparison of the execution flow between the ping-pong buffering strategy and the baseline. The program execution of the ping-pong buffering strategy looks like a pipeline.

is the repetition of "data loading-calculating-result storing", illustrated in the Fig. 3a, which is serial and low efficient. However, the ping-pong buffer can offer a parallel execution for this round of computation and the previous one. The pipeline parallelism of the ping-pong buffer is illustrated in Fig. 3b, which can hide the computation and storage period of the last round. This figure shows that the ping-pong buffering strategy improves the parallelism of the program.

Many DNN accelerators are using ping-pong buffers for the GEMM unit. The programmer needs to partition the input matrices, and send the blocks to the ping buffer and pong buffer alternately. In the compiler, we designed a model of the presented RoCC accelerator, describing the name, size, and data type of the buffers. The model will guide the programmer to map the DNN on the RoCC accelerator. The programmers can use TVM primitives to bind the values in the program to the RoCC buffers of the accelerator model. Also, the accelerator model will constrain the design of handcraft kernels for the RoCC accelerator. Violating the constraints will result in not getting the right generated code. This hardware model will help programmers make use of the ping-pong buffers to improve the efficiency of the workload on the RoCC accelerator.

C. Code Generation for Accelerator

The proposed automated compiler is illustrated in Fig. 4. Firstly, the programmer of TVM needs to add tags to the program they want to deploy on the RoCC accelerator. Tags can be added by tensor intrinsic or pragma. These tags are used to tell the code generator where to insert the RoCC instructions or replace the loop nests with handcrafted kernels. Then TVM will lower the program to Tensor IR (TIR) with tags. TIR is a program composed of nested loops. Third, the optimization passes of the automated compiler transform the nested loops into RoCC instructions according to the tags mentioned above. Finally, based on the TVM C backend, the C program is generated. At last, the compiler creates the RoCC-enabled target binary with RISC-V GNU and the instruction head file.

D. Tensor Intrinsic

The proposed automated compiler makes use of the ability to call extern functions of TVM, which makes the compiler can replace a unit of computation with the corresponding intrinsics. The programmers can leverage handcrafted micro-kernels with tensor intrinsic and embed RoCC instructions in their program. Because the instruction set of diverse accelerators differs from

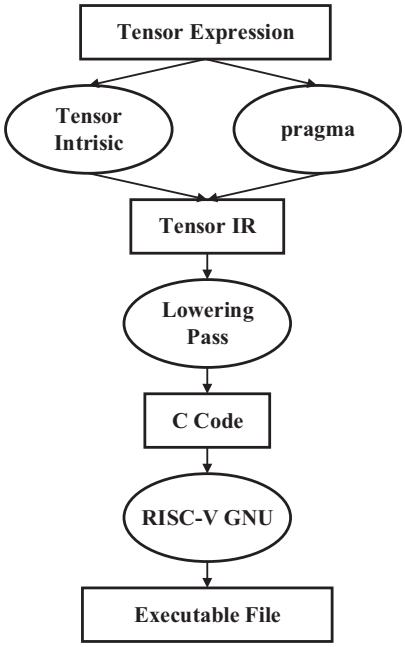


Fig. 4: The code generation flow of the compiler. The boxes represent the IR format of deep learning models while the ellipses represent the compilation stages.

design to design, it is significant to design the intrinsic for high-performance code.

In our RoCC-based accelerators, the ISA exposes the data movement instructions and computation unit instructions. The work in [10] has a similar ISA and its instruction pattern suggests that its intrinsics has three patterns, which is reset-update-finalize. They designed data moving out as an important part of intrinsic. However, our compiler decoupled the computation and data movement, giving more freedom to programmers designing handcrafted kernels. The natural structure of intrinsic only focus on computation and have two phases of computational unit operation, which is "reset-update":

1) The reset phase is designed to prevent the data remaining in the output buffer interfere with the accumulation of the partial results.
2) In the update phase, the input data in this round are calculated and combined with the partial results stored in the output data.

Most accelerators with a hierarchical memory structure need to tile the input data. RoCC accelerators usually rely on the Rocket core to perform the data layout transformation. The intrinsic defined in our compiler constrains the memory scope and computation. Violating the constraints will result in not getting the right generated code.

