

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

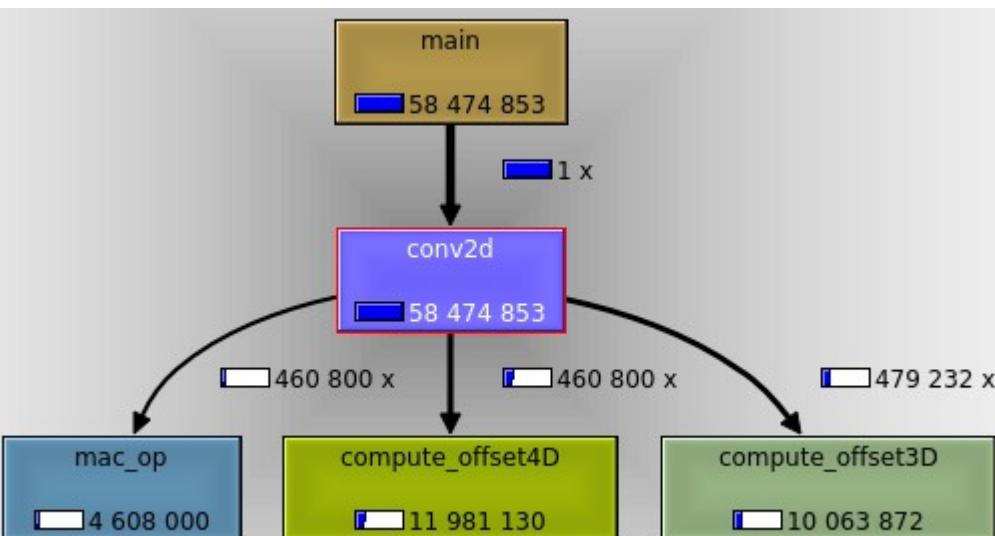
CEst	CEst per call	Count	Callee
 231 297 075	115 648 537	2	 conv2d (x86: convolute.c)
795 323	397 661	2	 maxpool2 (x86: maxpool.c)
401 679	401 679	1	 perceptron (x86: perceptron.c)
192 281	64 093	3	 malloc (libc-2.29.so: malloc.c)
4 410	630	7	 free (libc-2.29.so: malloc.c)

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



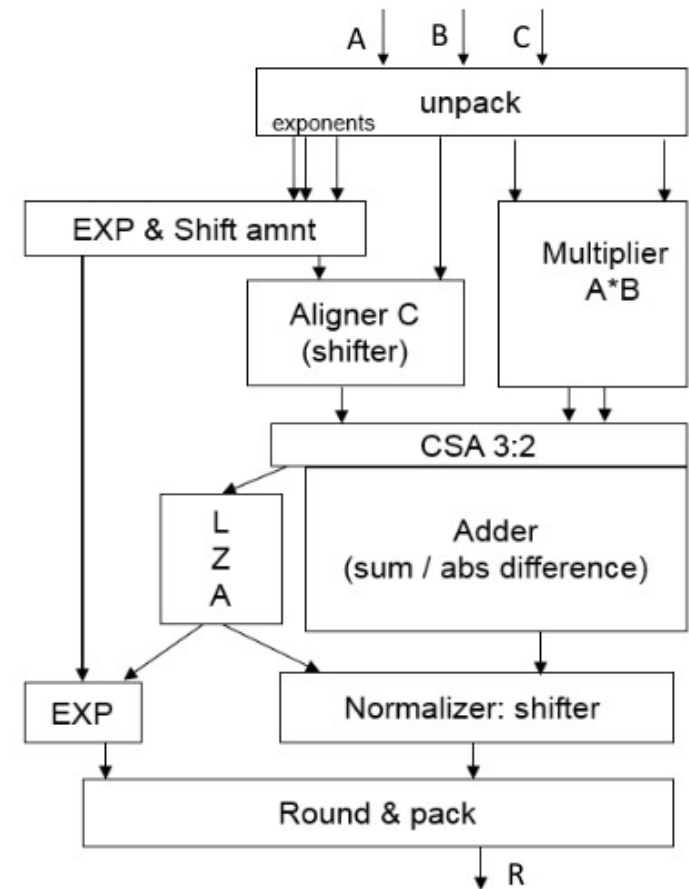
$$X = -1^s * 2^{(e-b)} * (1 + \frac{m}{512})$$

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 speeded up.
2. Exploration of Implementing Brain Float 16 FPU to be explored.

