Towards Deep Learning using TensorFlow Lite on RISC-V

Marcia Sahaya Louis
marcia93@bu.edu
Boston University

Zahra Azad
zazad@bu.edu
Boston University

Leila Delshadtehrani
delshad@bu.edu
Boston University

Suyog Gupta
suyoggupta@google.com
Google Inc.

Pete Warden
petewarden@google.com
Google Inc.

Vijay Janapa Reddi
vj@eecs.harvard.edu
Harvard University

Ajay Joshi
joshi@bu.edu
Boston University

Abstract

Deep neural networks have been extensively adopted for a myriad of applications due to their ability to learn patterns from large amounts of data. The desire to preserve user privacy and reduce user-perceived latency has created the need to perform deep neural network inference tasks on low-power consumer edge devices. Since such tasks often tend to be computationally intensive, offloading this compute from mobile/embedded CPU to a purpose-designed "Neural Processing Engines" is a commonly adopted solution for accelerating deep learning computations. While these accelerators offer significant speed-ups for key machine learning kernels, overheads resulting from frequent host-accelerator communication often diminish the net application-level benefit of this heterogeneous system. Our solution for accelerating such workloads involves developing ISA extensions customized for machine learning kernels and designing a custom in-pipeline execution unit for these specialized instructions. We base our ISA extensions on RISC-V: an open ISA specification that lends itself to such specializations. In this paper, we present the software infrastructure for optimizing neural network execution on RISC-V with ISA extensions. Our ISA extensions are derived from the RISC-V Vector ISA proposal, and we develop optimized implementations of the critical kernels such as convolution and matrix multiplication using these instructions. These optimized functions are subsequently added to the TensorFlow Lite source code and cross-compiled for RISC-V. We find that only a small set of instruction extensions achieves coverage over a wide variety of deep neural networks designed for vision and speech-related tasks. On average, our software implementation using the extended instructions set reduces the executed instruction count by 8X in comparison to baseline implementation. In parallel, we are also working on the hardware design of the in-pipeline machine learning accelerator. We plan to open-source our implementation in due course.

Keywords

Deep Learning, RISC-V Vector ISA extension, TensorFlow Lite

1 Introduction

Recent developments in deep learning have led to a resurgence in artificial intelligence. Various cognitive tasks such as image recognition [19, 23], speech recognition [31], and natural language processing [6, 20] extensively use deep neural networks. As these "intelligent applications" pervade into mobile/Internet of Things (IoT) platforms, there is a growing demand for efficient execution of deep neural networks on these low-power and resource-constrained platforms. However, state-of-the-art neural networks routinely have millions of parameters and a single inference task can invoke billions for arithmetic operations and memory accesses. Offloading the neural network execution to a dedicated hardware accelerator has emerged as a widely adopted solution for improving the execution time and energy efficiency. Manifestations of this concept are abundant: the Apple A12 Bionic [27] that has an Integrated Neural Processing Unit, the Qualcomm SD 855 that has a Hexagon DSP [5, 12] and an integrated Neural Processing Unit, Huawei's Kirin 980 SoC that has a Dual Neural Processing Unit [3], and Samsung Exynos 9820, that has an integrated Neural Processing Unit [4].

A heterogeneous solution comprised of accelerators and CPU often requires partitioning the work between the host CPU and the neural accelerator(s) and may trigger frequent host-accelerator communications. Consider a canonical machine learning application that comprises of a) pre-processing the inputs to render them consumable by a neural network, b) running a neural network inference using these inputs, and c) post-processing the predictions generated by the network. The net application-level speed-up is determined by the relative computational complexities of the components listed above as well as the overheads associated with communication between the host and the accelerator. Applications that involve frequent data and/or control exchanges between the host and accelerator land up severely under-utilizing the accelerator and may not see a net benefit of offloading work from the host.

