Extending a Soft-Core RISC-V Processor to Accelerate CNN Inference

Ross Porter
Department of Electrical, Computer & Software Engineering
University of Auckland
Auckland, New Zealand
rpor545@aucklanduni.ac.nz

Sam Morgan
Department of Electrical, Computer & Software Engineering
University of Auckland
Auckland, New Zealand
smor264@aucklanduni.ac.nz

Morteza Biglari-Abhari
Department of Electrical, Computer & Software Engineering
University of Auckland
Auckland, New Zealand
m.abhari@auckland.ac.nz

***Abstract*—Convolutional Neural Networks (CNNs) are the gold-standard for computer vision. Using CNN on embedded hardware that has limited computational capability is an area of active investigation and optimization. In this paper, we investigate the potential of extending the RISC-V Instruction Set Architecture for accelerating the inference of a CNN using in-pipeline hardware blocks and custom instructions. Our preliminary designs have a small footprint and minimal impact on maximum core frequency. The new designed instructions were used to extend an existing soft-core processor. This processor was synthesized to an FPGA for cycle-accurate testing and performance evaluation.**

Keywords—RISC-V, CNN, FPGA, Soft-core processor

I. Introduction

Computer vision has been in the spotlight over the past few years, driving innovation in many fields, including autonomous vehicles, automated assembly robots and 'smart homes' to name a few. Recently, computer vision systems have usually been implemented by using a Convolutional Neural Network (CNN) [1], however CNNs tend to be resource-hungry, as each output pixel in each layer requires many multiplications and additions. This makes employing such a system difficult when computational resources are limited, such as in an embedded environment.

RISC-V is an open-source Instruction Set Architecture (ISA) that was chosen as the target system due to: 1) the modular nature of the ISA makes it suitable for embedded systems and 2) the open-source nature makes academic extension of the ISA trivial compared to closed-source ISAs.

II. Related Work

Related literature includes [2], which proposes the hardware architecture for a low-power multi-core RISC-V processor including SIMD and DSP instructions, [3] also overhauls the RISC-V specification with a linear algebra accelerator. Both of these solutions come with a large trade-off in size and complexity, and cannot easily be interfaced with another RISC-V core. Off-chip accelerators, such as [4], are easier to interface with, but adding an off-chip component increases bus contention, requires more area and increases communication delays.

In a similar paper published recently, [5] implements a set of instructions inspired by RISC-V's work-in-progress 'V' vector extension. They focus on using the RISC-V simulator, Spike [6], and the work required to integrate this with TensorFlow Lite. We will explore a similar goal but will use soft-core processors instead of using Spike (for cycle-accurate testing and evaluation) and will consider the hardware blocks required for each instruction.

In this paper, we aim to create small-footprint, low-overhead instructions for speeding up CNN inference. The relative simplicity of the hardware should allow for easy integration with existing cores, while minimizing the impact on maximum clock frequency.

III. CNN Theory

The convolution operation takes in *M* input layers and produces *N* output layers using M×N weight matrices. Typically N > M, but the size of each output layer is usually smaller than the input layer. The weight matrices are usually 3x3 or 5x5, whereas the input and output layers are large enough to represent an image, so typically between 30x30 up to 200x200 or more.

To calculate all values in an output layer, a window (whose size matches the weight matrix) is passed over all input layers. The values inside the window are multiplied pointwise with the values in the associated weight matrix. This produces a matrix for each input layer. Each of these matrices is then summed together to produce a scalar, and all of these scalars are added together to produce one output layer pixel. This window is then moved along the input layers by some number of pixels (called the 'stride', which can be controlled to dictate the size of the output layer), and the process is repeated until the window reaches the end of the input layers. This whole process is then repeated for each output layer. Each output layer uses a different set of weight matrices. We can see that, considering only one output layer, the operation is of the form $c_{ij} = \sum_{m=i}^{i+k} \sum_{n=j}^{j+k} a_{mn} \times b_{mn}$ where c_{ij} is the output value in the *i*th row and the *j*th column, and likewise *a* is a value from the input layer and *w* is a value from the weight matrix, and *k* is the weight matrix size. Given the success of CNN architectures like VGGNet [7] that use only 3x3 kernels, to simplify the implementation, we will assume a kernel size of 3x3.