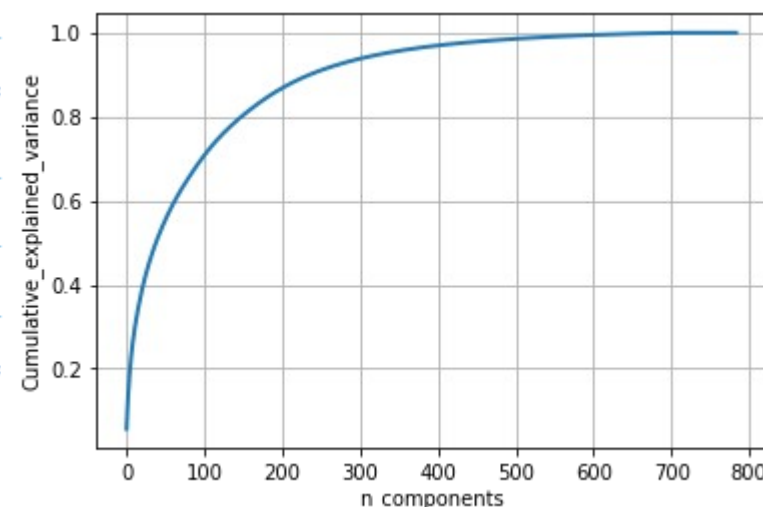
Week -3 :Profiling Other ML Algorithms

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 normalized 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

$INPUT_TRANSFORMED = [1 \times 784] * [784 \times 100]$ 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 normalizing 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.

Week 4:

Week3 Actions:

1. Cambricon is a useful paper. Get statistics on computational load of instructions used in Deep Learning operations from the paper.
2. Continue profiling for other NN topologies.

Week 4:

Realized mistakes in coding styles

1. Use of malloc() to be avoided as it consumes substantial amount of time itself. Instead create all arrays and allocate memory for them during compile time. Tradeoff: Generalization
2. Remove all function calls inside for loops . They add to enormous amounts of push-pop overheads.

Week-4 : Recurrent Neural Network

RNN for classifying Sequential MNIST dataset.

Each image is fed to the RNN one row at a time.

