

RISC-V for AI

Literature Survey and Profiling Analysis

Analysis of CNNs







CNN for MNIST dataset : Neural Network Architecture







Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 24, 24, 32)	832
max_pooling2d (MaxPooling2D)	(None, 12, 12, 32)	0
conv2d_1 (Conv2D)	(None, 8, 8, 64)	51264
max_pooling2d_1 (MaxPooling2D)	(None, 4, 4, 64)	0
flatten (Flatten)	(None, 1024)	0
dense (Dense)	(None, 10)	10250
Total params: 62,346		
Trainable params: 62,346		
Non-trainable params: 0		

Coded the network in C and executed on x86 and riscv platforms and results were verified.

Profiling Analysis of CNN

Done using Valgrind and Kcachegrind tools

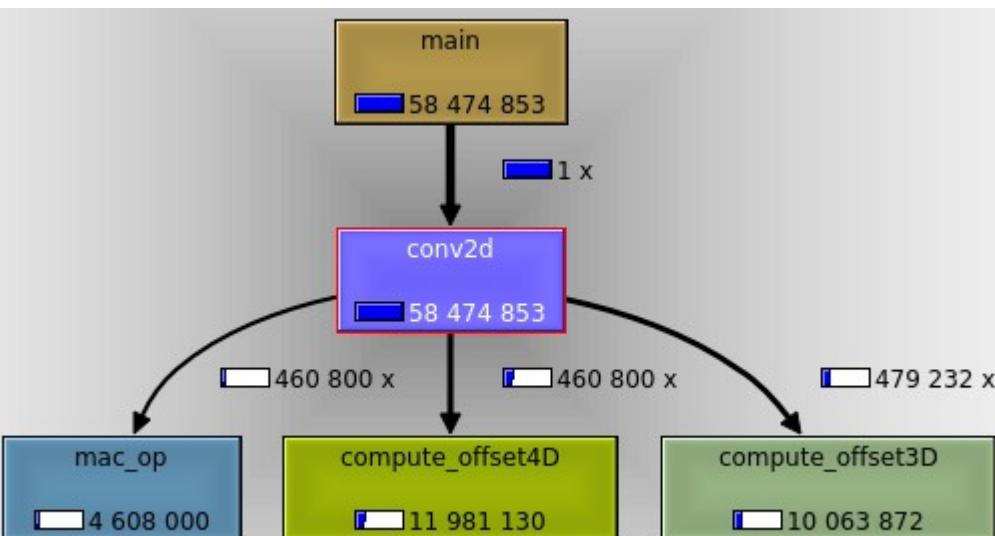
CEst	CEst per call	Count	Callee
 231 297 075	115 648 537	2	 conv2d (x86: convolute.c)
795 323	397 661	2	 maxpool2 (x86: maxpool.c)
401 679	401 679	1	 perceptron (x86: perceptron.c)
192 281	64 093	3	 malloc (libc-2.29.so: malloc.c)
4 410	630	7	 free (libc-2.29.so: malloc.c)

CEst	CEst per call	Count	Callee
 98.47	115 648 537	2	 conv2d (x86: convolute.c)
0.34	397 661	2	 maxpool2 (x86: maxpool.c)
0.17	401 679	1	 perceptron (x86: perceptron.c)
0.08	64 093	3	 malloc (libc-2.29.so: malloc.c)
0.00	630	7	 free (libc-2.29.so: malloc.c)

Convolution -only Profiling

Done using Valgrind and Kcachegrind tools

1. Work-load division



CEst	CEst per call	Count	Callee
20.49		26 460 800	compute_offset4D (x86: util.c)
17.21		21 479 232	compute_offset3D (x86: util.c)
7.88		10 460 800	mac_op (x86: util.c)
0.32		10 18 432	relu (x86: relu.c)
0.01	4 823	1	malloc (libc-2.29.so: malloc.c)

CEst	CEst per call	Count	Callee
11 981 130		26 460 800	compute_offset4D (x86: util.c)
10 063 872		21 479 232	compute_offset3D (x86: util.c)
4 608 000		10 460 800	mac_op (x86: util.c)
189 986		10 18 432	relu (x86: relu.c)
4 823	4 823	1	malloc (libc-2.29.so: malloc.c)

Analysis

1. Computing Array Indices consume considerable amount of time(about 33%)

Formula: $x1*d2*d3*d4 + x2*d3*d4 + x3*d4 + x4$

Idea: For upto 4D arrays, whose dimension along each axis is < 256 , the index computation can be vectorized.