In this paper, we present our work on developing a solution that seeks to eliminate these overheads that surface in a typical

CARRV '19, June 22, 2019, Phoenix, AZ

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

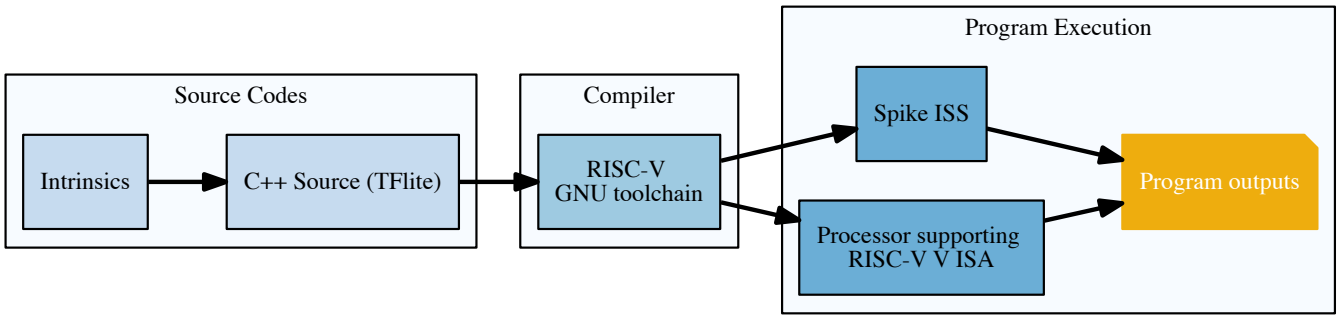

Figure 1: Overview of the Software infrastructure. 'Intrinsics' are implemented using C inline assembly functions.

heterogeneous system. Our solution hinges on developing ISA extensions customized for machine learning kernels and designing a custom in-pipeline execution unit for these specialized instructions. To explore this idea, as a first step, we developed the software infrastructure to support custom domain specific ISA extension for machine learning. We used the open source RISC-V ISA as our target ISA [8, 30]. RISC-V ISA consists of a base Integer (I) ISA which is mandatory for every RISC-V core implementation, and optional extensions to the base ISA. The capability of ISA-level customization provides an opportunity to specialize our processor designs for machine learning workloads.

To effectively accelerate ML on RISC-V processors, our ISA extensions are derived from the RISC-V vector ISA proposal [22]. We selected a subset of the instructions necessary to implement the key machine learning kernels. We developed the tool-chain by augmenting the software environment with the right inline assembly support and building the run-time that can effectively map the high-level macros to the low-level ISA execution. We added basic compiler support for the extended instructions using C inline assembly functions. The C inline assembly functions are used to implement TensorFlow Lite [1] kernel operations such as convolution and matrix multiplication. We added these optimized functions to TensorFlow Lite source code and cross-compiled them for RISC-V target. We modified Spike [7] to support the extended instructions. Subsequently, we used Spike for functional verification and for benchmarking machine learning models. We use the executed instruction count as the metric to compare the modified RISC-V ISA with ARM v-8A with NEON Advanced SIMD extensions [21].

2 Software Environment

We present our infrastructure for building TensorFlow Lite for RISC-V target (Figure 1). As part of the software infrastructure, we have implemented a subset of instructions from RISC-V V ISA extension (draft v0.5) [22]. Table 1 shows the list of supported instructions. These instructions are supported using C inline assembly functions. We provide detailed description of modifications to the compiler tool-chain, Spike and TensorFlow Lite in the following subsections.

2.1 Compiler support for ISA extensions

We use inline assembly functions to enable vector instruction support. The functions are known to the compiler and are mapped to a sequence of one or more assembly instructions. For example, the code snippet in Listing 1 shows the implementation of the vector load template function. The function loads an array of elements to the vector register "**va1**". The number of elements to load is configured at run-time by setting two Control Status Register (CSR), i.e., `vcfg` and `vl` as required by the RISC-V V ISA extension.

Listing 1: A function to load vector elements.

```
template <class T>
inline void __VectorLoadInput(const T* load_address) {
  asm volatile("vls va1, 0(%0), v \t\n"
               : : "r"(load_address));
}
```

The C inline assembly functions are compiled into assembly code using the RISC-V GCC tool-chain. The assembly code is then converted into machine code using GNU assembler (GAS) [11]. GAS is implemented in two sections, the **front-end** that handles the parsing of assembly code and the **back-end** that generates the machine code. We added support for each of the instructions in Table 1 in the GAS front-end to parse the extended instructions and check if the instruction has a valid opcode and operands. Subsequently, the GAS back-end generates the corresponding machine code for the extended instructions. We then modified the Spike ISA simulator to verify the functionality of the extended instructions.