Input Dimensions = 28

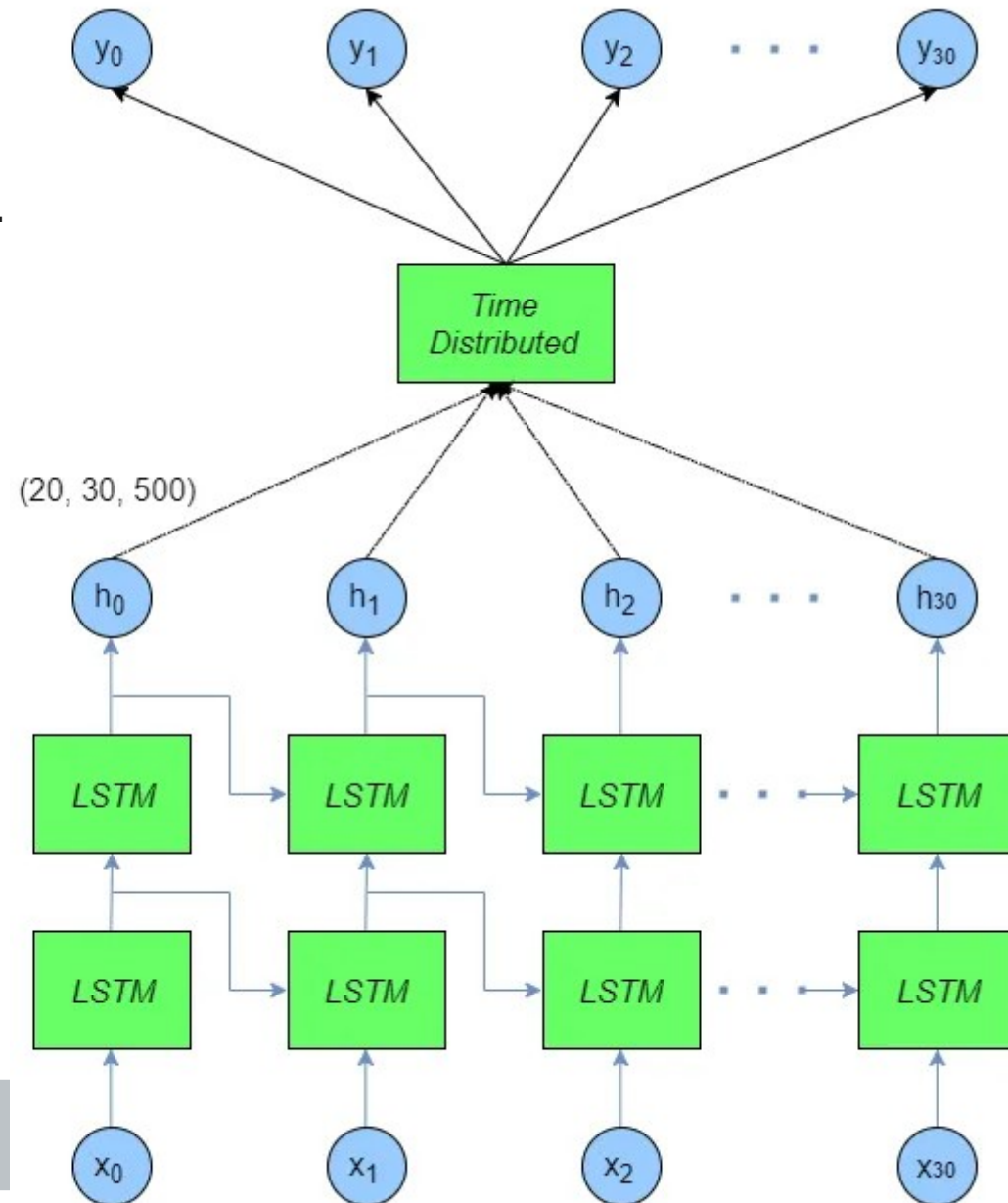
LSTM-1 Layer has 128 neurons

LSTM-2 layer has 128 neurons

The output of the last LSTM layer (128 dimensions) is fed to 3-layer perceptron to obtain class conditional densities 0-9.

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 28, 128)	80384
lstm_1 (LSTM)	(None, 128)	131584
dense (Dense)	(None, 64)	8256
dense_1 (Dense)	(None, 10)	650