IV. Implementations

A. Multiply Accumulate (MAC)

One of the simplest ways to accelerate the convolution operation is to use the concept of accumulation, where the new value is the sum of the old value and something else: $c = c + k$. In this case, we would set $k = a_{mn} \times w_{mn}$. This combines a multiplication and addition into one atomic instruction. The RTL block diagram can be seen in *Figure 1*.

However, multiplication is a costly operation, and is likely to be in a critical path of the CPU, so doing both a multiplication and an addition in a single cycle may not result in performance improvements due to the possible impact on the critical path of the processor. Instead, this is done over two cycles. In the case that the result of the MAC instruction is required in the next instruction, the result is not ready at the end of the execute stage, so the result cannot be bypassed to the next instruction. The pipeline must be stalled for one

978-1-7281-5584-5/19/$31.00 ©2019 IEEE
DOI 10.1109/CSCI49370.2019.00130

cycle, which effectively nulls out any benefit of combining the multiply and add instructions in the first place. We chose the two-cycle approach because in the context of CNN inference, it is possible to reorder the instructions such that the result is never needed immediately after the MAC instruction.

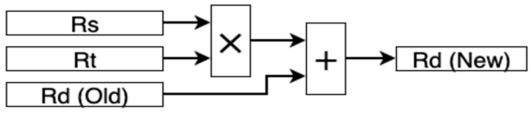

Figure 1 - RTL Block Diagram of the MAC Instruction. All busses are 32-bits wide.

B. SIMD Multiply-Accumulate (SMAC)

Vectorization of the MAC instruction is a clear pathway to improving performance. In order to vectorize, either the working elements must be made smaller or the vector registers must be made larger – we chose to reduce the size of the working element to 8-bit integers. We chose this route for several reasons: 1) TensorFlow Lite, a framework for performing machine learning (ML) on mobile and embedded hardware supports full quantization of most models to 8-bit integers [8]. 2) By moving to integer elements, the RISC-V floating-point extension is not required, bringing the minimum requirements down to RV32IM, which will likely be a popular instruction set for embedded devices. 3) Having to include large (e.g. 128-bit or larger) vector register(s) would increase the footprint of the extension. 4) The general-purpose registers can now be used as vector registers, removing the need for dedicated vector registers at all. 5) By reducing the size of the multiplication from 32-bit to 8-bit, the multiply-accumulate operation is likely not on the critical path, even if it is done in one cycle instead of two, improving performance.

Due to the geometry of the 3x3 kernel and the respective image window, it was found that operating on three element pairs was faster than operating on four: If misaligned memory accesses are valid, then four 8-bit elements can be loaded at once with a 'load word' instruction, however due to the 3x3 window size, the last element is outside the region of interest and must be masked out before the register is operated on. However, these masking instructions can be removed if the SMAC instruction ignores the last element pair. If misaligned memory accesses are not allowed (or slower than individual loads), then operating on either three or four element pairs requires the same number of instructions. Additionally, operating on four element pairs would also require a third stage of adders, which increases the footprint of the design, and may be difficult to complete in one cycle. Note that any row of the kernel can be loaded using a single aligned memory access by padding the kernel to a size of 4x3.

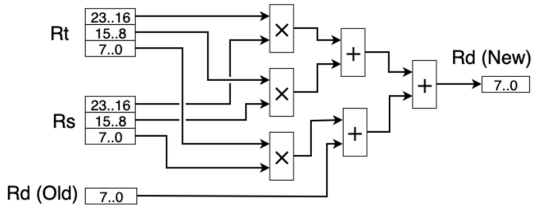

Figure 2 - RTL Block Diagram of SMAC Instruction

C. SIMD Multiply-Accumulate with fixed Kernel (SMACK)

For a given output layer and input layer, the kernel is unchanging, so rather than loading it repeatedly, it can be loaded into a special register file once and henceforth used as needed, reducing memory accesses and general-purpose register contention. However, as noted in the previous section, most of the memory accesses required for an operation involve loading the image values, as the respective kernel values can always be loaded in one aligned memory access, so time savings are not as large as what might be expected.

