RISC-V CPU Optimization For Machine Learning Applications

Kurt Meister
Department of Electrical & Computer Engineering
The University of Arizona
Tucson, AZ 85719
kmeister@email.arizona.edu

David Tondre
Department of Electrical & Computer Engineering
The University of Arizona
Tucson, AZ 85719
dat1@email.arizona.edu

Thao Vo
Department of Electrical & Computer Engineering
The University of Arizona
Tucson, AZ 85719
thaovo@email.arizona.edu

Benjamin Wichmann
Department of Electrical & Computer Engineering
The University of Arizona
Tucson, AZ 85719
wichmannb@email.arizona.edu

Abstract—

With the use of Machine learning growing in nearly all fields of consumer and business computing devices, there is a growing desire to push these capabilities out of the datacenter and to edge computing devices. Such devices are often small form factors that leverage low cost CPUs with RISC Architectures. Given the opportunity to leverage a configurable RISC-V core, this paper seeks to demonstrate a series of straightforward experiments which can leverage the configurability of open source CPU design to optimize a CPU core to execute an ANN classifier. This paper demonstrates simple configuration modifications which result in an approximation of 20% runtime performance improvement in a Neural Network Benchmark application.

Keywords—Deep Learning, RISC-V ISA, Gem5, Tensor Flow, ANN, MLP

I. Introduction

Machine Learning (ML) is a fast growing field used in many industries to optimize and scale their infrastructure. It plays a critical role in extracting meaningful information out of large data sets that are too complex for humans. Algorithms known as Deep Neural Networks use sophisticated mathematical modeling to process data in complex ways, and have revolutionized the concept of ML.

RISC architectures are typically reserved for lower power computing in smartphones and IoT devices. Historically, there has not been enough computing power to enable ML on the device itself, so most of the ML algorithms are pushed to the cloud. By optimizing a lower power embedded SoC computing device to execute ML, the work can further the goal of allowing edge computing devices to execute without needing to send data back to a data center to be processed.

The goal of this project will be to benchmark and optimize the Basic In-Order and Out of Order CPU models provided in the GEM 5 Simulator for use in ML applications. Specifically, image labeling using a multilayer perceptron (Deep Neural Network). This work will offer improvements in latency and improved privacy.

The team hopes to accomplish a meaningful improvement in the performance of the execution time of a fully trained perceptron in labeling a test dataset. To achieve this, the team will be developing a TensorFlow lite based benchmarking application, analyzing the performance of both the In-Order and Out-Of-Order RISC-V implementations, and finally proposing and implementing optimizations to improve performance for the benchmarking application.

II. Related Work

In "*Machine Learning at the Network Edge: A Survey,*" M.G. Sarwar Murshed et al. [1] discusses a variety of techniques for bringing ML out of the datacenter, and to the network's edge. Algorithms which are suitable for lower power computing devices such as K-NN, SVN, and decision trees were discussed. The challenges associated with bringing neural networks to lower power computing devices, and some algorithmic solutions to the problem were addressed. One technique in particular stuck out: the technique of transfer learning, where a model can be trained (the more computationally intensive task) on a more powerful computing device, but deployed to a lower power device. The team hopes to leverage this technique in developing the benchmark application, as it will save considerable time over trying to train an ML algorithm on a simulated SoC. The survey did not, however, go into detail on optimizing the underlying hardware for the application. That is where the team will be taking a novel approach.

In "*Towards Deep Learning using TensorFlow Lite on RISC-V*" [4] Louis, Marcia Sahaya, et al seeks to optimize the neural network's execution on RISC-V with ISA extensions. This specifically involved integrating RISC-V Vector ISA convolution and matrix multiplication instructions to particular applications. Integrating their optimization into TensorFlow Lite source code, saw a significant 8X reduction in the executed instruction count compared to the baseline implementation. By introducing their subset of RISC-V Vector instructions, they saw a significant improvement for a large range of ML applications. The team's approach differs from others since it seeks to improve application performance without altering the ISA.

Likewise, in "Improving the speed of neural networks on CPUs" Vanhoucke et al. [2] explores the optimization of machine learning applications from the perspective of optimizing the Neural Network itself through the use of existing ISA extensions offered by modern x86 CPUs. Like the approach outlined in this paper, they begin with a naive baseline neural network, though focused on speech interpretation. Unlike this team's approach though, it is the benchmark application itself that is optimized.

