Automatic Code Generation for Rocket Chip RoCC Accelerators

Pengcheng Xu
jsteward@pku.edu.cn
Peking University

Yun Liang
ericlyun@pku.edu.cn
Peking University

ABSTRACT

Heterogeneous SoCs that are coupled with accelerators are becoming prevalent for various deep learning applications thanks to their outstanding flexibility and performance. However, programming for these platforms remains hard due to their low-level programming interface and complex memory systems. Meanwhile, automatic code generation for tensor programs provides reasonable performance with great accessibility and flexibility. In this work, we bring these two topics together by proposing a flow of automatic code generation for heterogeneous SoCs. We present how to implement the proposed flow using TVM for RoCC. We also develop a performance evaluation platform to enable practical automatic code generation on embedded devices. Experiments using TVM for the Gemmini GEMM accelerator demonstrate that the generated code achieves a peak of 25.24 GIOPS under 100 MHz clock and a best-case 3.6x speedup compared to the hand-tuned kernels from Gemmini developers.

CCS CONCEPTS

• **Software and its engineering** → *Domain specific languages*; • **Computer systems organization** → **Reduced instruction set computing**; • **Hardware** → **Hardware accelerators**.

KEYWORDS

RISC-V, Accelerators, Automatic Code Generation, TVM

1 INTRODUCTION

FPGA-based accelerators have become prevalent for various deep learning training and inference workloads [5] [31] [35]. Heterogeneous System-on-Chip (SoC) designs that target deep learning applications, such as image classification, object detection, tracking, and voice recognition, tend to use accelerators coupled with a CPU to minimize the latency and maximize the throughput for these tasks [13] [33] [14]. Such a design paradigm excels in flexibility and can offer decent performance for various emerging applications. The Rocket Chip [3] project is an extensible SoC generator framework that allows users to easily build such a system with a RISC-V CPU and a custom accelerator through the Rocket Custom Coprocessor interface, also known as RoCC. Multiple novel designs of accelerators with the RoCC interface for common workloads such as matrix multiplication [15], vector extensions [17], and cryptography [28], are available through the Chipyard project [24]. It is also possible to design custom accelerators for specific workloads using Chisel infrastructure [2] [6].

The hardware accelerator requires high-performance, low-level software to release its computing power to the fullest. For example, the NVDLA [37] exposes a complicated register interface [12] for precise control of its hardware functional units. However, it is challenging to develop efficient software for complex heterogeneous systems. The software needs to map the computation into low-level interfaces of the accelerator. It is apparent that the mapping process directly affects the overall throughput and latency of the system. However, most of the current designs only provide a low-level programming interface. This requires application designers to manually write programs using low-level programming languages such as C, which poses a significant burden on the application developers. It also severely limits the capability to explore the schedule space thoroughly.

Meanwhile, automatic code generation is gaining increasing attention recently [18] [26] [30]. Recent works such as Halide [22] and TVM [10] introduces the paradigm of separating computation definition and optimization schedules to simplify the programming and optimization. With a common Intermediate Representation (IR), these frameworks enable efficient computation kernel generation for diverse hardware platforms such as CPUs, DSP, GPUs, and VTAs [19]. The decoupling of schedules and computation also makes automatic exploration of the schedule space for an optimal configuration possible. AutoTVM [11], as part of TVM, generates optimal kernels by iteratively exploring the parameters and tuning the parameters in a closed feedback loop. In each tuning round, a kernel is generated and tested on the target device for quality. The framework then uses learning algorithms [8] to digest the measurements and use them to instruct further tuning. Recently, FlexTensor also allows automatic search of schedule spaces in addition to parameters, which further enlarges the design space and improves the overall performance [36].

On the other hand, heterogeneous SoCs with dedicated accelerators are widely used for edge computing and Internet of Things (IoT) domains. In general, these bare-metal devices often do not have sophisticated operating systems to abstract and manage the peripherals *bare-metal.* This poses great challenges for automatic code generation because such a system does not provide the high-level abstractions for the hardware; thus, the programmers have to handle them manually. Besides, due to limited I/O available on these platforms, it is hard to interface with the device to perform automatic code generation, profiling, or measurement.