2.2 Instruction simulation support on Spike ISS

Spike is a RISC-V Instruction Set Simulator (ISS) [7] and implements a functional model of RISC-V processor. Spike is a functional simulator that ignores internal delays such as I/O accesses or memory transactions. Therefore, the simulations are not cycle accurate. Spike executes a user space program using proxy kernel for handling the system calls from a C standard library functions.

To support the simulation of the instructions in Table 1, we modified the Spike simulator. We extended the `class regfile_t` with vector registers and macros to read/write values to the registers. In order to load/store data from memory, we extended the `class mmu_t` with macros for loading/storing multiple data from memory. Similar to the scalar pipeline, a memory request is handled by the TLB unit in Spike.

We also modified the `class processor_t` to configure the two vector CSRs; `vcfg` CSR and `vl` CSR. As specified in RISC-V V ISA extension [22], the `vcfg` CSR configures the vector unit by setting the highest number of enabled vector registers in `vregmax` CSR and

Table 1: The subset of RISC-V Vector ISA extension [22] implemented in our software ecosystem.

Inst. Type	Instructions	Function
Memory access	vls{b,h,s,d} VR_d, RS_1, RS_2, m vlx{b,h,s,d} VR_d, RS_1, VRS_2, m	Loads a vector into VR_d from memory address in RS_1 with unit/const stride in RS_2 or indexed stride in VRS_2
	vss{b,h,s,d} VRS_3, RS_1, RS_2, m vsx{b,h,s,d} VRS_3, RS_1, VRS_2, m	Stores a vector in VRS_3 to memory address in RS_1 with unit/const stride in RS_2 or with indexed stride in VRS_2
Arithmetic Instructions	vadd VR_d, VRS_1, VRS_2, m vmul VR_d, VRS_1, VRS_2, m vfadd VR_d, VRS_1, VRS_2, m vfmul VR_d, VRS_1, VRS_2, m	Add/Multiply values in VRS_1, VRS_2 and writes to VR_d
	vmadd VR_d, VRS_1, VRS_2, VRS_3, m vfmadd VR_d, VRS_1, VRS_2, VRS_3, m	Multiply values in VRS_1,VRS_2 and add VRS_3, and writes to VR_d
	vmax VR_d, VRS_1, VRS_2, m vmin VR_d, VRS_1, VRS_2, m vfmax VR_d, VRS_1, VRS_2, m vfmin VR_d, VRS_1, VRS_2, m	Element-wise maximum/minimum of values in VRS_1, VRS_2 and writes to VR_d
Data Movement	vsplat VR_d, VRS_1, RS_2, m vbcastx VR_d, RS_1 vbcastf VR_d, FRS_1	Splats the element in VR1[RS_2] to VR_d Broadcasts value in RS_1/FRS_1 to VR_d
	vredsum VR_d, VRS_1 vredmin VR_d, VRS_1 vredmax VR_d, VRS_1 vfredsum VR_d, VRS_1	Reduction of VRS_1 based on sum/max/min, broadcast and store the result to VR_d

VR_d: Vector destination registers
$VRS_{1,2,3}$: Vector source registers,
m: Two bit encoding for masking; m=00 -> scalar shape destination, m=01 -> unmasked vector operation, m=10 -> mask enabled where v1.LSB=0, m=11 -> mask enabled where v1.LSB=1; here v1 is the mask register.

the maximum width of elements in `vemaxw` CSR. The `vl` CSR holds the current active vector length. Finally, we added support in Spike for all the instructions in Table 1. Listing 2 is an example of implementation *vadd* instruction in Spike. These modification enabled simulation of the vector instructions. We added functionality to Spike interactive debug mode to facilitate tracing and debugging.