Total params: 220,874
Trainable params: 220,874
Non-trainable params: 0



LSTM

$$f_t = \sigma((W_{hf} \times h_{t-1}) + (W_{xf} \times x_t) + b_f)$$

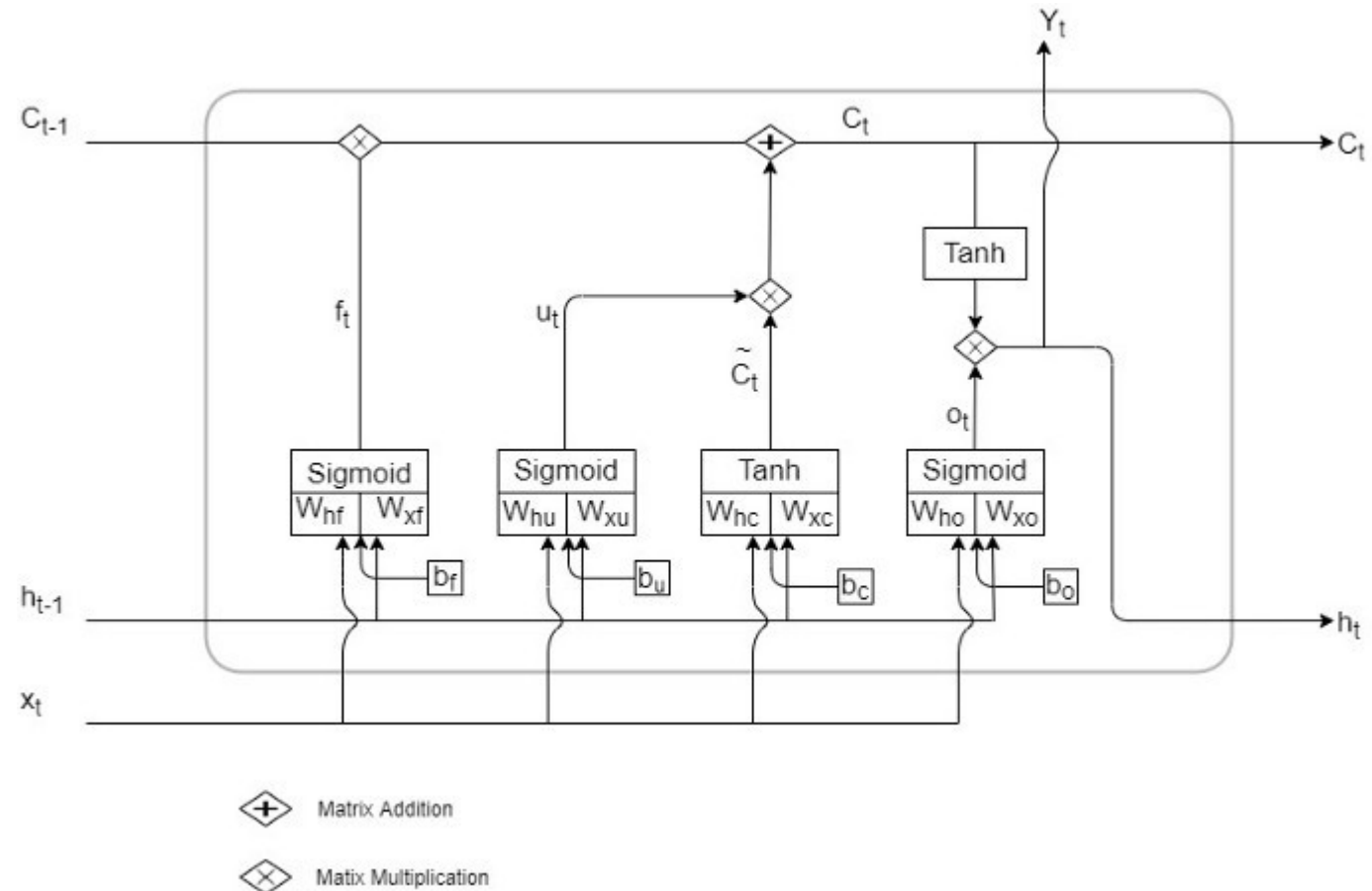
$$u_t = \sigma((W_{hu} \times h_{t-1}) + (W_{xu} \times x_t) + b_u)$$

$$\tilde{c}_t = \tanh((W_{hc} \times h_{t-1}) + (W_{xc} \times x_t) + b_c)$$

$$o_t = \sigma((W_{ho} \times h_{t-1}) + (W_{xo} \times x_t) + b_o)$$

$$c_t = [\tilde{c}_t \times u_t] + [c_{t-1} \times f_t]$$

$$y_t = [o_t \times \tanh c_t]$$



Source: <https://medium.com/analytics-vidhya/demystifying-lstm-weights-and-biases-dimensions-c47dbd39b30a>

Implementation and Profiling Results

C-code compiled and run on x86. For inferring 1 image, the following are the profiling results:-

Profiling Results show that RNN takes a total of **148,051,630** cycles to complete one run, almost thrice as long as CNN (115 648 537)

Mthematical Operation	%Exe. Time Of the total prgm	No. of times called	Avg Execution cycles per call
Matrix Vector Multiplication [1 x 128]* [128 x 128] [1 x 28] * [28 x 128] [1 x 128] * [128 x 64] [1 x 64] * [64 x 10]	96.92%	450	409, 245 251, 335 40, 680
2 vector Addition (128)	1.04%	506	~3,000
exponent(x)	0.72%	35, 904	30

Profiling Matrix Vector Multiply

1. In order to obtain percentage of time a given piece of code took for execution, a function was created. This function was being called from the program. Problem: Since the function is repeatedly called from inner loop, execution overheads such as stack operations were found to each away a significant amount of time. (almost 50% of original function time)
2. Therefore, all functions called within loops were removed; Code was flattened.