In this work, we propose an automatic code generation framework for the family of RoCC accelerators. We specialize the process of offloading computation to external procedures, also known as *tensorization*, to take the characteristics of RoCC-based accelerators into consideration. We structure the computation performed on the accelerator as *tensor intrinsics* to express the data flow pattern of RoCC accelerators. To realize practical auto-tuning required for automatic code generation on these RoCC platforms, we exploit the shared memory capability found on popular FPGA platforms to implement efficient code evaluation. Such FPGA platforms provide high-speed access to the memory of the programmable logic via a host processor. We make use of this capability to work around

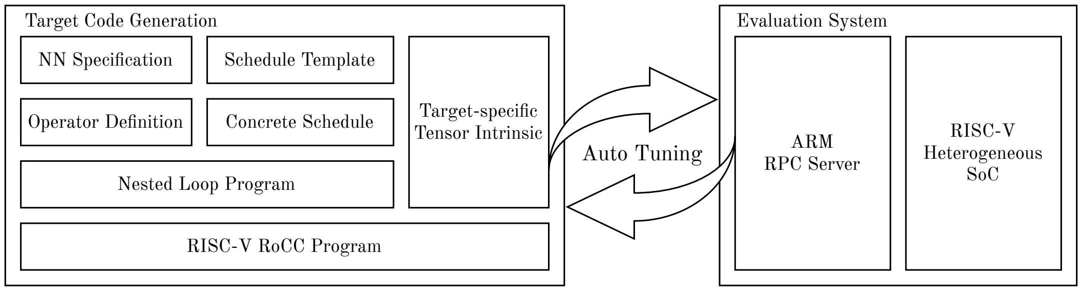

Figure 1: Automatic code generation framework for RoCC accelerators.

the communication bandwidth bottleneck between the code generation server and evaluation systems. Contributions of this work are:

- We propose a flow of automatic code generation with tensorization for the family of RISC-V RoCC-based accelerators.
- We formulate the design of tensor intrinsics for such accelerators.
- We present an efficient evaluation system for auto-tuning during code generation for such accelerators.

Experiments using TVM to generate code for the Gemmini GEMM accelerator shows that the generated code has consistent and comparable performance to the hand-tuned kernels provided by Gemmini developers, with a peak performance of 25.14 GIOPS under 100 MHz clock and best-case speedup of 3.6x.

2 BACKGROUND AND MOTIVATION

2.1 RoCC Overview

The Rocket Chip Coprocessor (RoCC) is a special feature of the Rocket Chip SoC generator to integrate custom coprocessors into the SoC. RoCC specifies an interface between the CPU core and the coprocessor. The CPU controls the accelerator via a custom instruction opcode reserved for RoCC, which is sent to the accelerator. The accelerator accesses system memory coherently, either via the TileLink [29] on-chip bus or over a simple interface to the CPU L1 data cache. The RoCC interface is ideal for accelerators that couple closely to the CPU in heterogeneous SoC designs compared to traditional Memory-mapped I/O (MMIO) together with Direct Memory Access (DMA) engines due to the elimination of typical peripheral bus protocol overheads.

2.2 Automatic Code Generation

Traditional flows of writing software for hardware accelerators require the programmers to manually transform the code, which is tedious and error-prone. Halide [22] allows users to specify the computation definition and schedule definition separately to solve this problem. The computation rules, expressed as lambda expressions, define the basic structure of the resulting nested loop program. The optimizations are structured as *schedule* that perform loop transformations. This paradigm dramatically eases the development of computation kernels, lifting the repetitive task off the developers' shoulders. TVM [10] takes the idea of Halide a level further with general support for specialized hardware accelerators. TVM supports expressing computation opaque to the TVM Domain Specific Language (DSL) in terms of *tensor intrinsics* that denote special hardware operations. The tensor intrinsics are implemented in C to perform the actual computation. The user uses the `tensorize` schedule to mark a part of a computation to be performed by the tensor intrinsic. Calls to the intrinsic functions for accelerator calls are then emitted during code generation.

All possible schedules of a kernel form the schedule space. One way to find the optimal schedule introduced by the code generation framework is to search the schedule space. AutoTVM, part of TVM, does so iteratively for an efficient schedule. The AutoTVM framework consists of a *compile server* that generates the code for the target, *runners* that evaluates the generated code, and a set of *tuning algorithms* that accepts the performance readings and instructs further code generation. Automatic code generation frameworks enable quick interface with popular deep learning frameworks. TVM, for example, allows the user to directly import models from deep learning frameworks that support the Relay IR [25] format, such as PyTorch [21], TensorFlow [1], or MXNet [9]. The user can either choose existing or write custom models in the deep learning framework. The framework then generates efficient code by auto-tuning the applications for the target hardware. Compared to the manual design flow, this greatly improves productivity.