Listing 2: Implementation of vadd instruction in Spike.

```
require_extension('V');
require_rv64;

WRITE_VRD(v_add(VRS1, VRS2, EW, insn.m(), VMASK, VL));
```

2.3 RISC-V target for TensorFlow Lite

TensorFlow Lite is a lightweight deep learning framework for mobile and embedded devices [1]. It compresses a TensorFlow model to a *.tflite* model that has a small binary size. This enables on-device machine learning and uses hardware acceleration to improve performance. The TensorFlow Lite source code has two implementations; `reference_ops` and `optimized_ops`, for machine learning kernels such as convolution and depthwise-convolution. The `reference_ops` implementation is portable, hardware-independent and uses standard C/C++ libraries. The `optimized_ops` is a hardware specific optimized implementation of kernel operations using *gemmlowp*, *Eigen* libraries [13, 18] and other processor specific optimizations. For example, in the case of ARM processors, the `optimized_ops` implementation leverages *gemmlowp*, *Eigen* libraries and Neon instructions [21] to optimize kernel operations.

To support RISC-V target for Tensorflow Lite, we modified some functions in `reference_ops` to remove library dependencies not supported by *Newlib* [1] [29]. This made the `reference_ops` implementation portable and capable of running on mobile and embedded device with RISC-V processors. The C inline assembly functions were used for constructing SIMD-aware optimized functions to be used in `optimized_ops` implementation for RISC-V vector processors. Listing 3 shows the implementation of a function that performs element-wise addition of two arrays. Using the instructions in Table 1, we can support a wide range of machine learning models.

We cross-compiled the TensorFlow Lite source code for RISC-V ISA and executed *.tflite* models on Spike. With the infrastructure in place, we generate binary that can run on a RISC-V processor that has micro-architectural support for the RISC-V V ISA extension.

[1] C standard library implementation intended for use on embedded system

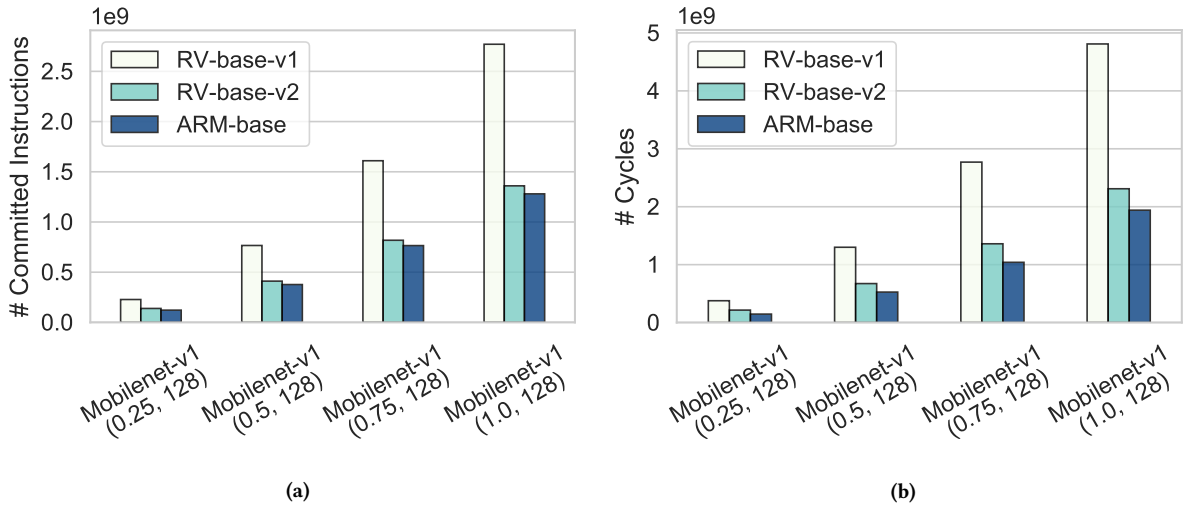

Figure 2: Comparison of committed instructions, cycles and IPC for *ARM-base, RV-base-v1* without loop optimization and *RV-base-v2* with loop optimization for four variants of MobileNet [14]. Here, Mobilenet-v1 (0.25, 128) means MobileNet-V1 model for input size of 128x128 pixels and 0.25 depth multiplier. The depth multiplier changes the number of channels in each layer.