Line-wise Assembly Level Code breakup for *Matrix vector Multiply*

Total Assembly Instructions(x86): 65

Innermost loop : 24 instructions

Break up:

Offset Calculation : 6

Load ith element : 7

Load ijth element : 8

Multiply : 3

Accumulate : 3

Ara: RISC-V Vector Processor, Oct 2019, arXiv

Specifications:

64-bit Vector processor based RISC-V's vector extension ISA

Freq of Operation ~ 1.2 GHz

Performance : 33 Dp-GFLOPS (41 DP-GFLOPS/Watt)

Power : 794 mW (max.)

Main Idea: Exploits regularity in Parallel Data Workloads

Vector Instruction: A single vector instruction can be used to express a data-parallel computation on a very large vector => Reduces Fetch, Decode , Store Overhead significantly.

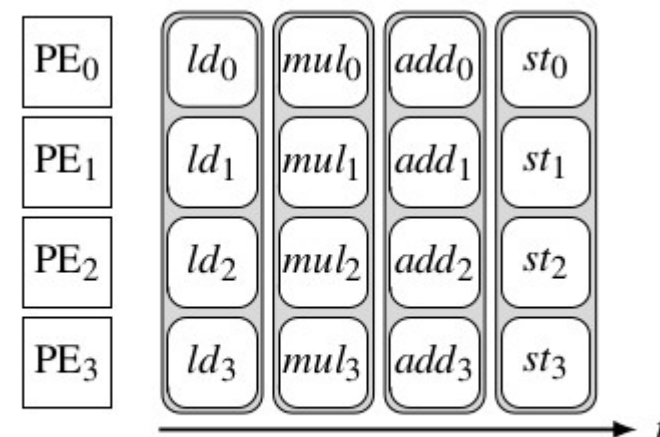
Advantage over GPUs

In Single Instruction- Multiple thread(SIMT) architectures, the fetch.decode,store overhead amortization is limited to the number of execution elements in a processing block (32, 64). Whereas, in Vector Processors, there is no cap on Vector length.

Architectures to handle Vector data

1. **Array Processors:** Implement Packed-SIMD architecture

- Several independent processing elements having a
- Common control unit
- Limitation : Vector length is fixed.
- Ex: Intel Streaming SIMD Extension (SSEs), ARM Neon, RISC-V DSP extension



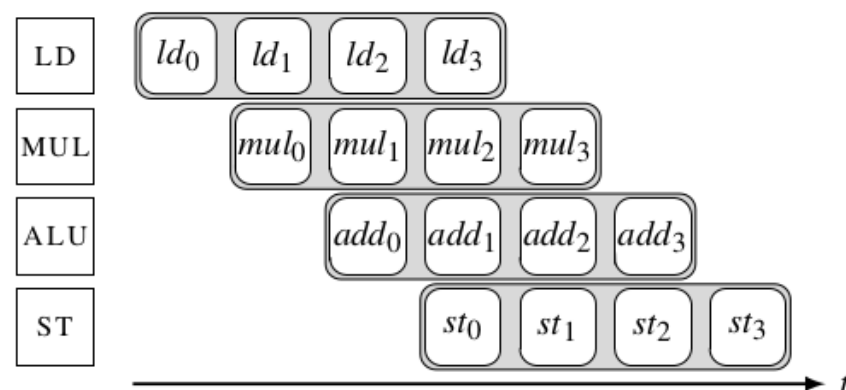
2. **Vector Processors**

Time-multiplexed version of Array Processors, implementing vector-SIMD instructions

Several specialized functional units stream the micro-operations on consecutive cycles, as shown in figure. By doing so, the number of functional units no longer constrains the vector length.

Ex: Cray SuperComputer, ARM Scalable Vector Extension, Fujitsu A64FX

Execution pattern on an array processor [20].