2.3 Performance Evaluation for SoCs

Performance evaluation on SoC platforms is not trivial. It is not possible to use modern operating systems on these platforms due to resource limits. Thus, a bare-metal runtime that handles communication and provides basic facilities is needed. μTVM [4], a recent addition to the TVM framework, provides a minimal runtime for executing TVM modules on the target devices. The project currently uses OpenOCD, the prevalent debug translator for embedded devices, to bridge TVM with the evaluation devices. The host and device talk over the Joint Test Action Group (JTAG) debug protocol, which is widely used across device manufacturers. SoCs often use hierarchical memory to improve performance. In the meantime, most deep learning accelerators tend to coalesce memory transactions to improve performance, making it difficult to accurately model cache behavior and overall performance for the resulting system. Automatic code generation circumvents this situation by measuring the end-to-end execution time of the program, which implicitly takes cache operation into consideration.

3 AUTOMATIC CODE GENERATION FOR ROCC ACCELERATOR

3.1 Overview

The proposed automatic code generation framework for RoCC accelerators is shown in Figure 1. To generate code of a DNN for the accelerator, the user needs to provide the *NN specification* with one of the supported deep learning frameworks such as TensorFlow or MXNet. They also need to provide the *schedule template* for possible optimizations. The code generation framework creates *operator definitions* for the computation kernels and picks a *concrete schedule* from the schedule space to generate the *nested loop program* IR. Then, the framework offloads computation to the accelerator per the user specified with the *target-specific tensor intrinsic* implementation. Specifically, the framework transforms the IR, replacing loop levels marked by the schedule with calls to the intrinsics to perform accelerator operations. The framework then lowers the IR into C program and creates *RoCC-enabled target binary* of the C program with RISC-V GCC. Finally, the auto-tuning system sends the binary to the *evaluation system* and uses the performance readings to pick the schedule for the next tuning round.

3.2 Intrinsic Design

With the ability to call external C functions in code generation frameworks, it is straightforward to embed accelerator calls in the generated code as intrinsics. However, due to accelerators in heterogeneous SoCs exposing their ISA with granularity that differs significantly from design to design, it is crucial to design a intrinsic properly for high-performance code. In RoCC-based accelerator designs, the ISA usually exposes explicit data movement instructions into and out of the accelerator memory. Such a pattern suggests that the natural structure of the intrinsics should be in the form of three procedures, "reset-update-finalize", to represent the three phases of accelerator operation:

(1) For an output region, the *reset* function is called first to initialize the output region in SoC memory.
(2) The *update* function then combines the partial results of previous computation calls for the current output region with this round's input data to produce the output.
(3) Finally, at the end of the intrinsic, the *finalize* function is called to move output from the accelerator back to the main memory.

For the computation of a specific region, the partial results do not need to be transferred back to SoC memory in each round but can stay in the accelerator memory. As memory movements are taking time on par with computation in accelerators, this is crucial for generating efficient code. Most DNN kernels have their input tiled in a system with hierarchical memory [20]. RoCC accelerators usually rely on the RISC-V CPU to perform tiling in order to fit data inside accelerator memory. This results in hard limits for the dimensions of data the intrinsic can handle; violating these constraints would result in invalid generated code. We enforce the constraints posed by accelerator memory size limits during code generation. The user shall specify the constraints the device requires in the intrinsic declaration. The code generation framework checks that the input and output shapes of the intrinsic conform to the constraints before proceeding to actual code generation.

3.3 Barrier Instruction

RoCC-based accelerator designs require explicit synchronization to guarantee data transaction completion. This results in an asynchronous memory access pattern: for the CPU to access memory just written back from the accelerator, the software needs to insert barrier instructions to force the memory operations to become visible to the CPU. The code generation process needs to take this into consideration.

Two possible approaches exist to emit the required barrier instruction. One is to fuse the barriers into the *finalize* stage of the intrinsic, resulting in an implicit barrier for each output region. The other approach is to provide separate control over when to emit the barrier instruction by using the code block annotation feature of the code generation framework. The user annotates that a barrier instruction should be inserted at the end of the DNN kernel. These two approaches also result in different performance. The first approach would result in unnecessary barrier instructions in the generated code, which will impact performance due to the introduced excessive waiting and unneeded CPU stalls. This effect may get worse if the RISC-V CPU has more than one core due to the memory ordering requirement the barrier instruction poses on them. The latter approach circumvents this by providing greater flexibility and better control over barrier insertion. This leads to better performance in generated code for allowing parallel functioning of the memory access units and the computation logic. The user may even choose to omit the barrier to seek for cross-kernel asynchronous memory operations as an optimization.