Listing 3: A example function for element-wise addition of two arrays.

```
void VectorVectorAdd(const float* input1,
                     const float* input2,
                     float* output, int len) {
  int new_len = len - (len & (kMaxVectorLength32 - 1));
  int len_diff = len & (kMaxVectorLength32 - 1);

  SetConfig(kElementWidthMax32, kMaxVectorLength32);

  for (int i = 0; i < new_len; i += kMaxVectorLength32) {
    __VectorLoad((input1 + i), (input2 + i));
    __VectorAddFloat();
    __VectorStore((output + i));}

  if (len_diff != 0) {
    SetVl(len_diff);
    __VectorLoad((input1 + new_len), (input2 + new_len));
    __VectorAddFloat();
    __VectorStore((output + new_len));}
}
```

3 Evaluation

In this section, we evaluate the code optimizations for RISC-V and compare it with ARM processors, as ARM processors are the most commonly used processors for mobile systems. For comparison purpose we define the Region Of Interest (ROI) as the execution of *interpreter->Invoke()* function in TensorFlow Lite. The deep learning models [14–17, 25, 26] used in our evaluation are listed in Table 2. These are commonly used machine-learning inference models that are deployed on mobile devices. We cover a wide range of applications using these benchmark models. The models are 32-bit floating point *.tflite* models and are hosted on TensorFlow Lite website [2].

To evaluate the performance of deep learning models listed in Table 2 for ARM processor, we used gem5 [9] in full system mode with ARM A-class, 4-stage pipeline High Performance In-order (HPI) core configuration [28]. The ARM HPI was configured with 16KB L1 I$, 16KB L1 D$ and without L2$ [2]. In this section, we will use term *ARM-base* for the baseline implementation of TensorFlow Lite using `reference_ops`, and *ARM-opt* for the implementation of TensorFlow Lite using `optimized_ops`. We inserted *m5_reset_stats* and *m5_dump_stats* functions in TensorFlow Lite source code to get gem5 performance stats for ROI. We used number of cycles and committed instructions as our performance metrics for evaluation.

For RISC-V, *RV-base* and *RV-opt* represents the RISC-V cross-compiled binaries of TensorFlow Lite using `reference_ops` and `optimized_ops`, respectively. We mapped a in-order 5-stage pipeline Rocket core [7] to Zedboard [10] to evaluate the performance of benchmarks in Table 2 for *RV-base*. The Rocket core is configured with 16KB L1 I$, 16KB L1 D$ and without L2$, as the current version of Rocket chip does not support L2$. We used hardware performance counters, specifically the `cycle` CSR and `instret` CSR for evaluation. Currently, the microarchitecture enhancement to Rocket-chip processor for supporting extended instructions in Table 1 is in the 'pre-pre-alpha stage'. For this paper we use Spike to benchmark number of committed instructions of deep learning benchmarks listed in Table 2 for *RV-opt*.

ARM-base and *RV-base* are cross-complied from the same source code. Although in both cases, we used "-O3" compiler flag, we noticed that the number of committed instructions and corresponding cycles in ROI were higher for *RV-base* as compared to *ARM-base*. Figure 2 shows the number of committed instructions and number of cycles for four variants of MobileNet [14] model used in

[2]We simulated ARM core without L2$ to perform a fair comparison with RISC-V Rocket core