The special register file consists of nine 8-bit registers arranged in a 3x3 grid, a row can be selected by providing the appropriate input to the register file.

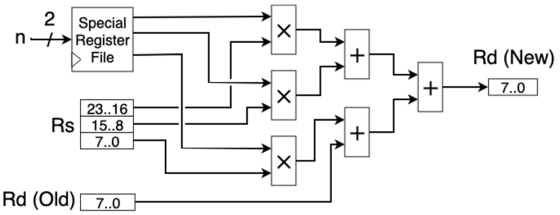

Figure 3 - RTL Block Diagram of SMACK Instruction

D. Integration

These instructions were added to the RISC-V GCC compiler/assembler so that they could be used in inline assembly sections where required. Automatic compiler inference was deemed out of scope for this work: Because most modern machine learning is done with libraries and frameworks such as TensorFlow, these instructions could be integrated into such libraries, making it transparent to end users, which removes most of the need for automatic inference.

V. Hardware Setup

To get cycle-accurate results, the custom instructions were integrated with an existing soft-core processor, which will be synthesized and programmed onto an FPGA. Given the number of soft-core RISC-V processors available [9], various requirements were considered, including the extensibility of the core, the amount and quality of documentation, FPGA compatibility, and the activeness of development and the community. A shortlist was constructed and the VexRiscv core [10] was chosen as the most suitable core to extend. The core's architecture is extremely modular and is well-suited to experimentation.

The DE2-115 Development Kit with an Intel (Altera) Cyclone IV FPGA was used in these experiments.

VI. Benchmark

At the time of writing, TensorFlow Lite does not work out-of-the-box with RISC-V cores running Newlib, but [5] has shown that it is possible to add compatibility. As their changes are not yet released, rather than reinventing the wheel, we decided to create a small custom benchmark which is representative of a convolutional workload. This decision was further informed by the fact that the DE2-115 board that will be used for cycle-accurate testing has limited memory – likely too small to fit a decently sized CNN, like those

available in TensorFlow Lite. By creating a custom benchmark, its size can be tailored to fit into the available memory while still being a reasonably representative of a real-world workload.

The custom benchmark models a single convolutional layer with input layer size 28x28x3, a kernel of size 3x3, and 47 output layers. No padding was used, and a stride of 1 was applied. The input layer size was chosen to imitate a CNN operating on the MNIST [11] dataset, which has the smallest images that would realistically be used on a CNN. The number of output layers was maximized to use up the rest of the available space in the FPGA.

Timing was done using the `rdcycle` and `rdinstret` pseudo-instructions, which read the CSR registers that contain the number of cycles taken, and instructions retired, respectively. This allows for cycle-accurate timing and allows for calculation of metrics like cycles per instruction (CPI). Reading these values proved to have negligible overhead relative to the size of the benchmark.

The benchmark was written in C, and inline assembly was used to call the custom instructions as necessary. It was compiled using the RISC-V GCC compiler.

VII. Results

For each implementation, the hardware was synthesized using Intel (Altera) Quartus 13.0. Various metrics (logic element usage, maximum frequency, etc.) were recorded, then the soft-core processor (with the benchmark loaded in RAM) was flashed to the Cyclone IV FPGA on the DE2-115 SoC. The unmodified VexRiscv core (referred to as 'Stock'), and our extensions thereof were run at the same frequency. The maximum frequency of each implementation can be seen in Table 1 under 'Fmax'. The benchmark was run several times for each hardware implementation. The number of cycles and instructions elapsed were then displayed over UART. Variance between test runs was very small – no more than 50 cycles. When compared to the size of the benchmark (several million cycles), this variance was negligible.

This was performed with no compiler optimization (-O0), then repeated with (-O3) optimization. A third test was also performed with optimization enabled and with a CNN of different dimensions (3 input layers, a 100x100 image, 3 output layers).