3.4 An Example of Generated Kernel

Listing 1 gives the pseudo-code for a generated GEMM kernel with input sizes of $512\times512\times512$ using our flow. The core computation logic in GEMM, the multiply-add operation, is delegated to the intrinsic stage `matmul_kernel.matmul_reset` and `matmul_finalize` operates on the output region for data movement. The matrix axes are split into 8×64, with the axis of extent 64 bound to the intrinsic. We pass 512 to the intrinsic as matrix access stride. We follow the explicit barrier approach described in Section 3.3. The `attr pragma_epilogue` attribute line marks that one memory fence instruction is required after the `i.o` loop. If we choose the implicit approach and fuse the `fence` instruction in `matmul_finalize`, a total of 64 barrier instructions will be issued during kernel execution (two levels of loop `i.o`, `j.o` of extent 8). This means more synchronization between memory and the accelerator and worse performance.

4 PERFORMANCE EVALUATION ON ACCELERATORS

Automatic code generation requires evaluation of the generated code to acquire feedback on its quality. Such feedback is not only crucial for choosing the best configuration but affects the search process as well. In this section, we propose an efficient communication scheme between the code generation host and target device

```
produce C {
  // attr pragma_epilogue = "do_fence"
  for (i.o, 0, 8) {
    for (j.o, 0, 8) {
      matmul_reset(access_ptr(C), 512)
      for (k.o, 0, 8) {
        matmul_kernel(access_ptr(A), access_ptr(B),
        ↪  access_ptr(C), 512, 512, 512)}
      matmul_finalize(access_ptr(C), 512)}}}
```

Listing 1: Simplified version of an example GEMM kernel. The target is expected to provide matmul_reset, matmul_kernel, matmul_finalize, **and** do_fence **as C functions.**

for efficient evaluation during code generation. We present the design of the Rocket Chip-based evaluation platform as an implementation of the proposed communication scheme.

4.1 Communication Scheme

Code generation frameworks feature device-host interfaces to perform arbitrary code evaluation on bare-metal devices shown in Listing 2. Previous implementations of the interface have a low bandwidth between the host and device can quickly become the bottleneck of code generation.

```
/* Read num_bytes from addr into buffer */
void Read(Ptr addr, void *buffer, size_t num_bytes);
/* Write num_bytes to addr from buffer */
void Write(Ptr addr, const void *buffer, size_t
↪  num_bytes);
/* Start execution at func_addr and return at
↪  stop_addr */
void Execute(Ptr func_addr, Ptr stop_addr);
```

Listing 2: Functions required for the μTVM device interface. Ptr **denotes a pointer in the target device's memory.** void * **denotes a pointer on the compile server.**

We propose an implementation of the host-device interface for accelerators hosted on heterogeneous FPGA platforms. An example of such a platform would be Xilinx Zynq, which has fabricated ARM CPU cores built into the same die as the FPGA. Such platforms enable mutual high-speed access to the system memory from both the FPGA and the host ARM processor [27] [23]. The fast interface improves code evaluation throughput and shortens the time for creating an optimal DNN kernel with AutoTVM. With shared memory, the Read and Write functions of the interface are naturally expressed as coherent reads and writes to the host processor's memory. The Execute function is implemented with the help of the monitor software that runs on the target CPU.

4.2 Hardware and Software System

The SoC platform for code evaluation is based on the Rocket Chip SoC generator, featuring a single 64-bit Rocket Chip CPU core with

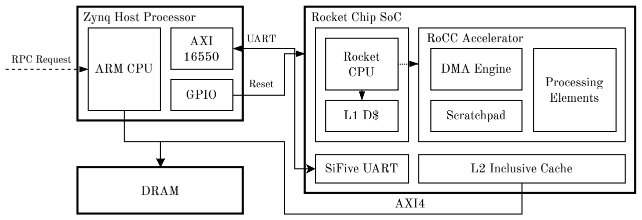

Figure 2: Block diagram for the evaluation platform in FPGA.

virtual memory support. An inclusive last-level cache (LLC) implementation from SiFive is included to explore the cached memory access behaviors discussed in Section 2.3. A UART controller provides a lightweight communication channel to implement the host-device interface, as described in Section 4.1. The block diagram for the evaluation platform is shown in Figure 2.