III. Methodology

The team's main goal is to optimize the baseline Out-of-Order CPU core "DerivO3CPU" provided by the GEM5 CPU Simulator, leveraging the large variety of customization options the simulation environment makes available. The process of doing so involves 3 steps. The first is to establish a performance baseline. To do that, the team developed a Neural Network application. The second involves assessing the baseline performance of the application on the chosen CPU, looking at the breakdown of instructions executed, and how the CPU might be altered to improve the execution of the application. Third, and finally, upon assessing the individual impact of a number of options on application performance, the team created a final configuration to assess the combined improvements. Steps 2 and 3 are discussed in more detail in the experimental results below. For this context, the team will be using IPC as the primary performance metric.

A. Benchmark Development

In order to attempt to optimize a CPU design for a given purpose, it's necessary to establish a benchmark that can be representative of the desired application. In the case of this project, it's necessary for the benchmark application to be able to operate on an embedded system without any of the niceties of an operating system. Fundamentally, most "Deep Learning" or Artificial Neural Networks (ANN's) are at heart Multilayer Perceptrons or MLPs. For this reason, the team chose to implement a simple MLP as the benchmark.

For the purposes of this experiment, the team created an MLP with 3 hidden layers using the TensorFlow library in Python and trained it on a synthetic dataset. A test batch of synthetic input data, as well as the trained weights are then written to a series of C-arrays which can be re-used in the benchmark application. The benchmark application itself re-implements the functionality of the perceptron in C++. The benchmark application works by looping over the test dataset and predicting a class for each input.

The inputs and perceptron are sized to represent a neural network performing labeling on a 24x24 black and white image to reflect the overall project goal of optimizing a RISC-V CPU core for machine learning based inference in edge computing applications. For this benchmark application, attempts were made to balance the execution time constraints of the simulations environment, and the desire to create a benchmark that would have no dependency on disk I/O, and no run-time dependencies against the desire to realistically encompass the behavior of an image classifying Artificial Neural Network. This balancing act resulted in the MLP architecture consisting of 3 layers containing roughly 2000 neurons.

The C++ version of the benchmark has been validated against the same Multi-layer perceptron running in the original TensorFlow environment and demonstrated to produce the same outputs given the same inputs.

The team used an Elastic Compute Cloud (EC2) instance on Amazon Web Services to implement the system above. The system was configured with the GEM5 simulation software, the RISC-V Toolchain, and other necessary dependencies to perform on tests. Appendix A. contains further installation instructions and setup.

B. Training the Neural Net

A script called "GenerateModel.py"[11] can generate an MLP with 3 hidden layers along with a few configurable hyperparameters. This will train the MLP on a synthetic dataset, then generate a header file containing a set of C arrays representing the weights of the trained model along with a test dataset.

Inside the benchmark folder, there is a C++ class layer which can use the generated header file to run predictions on the test data set, compiled without the "DEBUG" declaration to create a version with no outputs to stderr/stdout for embedded applications.

IV. Experimental Results

The initial benchmarks are run with default configurations. The adjustable variables includes: l1i_size (L1 cache size for instruction), l1i_assoc (L1 instruction associativity), l1d_size (L1 cache size for data), l1d_assoc (L1 data associativity), cacheline_size, cpu-clock, maxinsts (max instructions), and with L2 cache.

For the results, the team records results in the following fields: instruction count, # of cycles simulated, IPC, % load, % store, % branch, % int, % fp, %ALU, %mem, iCache Miss Rate, dCache Miss Rate, iCache AMAT, dCache AMAT, sim seconds, L2 Cache Miss Rate, Total Mem Ref, ALU.

Table 1. Default CPU Parameters

CPU Parameter	MinorCPU	DerivO3CPU
L1 iCache size (kB)	32	32
L1 dCache size (kB)	32	32
L1 iAssociativity	4	4
L1 dAssociativity	4	4
Cache Line Size	64	64
L2 Cache	No	No
L2 Cache Size (kB)	N/A	N/A
L2 Associativity	N/A	N/A
ROB Size	N/A	32
Load/Store Queue Size	N/A	32

A. Baseline Results

To establish the baseline performance of the initial benchmark, the team ran the benchmark through 100000000 instructions. Producing the following results.

Table 2
Baseline CPU Performance Metrics

	Minor CPU	DerivO3CPU
IPC	0.599059	1.641667
iCache Miss Rate	0.0058%	0.0092%
dCache Miss Rate	0.2742%	0.6237%
iCache AMAT	1.00464	1.00736
dCache AMAT	1.21936	1.49896
Runtime (sec)	0.10433	0.043014

The instruction content of the first 100000000 instructions was evaluated. The hypothesis was that this combined with the baseline performance would provide a good indication of what configuration options would likely yield the best result. The Gem5 CPU models break instructions down into operation classes which are simulated in the CPU core. Below is the distribution of the instructions within the Operation classes. Notice that the highest proportion of instructions are related to memory accesses. Given that fact, the best opportunity for optimization would be to attempt to optimize the cache configuration. In the following sections we'll discuss the options explored, and their overall impact on the benchmark runtime.