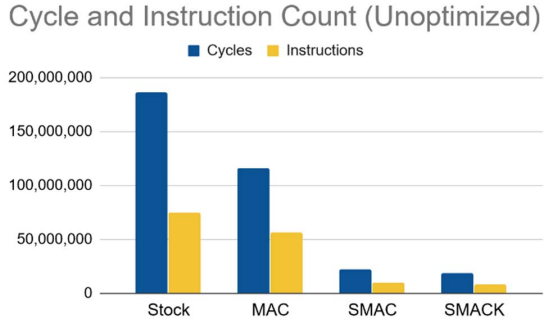

Figure 4 - Performance Results without Compiler Optimization

The optimized and unoptimized share the same overall trend, but optimization reduces the difference between each implementation: With no optimization, SMACK is almost 10x faster than the stock hardware. With optimization enabled, SMACK is only about 2.5x faster. Interestingly, with compiler optimizations enabled, the MAC instruction sees only a 6% improvement over the default hardware.

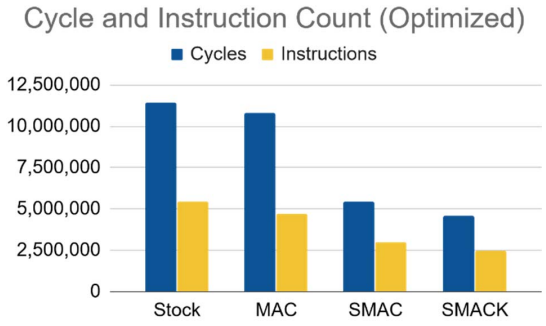

Figure 5 - Performance Results with Compiler Optimization

The third set of results was designed to see if the design of SMACK would perform better on a larger input layer. The results of the test show the opposite effect – SMAC closes the gap and shows almost identical performance to SMACK, 2.5x versus 2.53x. Analysis of these results are still in progress. This is not strictly an apples-to-apples comparison because of the limited system memory: The number of output layers had to be reduced to compensate for the larger input layers. More tests should be performed before any conclusions should be drawn between the usefulness of SMACK vs SMAC. Either of these, however, do show a noticeable improvement over the default.

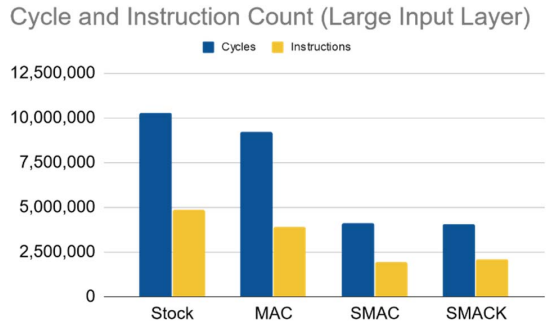

Figure 6 - Performance Results with Larger Input Layers

VIII. Discussion

Some opportunities for further improvements exist, as described below:

A. Reading from Memory

Many of the common issues related to misaligned memory accesses are not present in embedded systems, where no OS exists to partition memory or set permissions, etc. Further research should be investigated with regards to performing misaligned reads. Even if performing a misaligned 32-bit read is twice as slow as performing an aligned read, it would still be twice as fast as loading three 8-bit values and concatenating them into a single register. In addition, the difference in memory architecture between

Table 1 - Table of Performance Results with Compiler Optimization

Implementation	Measured				Derived			
	Fmax (MHz)	Cycles	Instructions	LE Usage	CPI	Speedup	Fmax Ratio	Speedup at Fmax
Stock	111.51	11,459,027	5,447,706	2122	2.10	1	1	1
MAC	103.63	10,800,093	4,691,479	2347	2.30	1.06	0.93	0.99
SMAC	110.34	5,451,865	2,965,200	2444	1.84	2.10	0.99	2.08
SMACK	105.17	4,608,075	2,494,601	2574	1.85	2.49	0.94	2.35

FPGA Block-RAM and regular silicon DRAM may lead to performance improvements in one and not the other. This should be investigated. Note that if a soft-core processor does not support misaligned memory accesses, then an additional benchmark would need to be created without misaligned accesses, or the memory interface would need to be extended. The benefits and drawbacks of these factors need to be considered.