A *monitor* software is needed on the Rocket Chip SoC to implement the host-device interface. The monitor software is based on the OpenSBI firmware. On Execute call from the code generation host, the host ARM processor sends the desired func_addr and stop_addr to the monitor over the UART link. The monitor software replaces the instruction at the stop address with an invalid opcode, unimp, and jumps to the function address. Once the code being evaluated runs to stop_addr, the CPU will jump to the invalid opcode trap handler, which is provided by the monitor software. The trap handler then reports the execution time to the host ARM processor. The run time of the code being evaluated is measured via the cycle Control and Status Register (CSR) to provide accurate timing. We run the generated code in S-mode and protect the monitor code and data with RISC-V Physical Memory Protection (PMP) [32]. The host ARM processor handles TVM RPC requests from the compiler server over Ethernet and forwards them to the evaluation platform in FPGA. The host processor software enforces the control flow of the RISC-V processor via the Rocket Chip reset signal connected over GPIO. The execution flow of the host and monitor software is shown in Figure 3.

5 CASE STUDY OF GEMMINI

To evaluate the proposed methodology for RoCC accelerators, we present the flow of generating code for the Gemmini [15] accelerator as a case study to show the quality of the code generated as well as the efficiency of the flow. We use TVM [10] as the code generation framework, with some modifications to implement the flow as described in Section 3. We test the performance of the code generated for Gemmini on a Rocket Chip SoC mapped to FPGA, running at 100 MHz. We implement the test platform described in Section 4 on a Xilinx Zynq UltraScale+ ZCU102 Evaluation Board [34].

5.1 Gemmini Overview

We first briefly describe the data flow design and programming model of the Gemmini GEMM accelerator. The Gemmini RoCC accelerator implements GEMM with a systolic array structure. The

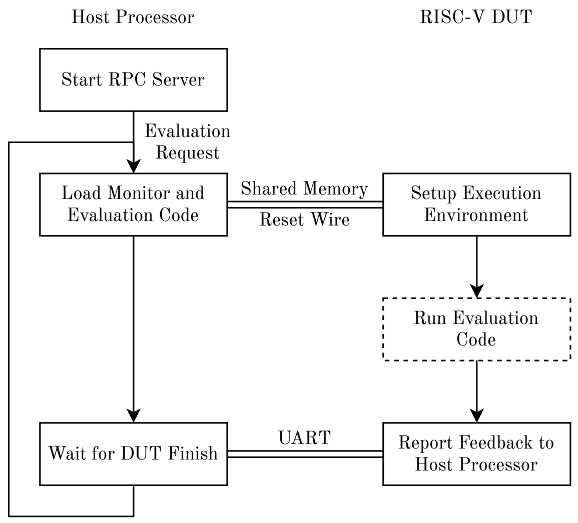

Figure 3: Execution flow of the host and monitor software. The box in dotted lines represents generated code from the compile server. The double lines between the RISC-V Design Under Test (DUT) and the host processor represents hardware links.

core logic of computation consists of an array of Processing Elements (PEs). The bias matrix is preloaded into the internal accumulators of the PEs, and the matrices are pushed through the systolic array, accumulating $A \times B$ on top of D. Gemmini includes two memory regions inside the accelerator, the *scratchpad* and the *accumulator*, for holding input and output data from the PE array. In the default Gemmini configuration, the scratchpad and accumulator are 256KiB and 64 KiB. Gemmini employs a decoupled-load-access memory access pattern, providing separate instructions for computation and data movement. Three main instructions are provided to perform data movement in two directions and to perform computation.

Gemmini uses mixed normal RISC-V and custom RoCC instructions. The RISC-V CPU code handles data partitioning while the accelerator performs DMA operations and calculation. The DMA engine operations' completion is asynchronous to the retire of the memory access instructions. This memory architecture requires explicit `fence` instructions for data consistency. Gemmini provides a hand-tuned GEMM kernel for applications. Two-levels of input tiling are performed to saturate accelerator memory with the input and output matrices for the optimal data reuse in accelerator scratchpad. The kernel is then used to implement a convolution kernel via the Im2Col [7] transformation. The kernels are used to implement deep learning applications for the accelerator.

5.2 Implementing the Code Generation Flow

Two sets of constraints apply to the maximum input sizes that can be handled in one turn for Gemmini. The limitations are due to two limiting factors, the scratchpad capacity, and the accumulator capacity. Given the set of hardware customization parameters, we derive the constraints for input matrix sizes. Assuming that the