Table 3
Breakdown of CPU Operations

Operation Class	Count of Instructions	Fraction of Total Instructions
No_OpClass	10	0.00%
IntAlu	37507149	37.51%
IntMult	3345	0.00%
IntDiv	3	0.00%
FloatAdd	2494424	2.49%
FloatCmp	3316	0.00%
FloatCvt	3628	0.00%
FloatMult	2494451	2.49%
FloatMultAcc	260	0.00%
FloatDiv	26	0.00%
FloatMisc	104	0.00%
MemRead	42492361	42.49%
MemWrite	5013061	5.01%
FloatMemRead	7490116	7.49%
FloatMemWrite	2497753	2.50%
IprAccess	0	0.00%
InstPrefetch	0	0.00%
total	100000007	

B. Impact of L1 Cache Size on Performance

Below, the impact of varying the L1 Cache size on overall performance can be seen. Of note is the inflection point beyond which increasing L1 cache size actually reduces IPC. Likely owing to the increasing time required to index the cache vs the benefit of additional close cached items.

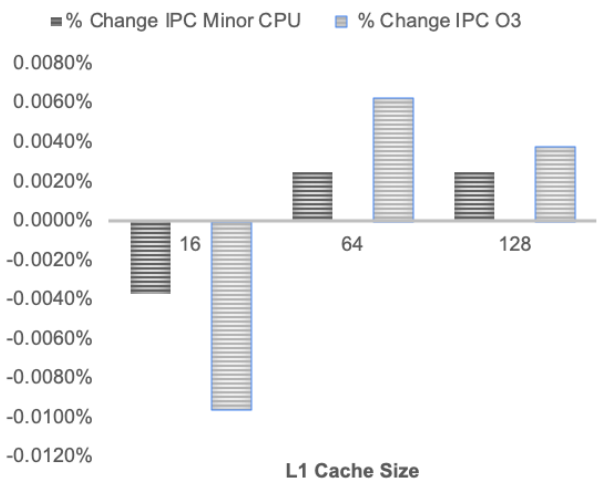
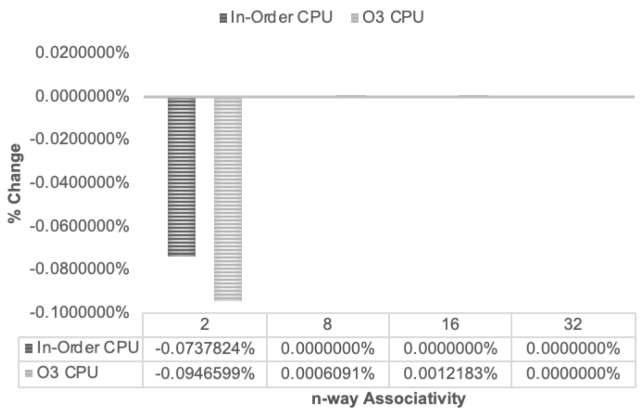


Fig 1. Impact of L1 Cache size on IPC

C . Associativity in L1 Cache

This result was somewhat surprising in that less than 4-way associativity had a very significantly larger performance impact than increasing the associativity had. Likely, the reduced associativity increased the number of ejections from the cache, while the increase did very little to reduce the number of ejections owing to some locality characteristic of the data being used as an input to the neural network.



	2	8	16	32
In-Order CPU	-0.0737824%	0.0000000%	0.0000000%	0.0000000%
O3 CPU	-0.0946599%	0.0006091%	0.0012183%	0.0000000%

Fig 2. Impact of L1 Cache Associativity on IPC

D. Impact of adding an L2 Cache on CPU Performance

Initial baseline expectation was for an improvement in performance from the addition of an L2 cache, however as seen below in Figure 3, that was not the case. Adding an L2 cache was uniformly bad for the performance of both CPU cores. This suggests that the L1 cache is sufficient for the chosen benchmark application, and therefore, the L2 cache is unlikely to contain anything of use that the L1 cache does not already have, leading to longer overall memory access times.