E. Flexible Loop Nest Substitution by RoCC Instructions

Due to the limited sources of the hardware, the DNN accelerators usually need to tile the input data flexibly according to the size of the computational units and its hierarchical memory structure. The tensor intrinsic is designed to support the complex computation workload such as reduction and sacrifices a little bit of flexibility. The programmer needs to follow its constraints after defining the intrinsic function.

The proposed compiler provides another way to replace the TVM program with handcrafted kernels which is more

flexible. This approach is made for some simpler computations such as elementary computation and data movement. It is different from tensor intrinsic which needs to match the computation patterns. It is based on pragma, which is one of the TVM primitives. We customize the pragma to insert a tag in TIR. The substitution based on pragma takes two steps to replace the loop nests into handcrafted kernels or RoCC instructions:

1) The programmer needs to embed pragma in the schedule structure of the TVM program. When the TVM program is lower to TIR, the pragma tags will show in the place where the programmer wants to replace the loop nests.
2) The compiler lowers the TIR and replaces the loop nest with the handcrafted kernels or instructions.

When writing a program for the accelerator, due to the constrained handcraft kernel, the intrinsic-based approach can only replace a fixed size of the loop nests with the kernel. In the way of substitution based on pragma, the proposed approach can replace any size of the loop nest of element computation. Also, because the outer part of the loop nests remains, the program has to repeat the kernel but with different input data blocks every round. Under this circumstance, the proposed approach can add data offset for the repeated kernel automatically to make sure the computation kernel can get the right data blocks each time. Besides, the proposed approach can calculate the sizes and shapes of the input tensors of the kernels or instructions automatically, which can offer the necessary operand for the called instructions. At last, this approach can classify the computation of the loop nest and generate the correct operation code for the VE instructions.

IV. Experimental Results

In this section, we will show how the proposed compiler generates efficient low-level code to make use of the ping-pong buffers on the accelerator and the performance improvement.

We designed a high-performance GEMM operator and a convolution operator for the RoCC accelerator mentioned above using the ping-pong strategy on the proposed compiler. GEMM operator does matrix block multiplication and the convolution is implemented by Im2Col [17] transformation and GEMM. The accelerator is implemented on a Xilinx VC707 platform.

The low-level code is generated to verify the functionality and performance. The GEMM unit is calculating A*B. With the constraint of the RoCC buffer sizes, the maximum input dimensions the GEMM unit can accept in one turn A is 8*1024 and 1024*8 for B. Here we evaluate the performance of matrix multiplication of 16*k and k*16, while k is the workload size. The baseline is the same GEMM computation but without the ping-pong strategy. We present the performance comparison results in Fig. 5. The low-level code with ping-pong strategy reduces the computing period at most by 25%. Also, the speed-up rate increases with the increase of the workload size.

Meantime, we evaluate the running time of matrix block multiplication of a different number of blocks. We present the performance comparison results in Fig. 6. With a workload size of 256, the low-level program generated by the proposed compiler can achieve speed up to 20% at most. Also, the speed-up rate increases with the growth of the matrix blocks.

As for the convolution operator, it can achieve speed up with at most 26% at the example convolution. We verified several examples of convolution layer in Resnet in Table I. All of the test examples achieve an acceleration rate higher than 20%. The results tell that our compiler can use the ping-pong strategy to implement large-scale programs more efficiently.

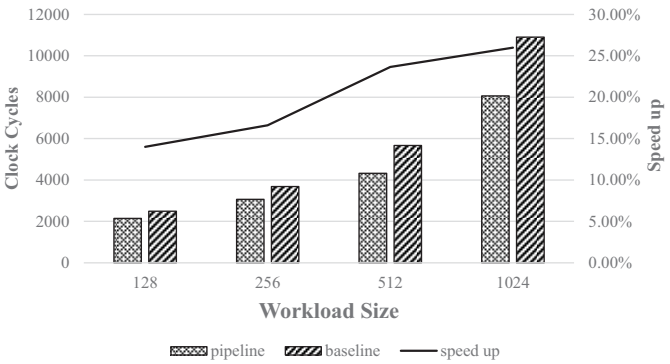


Fig. 5: The comparison of the program execution time between the ping-pong buffering strategy and the baseline with different workload sizes.

