An end-to-end RISC-V solution for ML on the edge using in-pipeline support

Zahra Azad[1], Marcia Sahaya Louis[1], Leila Delshadtehrani[1], Anthony Ducimo[1],
Suyog Gupta[2], Pete Warden[2], Vijay Janapa Reddi[3], and Ajay Joshi [1]
[1]Boston University, [2]Google Inc., [3]Harvard University

***Abstract*—Machine Learning (ML) is widely used today in many mobile applications. To preserve user privacy, there is a need to perform ML inference on the mobile devices. Given that ML inference is a computationally intensive task, the common technique used in mobile devices is offloading the task to a neural accelerator. However, the speed-up gained from offloading these tasks on the accelerators is limited by the overhead of frequent host-accelerator communication. In this paper, we propose a complete end-to-end solution that uses in-pipeline machine learning processing unit for accelerating ML workloads. First we introduce the software infrastructure we developed to support compilation and execution of machine learning models used in TensorFlow Lite framework. Then we discuss the micro-architecture we plan to implement for supporting the execution of our vectorized machine learning kernels.**

I. Introduction

Over the past few years, researchers have shown that Machine Learning (ML) is the best solution for a variety of mobile applications such as image recognition, speech recognition, and natural language processing, leading to a wide adoption and development of ML-based solutions. Although ML have been extensively adopted in many mobile applications, ML-based applications are power hungry and so most of the computations of these applications are offloaded to the cloud. However, to preserve user privacy and reduce user-perceived latency due to offloading of computations through a bandwidth-limited wireless link, there is a need to perform ML inference on the resource-constrained mobile devices.

To accelerate the computationally intensive tasks in ML-based applications, the tasks are commonly offloaded to accelerators like GPUs, DSPs, or special-purpose accelerators like Neural Processing Units (NPUs) in the mobile SOCs. This heterogeneous approach divides the task between the host CPU and the accelerator, which requires us to use special instructions to access the accelerator during the program execution. Depending on the application, this may trigger frequent host-accelerator communication. In these solutions the whole ML task is partitioned into three main steps: 1) pre-processing the input data to make it consumable by the accelerator; 2) running part of the application (for e.g. neural inference) on the accelerator using the input data; and 3) post-processing the results generated by the accelerator. The time spend on pre-processing and post-processing operations and the time required to transfer data between the processor and the accelerator makes this approach suitable only for applications that have large amount of computations that can be offloaded to the accelerator. Applications with a small amount of computations that can be offloaded to the accelerator cannot sufficiently amortize the communication overhead. Moreover, applications that involve frequent data and/or control exchanges between CPU and accelerator end up severely under-utilizing the accelerator and may not see any benefit of offloading work from the CPU to the accelerator.

In this paper we propose to create a custom processor configuration by extending the processor architecture with a special-purpose processing unit (similar to NEON [1], which is an Advanced SIMD extension for ARM processors) to speed up the execution of ML-based applications such as audio and video processing, voice and facial recognition, computer vision and deep learning on the processor. As mentioned in [2] today nearly all mobile inference on Android devices run on CPU rather than being offloaded to a co-processor or accelerator. The main reason is that performance gain is not substantial enough to justify the effort required to port all inference operations to a co-processor or accelerator. In fact, having a CPU with a SIMD unit which is decently provisioned and properly programmed provides sufficient performance for inference.

As the first step to explore this idea, we developed the software stack for supporting custom machine learning instructions. We use RISC-V [3] as our target ISA because it is open source and has specific regions of instructions decode space allocated to user custom instructions. We used a subset of the vector instructions to cover the key machine learning kernels. We developed the required inline assembly support and built the run-time environment to map TensorFlow Lite [4] kernel operations such as convolution and matrix multiplication to the low-level ISA execution.

As the next step, we plan to take this software stack and prepare an end-to-end solution to accelerate the execution of ML-based applications on the main processor. We will develop a RISC-V based in-pipeline ML processing Unit called RV-MLPU to accelerate DNN inference tasks on the main processor to avoid the overhead of communicating with an external accelerator. This in-pipeline unit will be implemented as a custom extension to Rocket Core, an in-order RISC-V based processor.