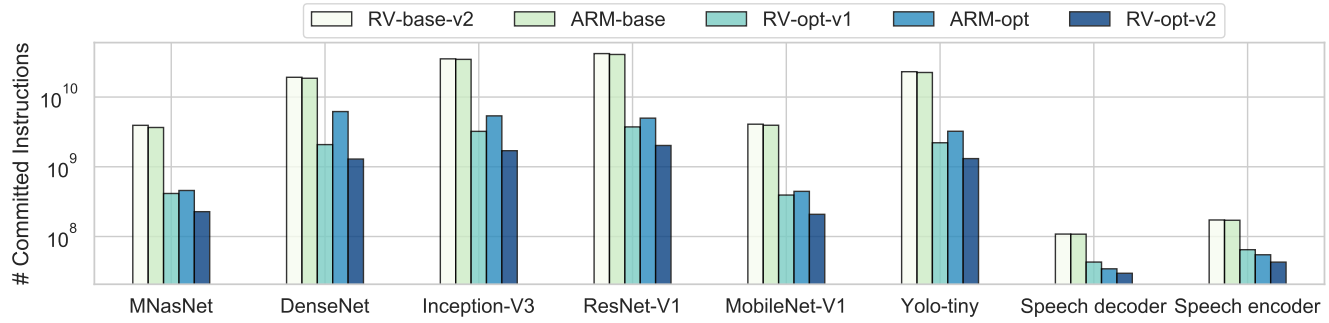

Figure 3: Number of committed instructions for *RV-base-v2, ARM-base, RV-opt-v1* optimized with 128bits registers, *ARM-opt* and *RV-opt-v2* optimized with 256bit registers for various deep learning models.

Table 2: List of deep learning models using in our evaluation. CONV = Convolution layer, LSTM = Long Short Term Memory.

Application	Model	Dominant layer
AutoML	MnasNet variants	CONV
Image Classification	DenseNet,	CONV
	Inception_V3,	CONV
	ResNet_50,	CONV
	MobileNet variants,	CONV
Object Detection	Yolo_tiny	CONV
Speech Recognition	Speech encoder/decoder	LSTM

image classification workload. Figure 2 show the number of committed instructions and cycles are ~2X higher for *RV-base-v1* in comparison to *ARM-base*. Here, *RV-base-v1* corresponds to cross-complied from TensorFlow Lite `reference_ops`. The difference in instruction and cycle count is due to the difference in the compiler optimizations. As ARM cross-compiler has matured over the years, the compiler optimizes a nested loops in source code such that the inner-most loop has few instructions. We updated the source code to replicate the compiler loop optimizations. We refer to this updated version as *RV-base-v2*. As shown in Figure 2a and 2b, the loop optimization reduced the number of committed instructions and cycles for *RV-base-v2*, and these numbers are now comparable to that of *ARM-base*. For the rest of our analysis we will use *RV-base-v2* and *ARM-base* as our baseline implementations.

We next compare *ARM-opt* and *RV-opt* implementations using the number of committed instructions for the deep learning model listed in Table 2. *ARM-opt* implements ARM Neon extension [21]. ARM Neon extension has a fixed SIMD width of 128bits. The benchmark models use single precision floating point, therefore the processor operates on 4 single floating precision values in one instruction. We fixed the RISC-V vector register width to be 128bits for a fair comparison with ARM processor with Neon extension. Also, we evaluated the setup for vector register width of 256bits. Figure 3 shows the comparison of *ARM-base*, *RV-base-v2*, *ARM-opt* and *RV-opt-v1* with 128bits register widths and *RV-opt-v2* 256bits register widths using deep learning models in Table 2. As expected, the number of committed instructions are similar (across all the models) for *ARM-base* and *RV-base-v2*. On average, across all benchmarks the number of committed instructions for *RV-opt-v1* is 1.25X lower than the *ARM-opt*. In deep learning models where 'CONV' are the dominant layers, *RV-opt-v1* has consistently less instructions than *ARM-opt*. In the case of models where LSTM layers are dominant, *ARM-opt* has consistently less instructions than *RV-opt-v1*. This is because of difference in code optimization for ARM and RISC-V. *ARM-opt* implementation uses block vector-matrix multiplication for LSTM layers. The instruction count for *RV-opt-v1* can be improved by implementing block vector-matrix multiplication.

On average, we achieved a 8X reduction on number of committed instructions using *RV-opt-v1* implementation in comparison to *RV-base*. We see an additional ~2X reduction in the number of committed instructions using *RV-opt* with 256bits register width.

4 Summary and Future Work

In this paper, we present the software infrastructure we developed to support compilation and execution of machine learning models used in TensorFlow Lite framework. We are able to support a large range of machine learning applications using a subset of RISC-V Vector instructions. On average, we are able to reduce the number of committed instructions by 8X using *RV-opt* implementation in comparison to the *RISC-V reference* implementation.