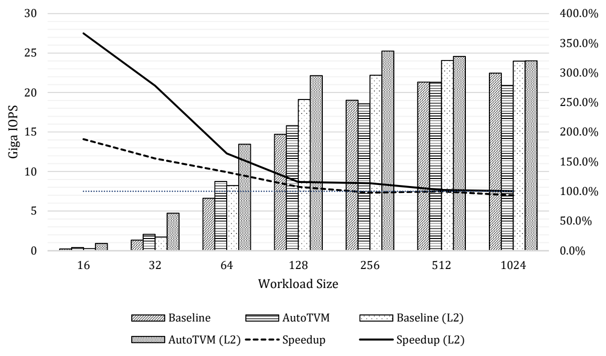

Figure 4: Performance comparison between baseline (hand-tuned kernel) and AutoTVM-generated results for different input matrix sizes in terms of Integer Operations Per Second (IOPS). Data with label marked "L2" indicates that the L2 cache in Rocket Chip is enabled. The dotted horizontal line marks 100% speedup.

input words are w_i bits each, the accumulator words are w_a bits each, the scratchpad memory is b_s bytes, the accumulator memory is b_a bytes, and the input matrices are of $i \times j$ and $j \times k$, the size constraints for the tensor intrinsic are:

$$i \cdot k \cdot w_a / 8 \leq b_a \tag{1}$$

$$(i \cdot j + j \cdot k) \cdot w_i / 8 \leq b_s \tag{2}$$

With default Gemmini parameters of 8-bit input words, 32-bit accumulator words, 256 KiB scratchpad, and 64 KiB accumulator, the maximum input dimensions the accelerator can accept in one turn A is 128×1024 and 1024×128 for B. The minimum block size of the systolic array accepts is 16. We implement the tensor intrinsic for a GEMM kernel in C and inline assembly. Besides input with sizes that match the accelerator memory, inputs that are smaller than the maximum allowed size are also accepted to explore the trade-off between accelerator data reuse and cache performance. The DMA engine of Gemmini supports partial loads that does not fill a single line of the systolic array. In case the input size is not divisible by the systolic array dimensions, the generated code issues such partial loads and stores to the DMA engine to provide padding in the accelerator memory correctly.

5.3 Generated Code Quality

We generate code targeting Gemmini with AutoTVM to verify functionality and performance. We define a simple schedule space that performs loop axis split on the three axes in GEMM. The hand-tuned kernel applies the same tiling and reordering optimization, but with predetermined factors to saturate accelerator memory. We present the performance comparison results in Figure 4. The baseline performance is from Gemmini's provided hand-tuned kernel that saturates the accelerator memory. The AutoTVM kernel is selected from up to 100 random trials from the schedule space. We can see that the L2 cache does provide a consistent performance boost of over 10% across all input sizes.

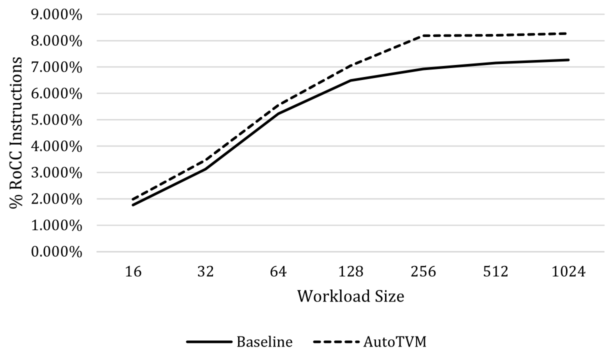

Figure 5: The percent of RoCC instructions in all instructions executed for different input sizes. The ratio between RoCC and normal instructions approach a fixed value as input size increases.

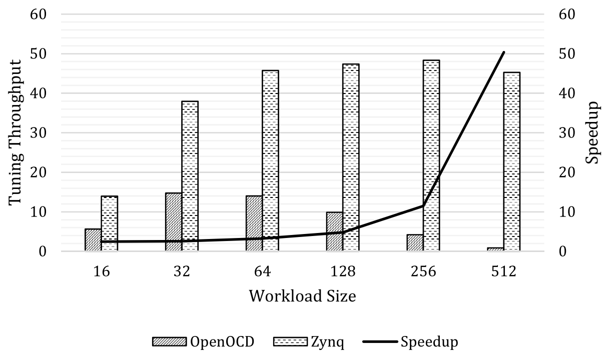

Figure 6: Tuning performance comparison between the proposed Zynq platform and the OpenOCD JTAG platform. The tuning throughput is shown in trials per minute.

We recognize that the AutoTVM-generated kernels show performance on par with the hand-tuned kernel for all input sizes. An interesting observation is that the generated kernel performs considerably better on smaller inputs. We anticipate that this is due to the hand-tuned kernel having more overhead in non-computation parts than the automatically generated version, as the code generation framework tends to perform inlining and optimizations more radically. Figure 5 shows the proportion of RoCC instructions issued to all instructions issued for the generated and hand-tuned code for different input sizes. We can see that as the input sizes increase the proportion approaches a fixed value. This renders the overhead less significant in time consumption as input size increases, explaining the similar performance for both kernels on larger inputs.