II. Software Environment

We developed the software infrastructure to support custom domain-specific ISA extension for ML. We used the open source RISC-V ISA as our target ISA and our ISA extensions are derived from the RISC-V vector ISA proposal. We selected a subset of the instructions necessary to implement the key ML 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 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.[1] We modified Spike [5], an instruction set simulator, to support the extended instructions. Subsequently, we used Spike for functional verification and for benchmarking ML models. All our work has been open sourced. [2]

III. Evaluation and Results

We evaluated the performance of deep learning models such as DenseNet, MnasNet, Inception V3, ResNet 50, MobileNet, Yolo tiny and Speech encoder/decoder. These are commonly used ML 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. 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. Figure 1 shows the comparison of ARM baseline implementation *(ARM-base)*, RISC-V baseline implementation *(RV-base)*, *ARM-opt* and *RV-opt-v1* with 128bits register widths and *RV-opt-v2* 256bits register widths using the mentioned deep learning models. In the *RV-base* implementation we have updated the source code to replicate the compiler loop optimizations that are available in ARM. In *RV-opt* and *ARM-opt*, we used RISC-V cross-compiled binaries and ARM cross-compiled binaries, respectively, of TensorFlow Lite using `optimized_ops`. The `optimized_ops` is a hardware specific optimized implementation of kernel operations using *gemmlowp*, *Eigen* libraries and other processor specific optimizations.

As expected, the number of committed instructions are similar (across all the models) for *ARM-base* and *RV-base*. Using *RV-opt-v1* implementation, on average, we achieved a 8× reduction in number of committed instructions in comparison to *RV-base*. Compared to *ARM-opt*, on average, across all benchmarks the number of committed instructions for *RV-opt-v1* is 1.25× lower. We see an additional 2× reduction in the number of committed instructions using *RV-opt* with 256bits register width. A more detailed discussion about these results can be found in our prior work [6].

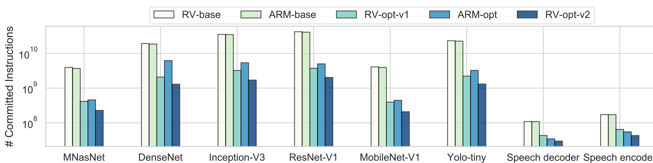

Fig. 1: Number of committed instructions for *RV-base*, *ARM-base*, *RV-opt-v1* optimized with 128bits registers, *ARM-opt* and *RV-opt-v2* optimized with 256bit registers for various deep learning models.

[1] https://github.com/mars20/tensorflow

[2] https://github.com/bu-icsg/sparse-mat-risc-v

IV. RV-MLPU Microarchitecture

In this section, we explain the microarchitecture of RV-MLPU, which basically is a configurable RISC-V based SIMD unit. We support a subset of instructions from the V-extension of the RISC-V ISA. In RV-MLPU microarchitecture design the number of lanes and the width and number of data elements are configurable. RV-MLPU is implemented as a SIMD extension to Rocket Core processor. We extended Rocket Core decoder to support the subset of vector instructions. We also integrated 32 vector registers to Rocket Core. Since each vector instruction operates on multiple data elements, vector register file must be able to support enough throughput to supply the functional lanes with operands. Once the vector instruction is fetched and decoded, the required registers are read from both scalar and vector register files, and the register values are passed down to the Vector Processing Unit (VPU) to perform the required operations. VPU is a configurable unit, and it can be configured with both the number of lanes and the width of the data elements to be processed. Each lane contains one Floating Point unit and all lanes have identical microarchitecture. We also modified the DCache implementation in Rocket Core to support higher memory bandwidth for vector load and store operations. RV-MLPU is equipped with a vector load/store unit to handle three different vector memory accesses: 1) unit-stride loads and stores, which access a contiguous chunk of memory; 2) constant-stride loads and stores, which access memory addresses spaced with a fixed offset; and 3) scatter and gather memory operations, which use a vector of offsets to allow general access patterns. We are currently testing our microarchitecture implementation and plan to open source it in the near future.

V. Conclusion

In this paper, to accelerate ML-based applications on mobile devices, we present an end-to-end support for in-pipeline execution of the ML operations in RISC-V based processors. First, we discussed the software infrastructure we developed to support compilation and execution of machine learning models (used in TensorFlow Lite framework) using a subset of RISC-V vector extension. As the next step, we plan to complete the micro-architecture implementation of the machine learning processing unit, RV-MLPU, to support the execution of our vectorized machine learning kernels in the pipeline.

References

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

[2] C. Wu et al., Machine Learning at Facebook: Understanding Inference at the Edge, 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), Washington, DC, USA, 2019, pp. 331-344.

[3] Waterman et al. The risc-v instruction set manual. volume 1: User-level isa, version 2.0. Technical report, 2014.

[4] 2017. TensorFlow Lite — TensorFlow. https://www.tensorflow.org/lite

[5] Krste Asanovic, et al. The rocket chip generator. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17.

[6] M. S. Louis, Z. Azad, L. Delshadtehrani et al., "Towards Deep Learning using TensorFlow Lite on RISC-V," in Proc. ACM CARRV, 2019.