Algorithm for Accelerating the above computation

We store these in 2 registers say, r1,r2 as 8-bit unsigned integers:

t	r1	r2	r3	r4	r5
0	x1,x2, x3,x4	d1,d2, d3,d4			
1			x1*d2, d3*d4		x3*d4
2				r3[31:16] +x2	r5+x4
3				r3+r4	
4					r4+r5

r1	x1	x2	x3	x4
r2	d2	d3	d4	

Using 8 bit Multiplier with
4 clock cycles
(CSA+Wallace Tree),
this operation can be
completed within 10 clocks

High Level Synthesis of Offset Compute

Hardware accepts 2 int arrays , x and d as inputs and returns an integer
It extracts x1,x2,x3 and x4 ; d2,d3,d4 using AND and Shift Ops and passes it to offset_compute

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	6.492	1.25

Latency (clock cycles)

Summary

Latency		Interval		Type
min	max	min	max	
3	3	3	3	none

Register

Name	FF	LUT	Bits	Const Bits
ap_CS_fsm	4	0	4	0
d3_reg_187	8	0	8	0
d4_reg_192	8	0	8	0
tmp1_i_reg_202	24	0	24	0
tmp_8_i_reg_197	16	0	16	0
x2_reg_172	8	0	8	0
x3_reg_177	8	0	8	0
x4_reg_182	8	0	8	0
Total	84	0	84	0

DSP48

Instance	Module	Expression
main_comp_ama_addbkb_U1	main_comp_ama_addbkb	$(i0 + i1) * i2 + i3$
main_comp_mac_mulcud_U2	main_comp_mac_mulcud	$i0 * i1 + i2$

Desirable elements in an low-power RISC processor for ML

1. ISA Extensions for commonly used instructions such as vector/tensor operations
2. Scratch-pad memory for working with intermediate results
3. Resource efficient Floating Point Unit – bfloat16, DLFloat

Brain Floating Point - bfloat16

16-bit FPU alternative to IEEE754 Half Precision FPU

Format : 1(sign) + 8(exp) + 7(mantissa).

Used in Google TPU cloud v3 MAC units. Multiplication is done in bfloat16 while Accumulation is done in FP32.

AMD likely to incorporate

DL Float16- Floating Point format for DL Training and Inference

1. Format : 1(sign) + 6(exp) + 9(mantissa)
2. FMA Hardware : Fused Multiply Add



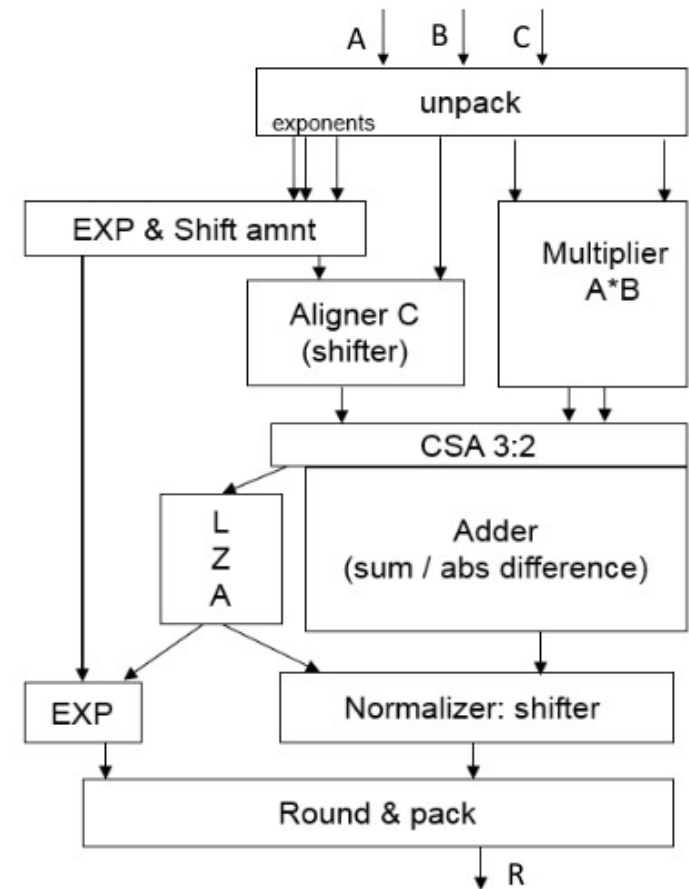
$$X = -1^s * 2^{(e-b)} * (1 + \frac{m}{512})$$

Fused Multiply Add Hardware

Useful for Matrix Multiplications, Convolutions

$$R = C + A*B$$

FMA in bfloat is mixed(bfloat-16 for multiplications and FP-32 for add). However, in DLFloat FMAs, it is fully DLFloat-16



DSP ISA augmentation to RISC-V

Paper on PULP-RISCV core for IoT

Microarchitectural Improvements to RISCV32IM

1. 4-core with shared TCDMs
2. Instruction Pre-fetch buffer + Shared I-Cache
3. ALU Datapath enables operations on 8-bit / 16-bit vectors, supporting unaligned memory accesses.
4. Instructions for Packed-SIMD, Fixed-point, shuffling bytes in a word and hardware loop unrolling, dot-product

Algorithms for Accelerating Activations

CORDIC : To study

Week-3 (Aug 7-14, 2020)

Week 2 Actions

1. Perform profiling of various other ML algorithms and other Neural Network Topologies and find out if there can be any general instructions for speed up.
2. Exploration of Implementing Brain Float 16 FPU to be explored.