5.4 Auto-tuning Throughput

We share the early results of the performance of the proposed evaluation platform. Figure 6 shows the tuning throughput in terms of numbers of evaluations per minute for a GEMM kernel on CPU for different input matrix sizes compared to the original OpenOCD JTAG implementation. The proposed platform shows significantly better tuning efficiency compared to the baseline design.

We believe that the improvement is due to the improvement of efficiency of the communication scheme. The amount of data to be transferred in each trial is proportional to the square of input size. A back-of-the-envelope calculation shows that the amount of data needed for one turn of tuning quickly approaches several megabytes for larger inputs. This matches what we can see in Figure 6: as input size increases, the tuning throughput of the OpenOCD implementation quickly collapses due to bandwidth bound. In the meantime, the Zynq platform shows stable tuning throughput regardless of workload size due to the surplus bandwidth from the shared memory system. The proposed evaluation platform design greatly improves the efficiency of code evaluation, especially for larger inputs, in auto-tuning on heterogeneous SoC platforms.

6 RELATED WORK

The introduction of closely-coupled accelerators with RoCC greatly extends the frontier of hardware-accelerated processing by allowing easy addition to the processor ISA. Besides Gemmini discussed in this work, Hwacha [17] introduces a new vector architecture for elegant, performant, and energy-efficient vector processing on modern data-parallel architectures. The SHA3 RoCC accelerator [28] enables high-performance secure hash calculation on embedded platforms. SMURF [6] brings hardware-accelerated Variable Precision (VP) Floating Point (FP) for high-performance computing servers as an alternative to VP FP software routines.

Besides the pattern in TVM of the user providing operator definition and schedule instructions, automatic code generation can have different approaches. Glow [26] lowers the traditional neural network dataflow graph into a two-phase strongly-typed intermediate representation, allowing the optimizer to perform domain-specific optimizations. The MLIR [16] project defines a common intermediate representation (IR) that unifies the infrastructure required to execute high-performance machine learning models in TensorFlow [1] and similar ML frameworks. μTVM [4] first introduced and implemented the host-device interface described in Section 4.1. In μTVM's implementation, the host uses OpenOCD with JTAG or other debug protocols to implement the interface functions. TVM also offers an RPC interface to communicate with more powerful targets such as Android devices to perform code evaluation.

7 CONCLUSION

In this work, we introduced the paradigm of automatic code generation for RoCC accelerators. We validated the feasibility of the proposed flow via a case study of code generation for the Gemmini accelerator with TVM. We also presented an efficient evaluation platform design to make auto-tuning for heterogeneous platforms realistic for real-world applications. We hope that this will open the possibilities of automatic code generation for accelerators in the RISC-V ecosystem.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Tutu Ajayi, Khalid Al-Hawaj, Aporva Amarnath, Steve Dai, Scott Davidson, Paul Gao, Gai Liu, Anuj Rao, Austin Rovinski, Ningxiao Sun, Christopher Torng, Luis Vega, Bandhav Veluri, Shaolin Xie, Chun Zhao, Ritchie Zhao, Christopher Batten, Ronald G. Dreslinski, Rajesh K. Gupta, Michael Bedford Taylor, and Zhiru Zhang. 2017. Experiences Using the RISC-V Ecosystem to Design an Accelerator-Centric SoC in TSMC 16 nm. *First Workshop on Computer Architecture Research with RISC-V (CARRV 2017)* (2017).

[3] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator.* Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[4] The TVM authors. 2019. [RFC][MicroTVM] Bringing TVM to Bare-Metal Device. https://github.com/apache/incubator-tvm/issues/2563

[5] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An OpenCLTM Deep Learning Accelerator on Arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. Association for Computing Machinery, New York, NY, USA, 55–64. https://doi.org/10.1145/3020078.3021738

[6] Andrea Bocco, Yves Durand, and Florent De Dinechin. 2019. SMURF: Scalar Multiple-Precision Unum Risc-V Floating-Point Accelerator for Scientific Computing. In *Proceedings of the Conference for Next Generation Arithmetic 2019 (CoNGA'19)*. Association for Computing Machinery, New York, NY, USA, Article Article 1, 8 pages. https://doi.org/10.1145/3316279.3316280

[7] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High Performance Convolutional Neural Networks for Document Processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition, Université de Rennes 1*.

[8] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv:cs.DC/1512.01274

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[11] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 3393–3404.