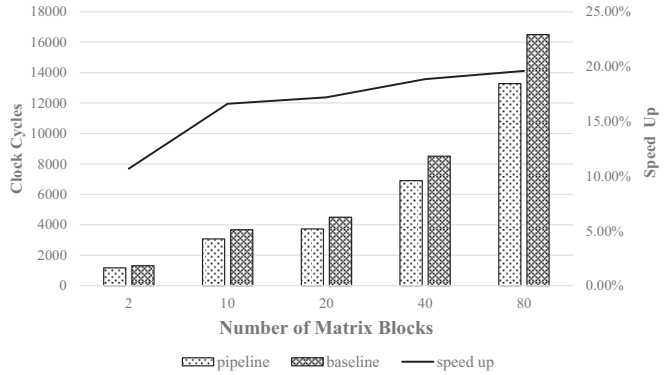


Fig. 6: The comparison of the program execution time between the ping-pong buffering strategy and the baseline with different matrix blocks.

TABLE I: The speed comparison between the ping-pong buffering strategy and the baseline on some example convolutions. The designed convolution operator can achieve 26% speed up at most.

Feature Map Shape	Weight Shape	Pipeline	Baseline	Speed Up
(1, 16, 14, 14)	(16, 16, 3, 3)	8942	10812	20.91%
(1, 16, 28, 28)	(16, 16, 3, 3)	33633	42075	25.10%
(1, 16, 56, 56)	(16, 16, 3, 3)	133308	167992	26.02%
(1, 128, 28, 28)	(128, 128, 1, 1)	252993	314297	24.23%

V. Conclusion

In this paper, we introduced an automated compiler for the RoCC accelerator, which can lower the TVM program to the executable file for the RoCC accelerator flexibly and high efficiently. Also, we made use of the ping-pong strategy on the accelerator with the proposed compiler, which accelerates the convolution up to 26%. We hope our work will provide an efficient approach for code generation on the RISC-V based accelerator platform.

Acknowledgement

This research is supported partly by National Key R&D Program of China 2019YFB2205002, partly by National Natural Science Foundation of China (NSFC) 61822402, 62090025, 62011530132 and 61929102.

REFERENCES

[1] Thierry Moreau Tianqi Chen and Eddie Yan. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems*, pages 579–594. IEEE, 2018.

[2] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Palmer Dabbelt, John R. Hauser, Adam M. Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, Jack Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moretó, Albert J. Ou, David A. Patterson, Brian C. Richards, Colin Schmidt, Stephen Twigg, Huy D. Vo, and Andrew Waterman. The rocket chip generator. 2016.

[3] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C Ling, and Gordon R Chiu. An opencl™ deep learning accelerator on arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 55–64, 2017.

[4] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. Dlau: A scalable deep learning accelerator unit on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):513–517, 2016.

[5] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 161–170, 2015.

[6] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert Ou, Max Banister, et al. Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. *arXiv preprint arXiv:1911.09925*, 3, 2019.

[7] Yunsup Lee, Andrew Waterman, Rimas Avizienis, Henry Cook, Chen Sun, Vladimir Stojanović, and Krste Asanović. A 45nm 1.3 ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators. In *ESSCIRC 2014-40th European Solid State Circuits Conference (ESSCIRC)*, pages 199–202. IEEE, 2014.

[8] Spike. https://github.com/riscv-software-src/riscv-isa-sim.

[9] Riscv-dlr. https://github.com/nthu-pllab/RISCV-DLR.

[10] Pengcheng Xu and Yun Liang. Automatic code generation for rocket chip rocc accelerators.

[11] Zhiqiang Liu, Paul Chow, Jinwei Xu, Jingfei Jiang, Yong Dou, and Jie Zhou. A uniform architecture design for accelerating 2d and 3d cnns on fpgas. *Electronics*, 8(1):65, 2019.

[12] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[13] SiFive. Sifive tilelink specification. 2017.

[14] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, page 58–68, 2018.

[15] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[16] Apache. Apache tvm documentation.

[17] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Tenth international workshop on frontiers in handwriting recognition*. Suvisoft, 2006.