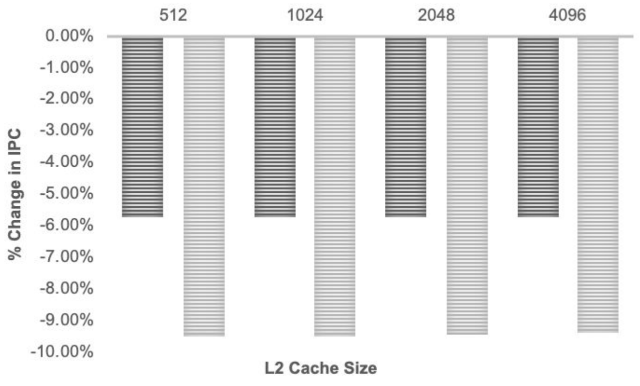


Fig 3. Impact of L2 Cache size on IPC

E. Advanced customization of the DerivO3 Core

The out of order core offered several additional avenues of customization not available on the MinorCPU owing to the fact that it contains a number of hardware elements with configurable parameters that enable it to execute instructions out of order. These include configurable Load/Store Queue sizes (Collectively LSQ), configurable Reorder Buffer (ROB) depth, and a configurable number of types of Functional Units (FU).

Initially, modifying the CPU functional unit allocation was of particular interest due to the high proportion of intALU operations (see Table 3) however, further investigation revealed that the configuration of the DerivO3 CPU already contains 4 IntALU Functional units, and adding additional units had no noticeable impact on the Application runtime. One change however did have a very small impact on the overall performance. Moving from 4 combined Load/Store units to independent Load/Store units did have the impact of slightly increasing (~0.0006%) IPC. This is most likely due to the vagaries of the simulation environment, and not an actual improvement worth pursuing.

However, modifying the LSQ did show some capacity to impact CPU performance in a meaningful way.

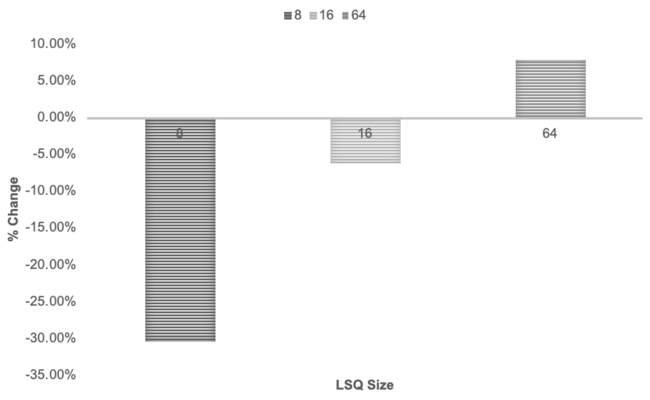


Fig 4. Impact of LSQ Size on IPC

Other modifications experimented with included modifying the ROB, and the Commit Width, however those values are maximized in the default configuration for the DerivO3CPU, and while the team could degrade performance by using them. It would not be possible to improve the performance of the benchmark without significant modification to the underlying CPU design.

F. Final Selected Configurations:

The configuration below represents the best modifications the team could identify for improving the performance of the Machine learning benchmark for the application.

Table 4
Default CPU Parameters

CPU Parameter	MinorCPU	DerivO3CPU
L1 iCache size (kB)	64	64
L1 dCache size (kB)	64	64
L1 iAssociativity	4	16
L1 dAssociativity	4	16
Cache Line Size	64	64
L2 Cache	No	No
L2 Cache Size (kB)	N/A	N/A
L2 Associativity	N/A	N/A
ROB Size	N/A	32
Load/Store Queue Size	N/A	64

Table 4
Final CPU Performance Metrics

	Minor CPU	DerivO3CPU
IPC	0.599074	1.768934
iCache Miss Rate	0.0057%	0.0091%
dCache Miss Rate	0.2742%	0.0127%
iCache AMAT	1.00456	1.00728
dCache AMAT	1.21936	2.0132
Runtime (sec)	0.104328	0.035332

V. Conclusion

Overall, tuning of high level CPU configuration parameters, while not altogether ineffective for impacting application performance is not in and of itself sufficient to achieve a meaningful increase in the runtime of the chosen benchmark application. This team was however able to achieve an ~20% reduction in runtime on the DerivO3CPU relative to initial baseline configuration.

A. Future Work

Future work in the area of CPU optimization for machine learning applications should focus on lower level optimizations along the lines of instruction specific functional units, ISA extensions. Specifically in the case of the cores evaluated in this paper, addition of SIMD instruction sets would be of great benefit to machine learning applications where highly vectorizable operations make up the lion's share of the work.