[12] NVIDIA Corporation. 2019. Scalability parameters and ConfigROM - NVDLA Open Source Project. http://nvdla.org/hw/v2/scalability.html

[13] G. Desoli, N. Chawla, T. Boesch, S. Singh, E. Guidetti, F. De Ambroggi, T. Majo, P. Zambotti, M. Ayodhyawasi, H. Singh, and N. Aggarwal. 2017. 14.1 A 2.9TOPS/W deep convolutional neural network SoC in FD-SOI 28nm for intelligent embedded systems. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. 238–239. https://doi.org/10.1109/ISSCC.2017.7870349

[14] Farzad Farshchi, Qijing Huang, and Heechul Yun. 2019. Integrating NVIDIA Deep Learning Accelerator (NVDLA) with RISC-V SoC on FireSim. *2019 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)* (Feb 2019). https://doi.org/10.1109/emc249363.2019.00012

[15] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert Ou, Max Banister, Yakun Sophia Shao, Borivoje Nikolic, Ion Stoica, and Krste Asanovic. 2019. Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures. arXiv:cs.DC/1911.09925

[16] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. arXiv:cs.PL/2002.11054

[17] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović. 2014. A 45nm 1.3GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators. In *ESSCIRC 2014 - 40th European Solid State Circuits Conference (ESSCIRC)*. 199–202. https://doi.org/10.1109/ESSCIRC.2014.6942056

[18] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. 2019. A Coordinated Tiling and Batching Framework for Efficient GEMM on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 229–241. https://doi.org/10.1145/3293883.3295734

[19] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. A Hardware-Software Blueprint for Flexible Deep Learning Specialization. arXiv:cs.LG/1807.04188

[20] N. Park, B. Hong, and V. K. Prasanna. 2003. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems* 14, 7 (July 2003), 640–654. https://doi.org/10.1109/TPDS.2003.1214317

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[22] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* 48, 6 (June 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[23] V. Rajagopalan, V. Boppana, S. Dutta, B. Taylor, and R. Wittig. 2011. Xilinx Zynq-7000 EPP: An extensible processing platform family. In *2011 IEEE Hot Chips 23 Symposium (HCS)*. 1–24. https://doi.org/10.1109/HOTCHIPS.2011.7477495

[24] Berkeley Architecture Research. 2020. Chipyard Documentation. https://chipyard.readthedocs.io/_/downloads/en/latest/pdf/

[25] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. 2018. Relay: a new IR for machine learning frameworks. *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages - MAPL 2018* (2018). https://doi.org/10.1145/3211346.3211348

[26] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. arXiv:cs.PL/1805.00907

[27] Mohammadsadegh Sadri, Christian Weis, Norbert Wehn, and Luca Benini. 2013. Energy and Performance Exploration of Accelerator Coherency Port Using Xilinx ZYNQ. In *Proceedings of the 10th FPGAworld Conference (FPGAworld '13)*. Association for Computing Machinery, New York, NY, USA, Article Article 5, 8 pages. https://doi.org/10.1145/2513683.2513688

[28] Colin Schmidt and Adam Izraelevitz. 2013. A Fast Parameterized SHA3 Accelerator. https://people.eecs.berkeley.edu/~kubitron/courses/cs262a-F13/projects/reports/project11_report.pdf

[29] SiFive. 2017. SiFive TileLink Specification. https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf

[30] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. arXiv:cs.PL/1802.04730

[31] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie, and X. Zhou. 2017. DLAU: A Scalable Deep Learning Accelerator Unit on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 3 (March 2017), 513–517. https://doi.org/10.1109/TCAD.2016.2587683

[32] Andrew Waterman and Krste Asanović. 2019. The RISC-V Instruction Set Manual - Volume II: Privileged Architecture. https://riscv.org/specifications/privileged-isa/

[33] P. N. Whatmough, S. K. Lee, H. Lee, S. Rama, D. Brooks, and G. Wei. 2017. 14.3 A 28nm SoC with a 1.2GHz 568nJ/prediction sparse deep-neural-network engine

with >0.1 timing error rate tolerance for IoT applications. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. 242–243. https://doi.org/10.1109/ISSCC.2017.7870351

[34] Xilinx. [n. d.]. Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html

[35] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15)*. Association for Computing Machinery, New York, NY, USA, 161–170. https://doi.org/10.1145/2684746.2689060

[36] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 859–873. https://doi.org/10.1145/3373376.3378508

[37] G. Zhou, J. Zhou, and H. Lin. 2018. Research on NVIDIA Deep Learning Accelerator. In *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*. 192–195. https://doi.org/10.1109/ICASID.2018.8693202