B. Load Straight into Special Register File

The current implementations must load from memory, and then store the contents of a register into the special-purpose register file. These two instructions could be combined to load values from memory and store them straight into the special register file.

C. Investigate the Critical Path of SMACK

The SMACK instruction has a much larger impact on the maximum frequency of the core when compared to SMAC, which reduces it by only 1 percentage point. Given the similarity of the two instructions, this points to the special register file as the likely bottleneck. It is possible that it could be redesigned to reduce this impact by analyzing the performance of the current system.

D. Image-shifting SMACK (ISMACK)

As alluded to in previous sections, loading the image is a major time sink, as without misaligned memory accesses, it should be loaded byte-by-byte. This could be improved by adding another special-purpose storage for the image, as SMACK used for the kernel, because some of these values are re-used in future calculations (depending on the stride of the window). For a stride of 1, two thirds of the values are re-used in the next calculation, and even with a stride of 2, one third is re-used. This register file would act like a large shift register, where each load instruction introduces a column of values and moves the current contents along one column, discarding the last.

When the window completes a row and moves to the left-most column of the next row, however, no values are re-used. This could be mitigated by exploring more complex window movement patterns – instead of the window returning to the left-most column of the next row, the window could instead move down by one and then begin moving left instead. This would further complicate the architecture of this 'shift register file', though, as now values can either be shifted down rows or across columns.

IX. Conclusions

Over the course of this work, RISC-V was identified as a suitable platform for the development and extension of application-specific processors – due in part to the modularity of the ISA, and also due to the opcode space that is reserved for custom instructions.

In the quest to create low-overhead, in-pipeline accelerators for CNN inference, we extended the RISC-V ISA, which resulted in a performance improvement of 2.1x with only a 1% drop in maximum core frequency, or 2.5x with a 6% drop. More work should be done on the analysis of the generated hardware and written assembly code to see if further refinements can be made.

References

[1] U. Shah and A. Harpale, "A Review of Deep Learning Models for Computer Vision," in *2018 IEEE Punecon*, 2018.

[2] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak and L. Benini, "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* pp. 2700-2713, 2017.

[3] S. Steffl and S. Reda, "LACore: A Supercomputing-Like Linear Algebra Accelerator for SoC-Based Designs," in *2017 IEEE International Conference on Computer Design (ICCD)*, 2017.

[4] D. Li, H. Gong and Y. Chang, "Implementing RISCV System-on-Chip for Acceleration of Convolution Operation and Activation Function Based on FPGA," in *2018 14th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, 2018.

[5] M. S. Louis, Z. Azad, L. Delshadtehrani, S. Gupta, P. Warden, V. J. Reddi and A. Joshi, "Towards Deep Learning using TensorFlow Lite on RISC-V," in *Third Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2019.

[6] RISC-V Foundation, "Spike, a RISC-V ISA Simulator," Github, [Online]. Available: https://github.com/riscv/riscv-isa-sim.

[7] S.-H. Tsang, "Review: VGGNet — 1st Runner-Up (Image Classification), Winner (Localization) in ILSVRC 2014," 22 August 2018. [Online]. Available: https://medium.com/coinmonks/paper-review-of-vggnet-1st-runner-up-of-ilsvlc-2014-image-classification-d02355543a11.

[8] "TensorFlow Lite Quantisation Specification," TensorFlow, [Online]. Available: https://www.tensorflow.org/lite/performance/quantization_spec. [Accessed 2019].

[9] RISC-V Foundation, "RISC-V Cores and SoC Overview," [Online]. Available: https://riscv.org/risc-v-cores/. [Accessed June 2019].

[10] "VexRiscv Core," 2017. [Online]. Available: https://github.com/SpinalHDL/VexRiscv. [Accessed July 2019].

[11] Y. LeCun and C. Cortes, "MNIST Handwritten Digit Database," 2010.