In our current software pipeline, we handle register naming and register allocations. Moving forward we want the compiler to handle this task. To enable the compiler to do this task, we need to support the new instructions in GCC using intrinsics. The GCC compiler has three stages, the front-end, middle-end and back-end [24]. At a high level, the **front-end** generates a *parse-tree* from the input program, the *parse-tree* is used by **middle-end** to generate a *generic-tree*, and the **back-end** converts the *generic-tree* to *assembly* code. As part of the future work, we will modify the front-end to include the specification of the intrinsics in GCC source code. We will also modify the back-end of GCC to create machine description of the new instructions and generate the assembly code.

We are developing in-pipeline microarchitectural support for a subset of the vector instructions for machine learning accelerator. Our microarchitecture will include dedicated vector registers, vector caches and support for multiple precision arithmetic and logical

operations. Additionally, we will explore sparsity-aware microarchitecture design for the in-pipeline accelerator. As we develop the in-pipeline accelerator we will modify the subset of instructions in Table 1 as needed and update the software tool-chain accordingly. We will evaluate our design in terms of performance, power and area. We will open source our microarchitecture design to the wider community in due course.

References

[1] 2017. TensorFlow Lite | TensorFlow. https://www.tensorflow.org/lite

[2] 2018. Hosted models | TensorFlow Lite | TensorFlow. https://www.tensorflow.org/lite/guide/hosted_models

[3] 2018. Huawei Kirin 980. https://consumer.huawei.com/en/campaign/kirin980/

[4] 2019. Exynos 9 Series 9820 Processor. https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-9-series-9820/

[5] 2019. Snapdragon 855 Mobile Platform. https://www.qualcomm.com/products/snapdragon-855-mobile-platform

[6] Ahmad Abdulkader, A Lakshmiratan, and J Zhang. 2016. Introducing DeepText: FacebookâĂŹs text understanding engine. *Facebook Code* (2016).

[7] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).

[8] Krste Asanović and David A Patterson. 2014. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[10] Louise H Crockett, Ross A Elliot, and Martin A Enderwitz. 2015. *The zynq book tutorials for zybo and zedboard.* Strathclyde Academic Media.

[11] Dean Elsner, Jay Fenlason, et al. 2000. *Using as: the GNU Assembler.* IUniverse Com.

[12] Andrei Frumusanu. 2018. The Qualcomm Snapdragon 855 Pre-Dive: Going Into Detail on 2019's Flagship Android SoC. https://www.anandtech.com/show/13680/snapdragon-855-going-into-detail/2

[13] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

[14] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[15] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 4700–4708.

[16] Rachel Huang, Jonathan Pedoeem, and Cuixian Chen. 2018. YOLO-LITE: A Real-Time Object Detection Algorithm Optimized for Non-GPU Computers. In *2018 IEEE International Conference on Big Data (Big Data).* IEEE, 2503–2510.

[17] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[18] Benoit Jacob and Pete Warden. 2017. gemmlowp: a small self-contained low-precision GEMM library.

[19] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436.

[20] William D Lewis. 2015. Skype translator: Breaking down language and hearing barriers. *Translating and the Computer (TC37)* 10 (2015), 125–149.

[21] Venu Gopal Reddy. 2008. Neon technology introduction. *ARM Corporation* 4 (2008), 1.

[22] Riscv. 2019. riscv/riscv-v-spec. https://github.com/riscv/riscv-v-spec

[23] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural networks* 61 (2015), 85–117.

[24] Richard M Stallman. 2002. GNU compiler collection internals. *Free Software Foundation* (2002).

[25] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2818–2826.

[26] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V Le. 2018. Mnasnet: Platform-aware neural architecture search for mobile. *arXiv preprint arXiv:1807.11626* (2018).

[27] Techinsights.com. 2018. Apple iPhone Xs Max Teardown. https://www.techinsights.com/about-techinsights/overview/blog/apple-iphone-xs-teardown/

[28] Ashkan Tousi and Chuan Zhu. 2017. Arm Research Starter Kit: System Modeling using gem5. (2017).

[29] Corinna Vinschen and Jeff Johnston. 2013. Newlib.

[30] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanovic. 2014. The RISC-V instruction set manual. *volume I: User-level ISA, version 2.0, EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54* (2014).

[31] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).