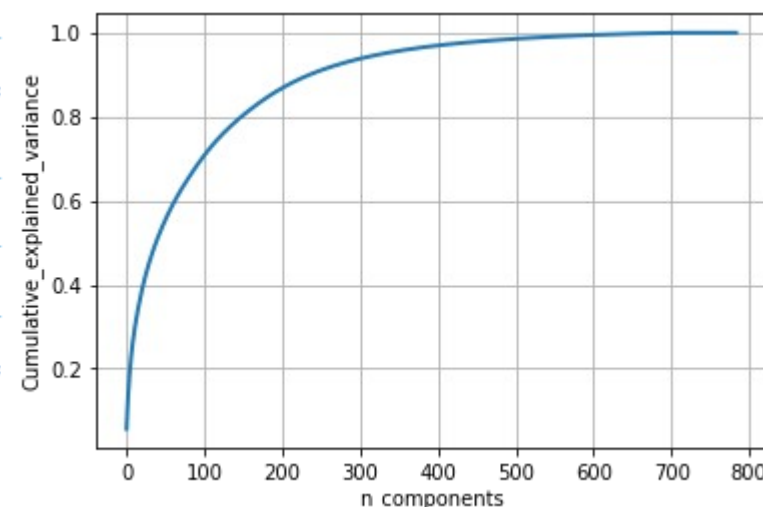
Week -3 :Profiling Other ML Algorithms

1. Unsupervised Learning - Principle Component Analysis (PCA)

Use Case: To reduce the Size of Input Dimensions for classifying MNIST Dataset

A 4-layer fully-connected Neural Network is used for MNIST digit classification where the input dimension is 784.(28x28)

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 32)	25120
dense_1 (Dense)	(None, 16)	528
dense_2 (Dense)	(None, 10)	170
Total params: 25,818		
Trainable params: 25,818		
Non-trainable params: 0		



Since Image matrices are sparse, their variances will be limited to only a subset of 784 dimensions. PCA is used to capture the most significant Eigen Vectors in the Image Dataset (60,000 x 784).

The plot aside shows the fraction of variance of the Image Dataset that can be captured as a function of the number of significant eigen vectors used. So, in order to capture 75% of the variance in the images, only **100 out of 784** dimensions are sufficient.

PCA-Inference

The Neural Network Model for the reduced set of Inputs (100) is shown below:

It can be seen that the new model features only ~4,000 parameters as Compared to ~26000 in the original one

Layer (type)	Output Shape	Param #
dense_3 (Dense)	(None, 32)	3232
dense_4 (Dense)	(None, 16)	528
dense_5 (Dense)	(None, 10)	170
Total params: 3,930		
Trainable params: 3,930		
Non-trainable params: 0		

Computational Overhead

In the inference engine,

1. All inputs have to be normallized to Zero-mean and UNIT standard deviation.
2. These scaled inputs(784 dim) have to be transformed to 100-dimension space using SVD matrix. This results in a matrix - vector multiplication

INPUT_TRANSFORMED = [1 x 784] * [784 x 100] with 78400 multiplications.

The PCA-transformation + 4 layer NN was coded in C language and profiled with Valgrind.

Result : 85% of the processor cycles are used for PCA.(3 442 401 cycles) (115 648 537 cycles for Conv2D)

Followed by 3% load (117 266) cycles for *perceptron*

Followed by 1.25% for 47 136 cycles for normallizing function.

Autoencoder

Studied Auto-encoder

Use Case: Same as PCA

Auto-encoders can be designed using the same structure as used for PCA.

Auto-encoders and PCA differ in the way they are trained.

Thus, computational load is expected to be the same.

RNN and LSTM

Studied Rand Implemented RNN using LSTM on Python.
C coding for LSTM in progress ...

Paper Summary: “Cambricon : AN ISA for Neural Networks” ACM/IEEE-2016

Propose a domain specific Instruction Set Architecture for NN Accelerators

Integrates Scalar, Vector, Matrix, logical, data transfer, and control instructions based on a comprehensive analysis of existing Neural Network Techniques

Architecture Highlights:

1. Accelerator is designed to accept instruction sequentially and run like a processor.
2. *Data-level parallelism* :- Instructions to load Vectors and matrices onto scratch-pad memory from Main Memory
3. *Scratch Pad Memory* :- Uses Scratch Pad memory(High speed) for fast computations instead of the Register Files and Cache Memories.
4. *Vector /Matrix Instructions* :- Dot-product, ADD, SUB, MULT for matrix and vectors
5. Engines for computing vector-wise ($\exp(x)$, $\log(x)$, $\tanh(x)$) using CORDIC algorithm

Evaluation:

Evaluated over ten representative NN techniques:- MLP, CNN, RNN, LSTM, Autoencoder, Sparse Autoencoder, Boltzmann machine, RBM, Self Organized Mapping and Hopfield NN.