Acknowledgment

The team would like to thank Dr. Tosiron Adegbija for his guidance throughout the semester.

References

[1] Murshed, Murphy, et al. "Machine Learning at the Network Edge: A Survey" arXiv:1908.00080v2 [CS.LG] , Jul 2019

[2] Vanhoucke, Vincent, et all. "Improving the speed of neural networks on CPUs". Google, 2011.

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

[4] Louis, Marcia Sahaya, et al. "Towards Deep Learning using TensorFlow Lite on RISC-V." *Proc. ACM CARRV* (2019).

[5] Sze, Vivienne, et al. "Hardware for machine learning: Challenges and opportunities." *2017 IEEE Custom Integrated Circuits Conference (CICC)*. IEEE, 2017.

[6] Nöltner-Augustin, M. "RISC-V—Architecture and Interfaces The RocketChip." *COMPUTER ENGINEERING* (2016): 6.

[7] Branco, Sergio, et al. "*Machine Learning in Resource-Scarce Embedded Systems, FPGAs, and End-Devices: A Survey*". *Electronics* (2019):8.

[8] Kotas, Gerald. "*Exploration of GPU Cache Architectures Targeting Machine Learning Applications*". RIT Scholars.

[9] Gem5, "*Gem5 Installation*"."https://github.com/gem5/gem5

[10] RISC-V, "*RISCV-GNU-Toolchain*". https://github.com/riscv/riscv-gnu-toolchain

[11] Meister, Kurt. "*ML Benchmark*". https://github.com/kmeister/ML_Benchmark

[12] Kukunas, Jim (2015). "*Power and Performance: Software Analysis and Optimization*". Morgan Kaufman. p. 37. ISBN 9780128008140.

APPENDIX

A. Environment Set Up and Software Installations

To perform this research, the team used the following system configurations:

- **AWS EC2 Instance:**
  Model: C5.xlarge (vCPU=4, Memory=4Gb, Storage>=30GB)
  OS: Ubuntu 18.04

The following steps were used to perform the software configuration:

1. **Build GEM5 for RISC-V**

   Install GEM5 dependencies:
   ```
   $ sudo apt install build-essential
   $ apt install m4 zlib1g-dev scons python-six python-dev
   ```

   Clone and Build GEM5
   ```
   $ git clone https://gem5.googlesource.com/public/gem5
   $ cd gem5
   $ scons build/RISCV/gem5.opt
   ```

   To test gem5:
   ```
   $ build/RISCV/gem5.opt configs/example/se.py -c tests/test- progs/hello/bin/riscv/linux/hello
   ```

2. **Install RISC-V toolchain**:
   Install RISC-V Toolchain Dependencies:
   ```
   $ sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
   ```

   Clone and Build RISC-V Toolchain
   ```
   $ git clone https://github.com/riscv/riscv-gnu-toolchain
   $ ./configure --prefix=/opt/riscv
   $ PATH=/opt/riscv/bin/:$PATH
   $ make linux
   ```

3. **Clone Custom Code Repository and Build Benchmark:**

   Clone Benchmark repository:
   ```
   $ git clone https://github.com/kmeister/ML_Benchmark.git
   ```

   Build the Benchmark:
   ```
   $ cd ML_Benchmark/benchmarks
   $ riscv64-unknown-linux-gnu-gcc -static -Wall -O0 -I. -c main.cpp -o main.o
   $ riscv64-unknown-linux-gnu-g++ -static -Wall -L. -o mlbench main.o
   ```

   Run Benchmark:
   ```
   $ build/RISCV/gem5.opt configs/example/se.py --cpu-type DerivO3CPU  -c  ../ML_Benchmark/Benchmarks/mlbench
   --caches --l1i_size=32kB --l1i_assoc=4 --l1d_size=32kB --l1d_assoc=4 --cacheline_size=64 --cpu-clock=1.6GHz
   --maxinsts=1000000
   ```

4. **Retrain Neural Network:**
   Retrain the Neural Network
   ```
   $ cd ML_Benchmark/scripts
   $ ./ModelGenerator.py ../Benchmarks/Weights.h
   ```

   (if desired) Use the following to alter the Neural Network Weights:
   To see the options for configuring a model:
   ```
   $ ./ModelGenerator.py -h
   ```

   Generate a weights.h model:
   ```
   $ ./ModelGenerator.py ../Benchmarks/Weights.h --test_size=1000 --n_classes=26 --n_features=576 --layer1=1000
   --layer2=1000 --n_epochs=35
   ```