2. Execution pattern on a vector processor [20].

Microarchitecture

Ara works in tandem with Ariane, an open source Linux Capable application-class core

Ariane has been extended to drive the accompanying vector unit as a tightly coupled co-processor

Ariane:

Six-Stage Pipeline

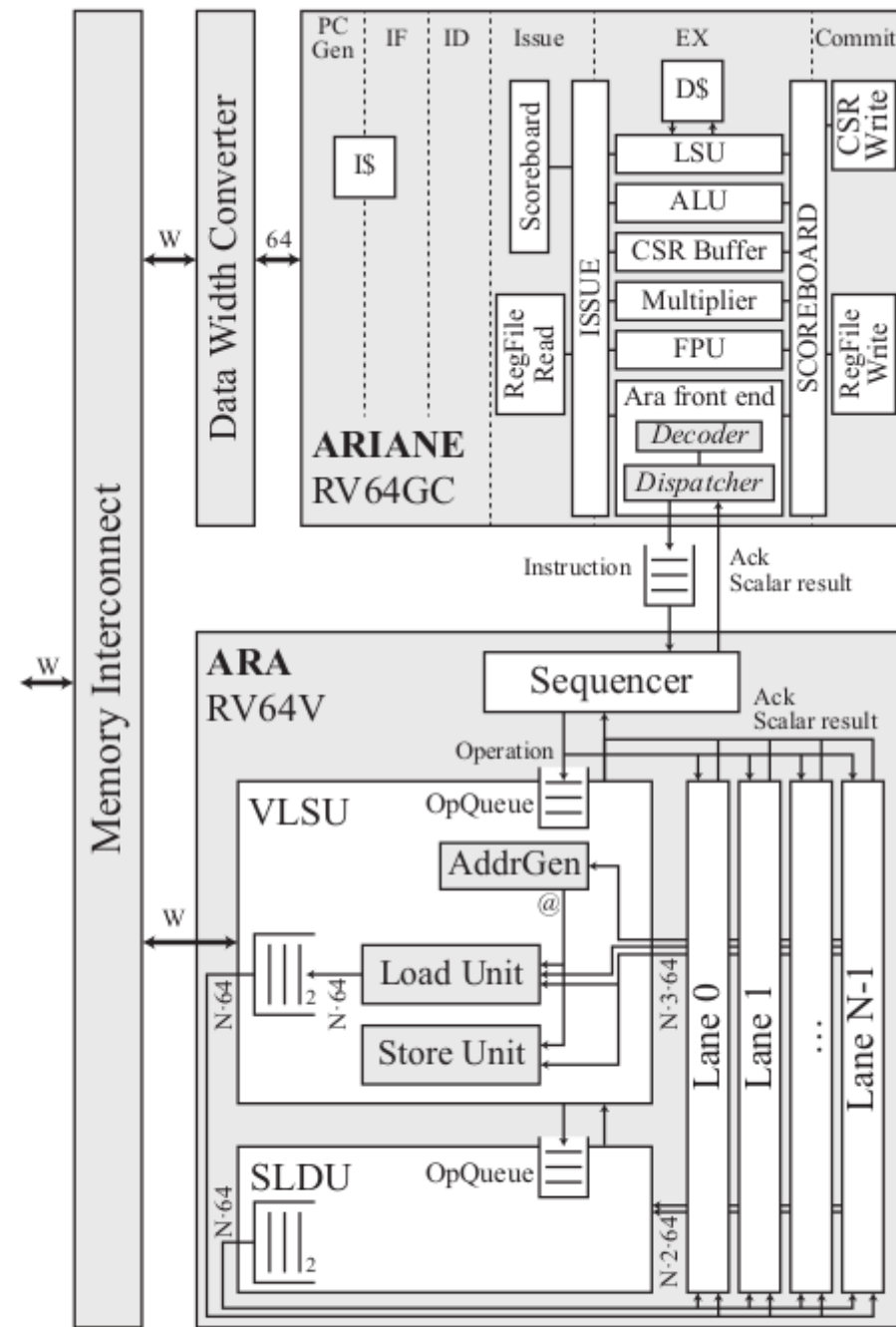
In-order

All vector instructions are decoded in Ara.

Handshaking mechanism between Ara and Ariane

Consists of Several lanes.

Each lane can handle Upto 8 instructions in Parallel



(a) Block diagram of an Ara instance with N parallel lanes. Ara receives its commands from Ariane, a RV64GC scalar core. The vector unit has a main sequencer; N parallel lanes; a Slide Unit (SLDU); and a Vector Load/Store Unit (VLSU). The memory interface is W bit wide.

Results

Benchmarking:

$Y \leftarrow aX + Y \Rightarrow$ Memory Intensive

Convolution \Rightarrow Compute Intensive

Matrix Multiplication : Moderately
Computationally Intensive

Performance figures:

No comparison done with a non-vector
processor

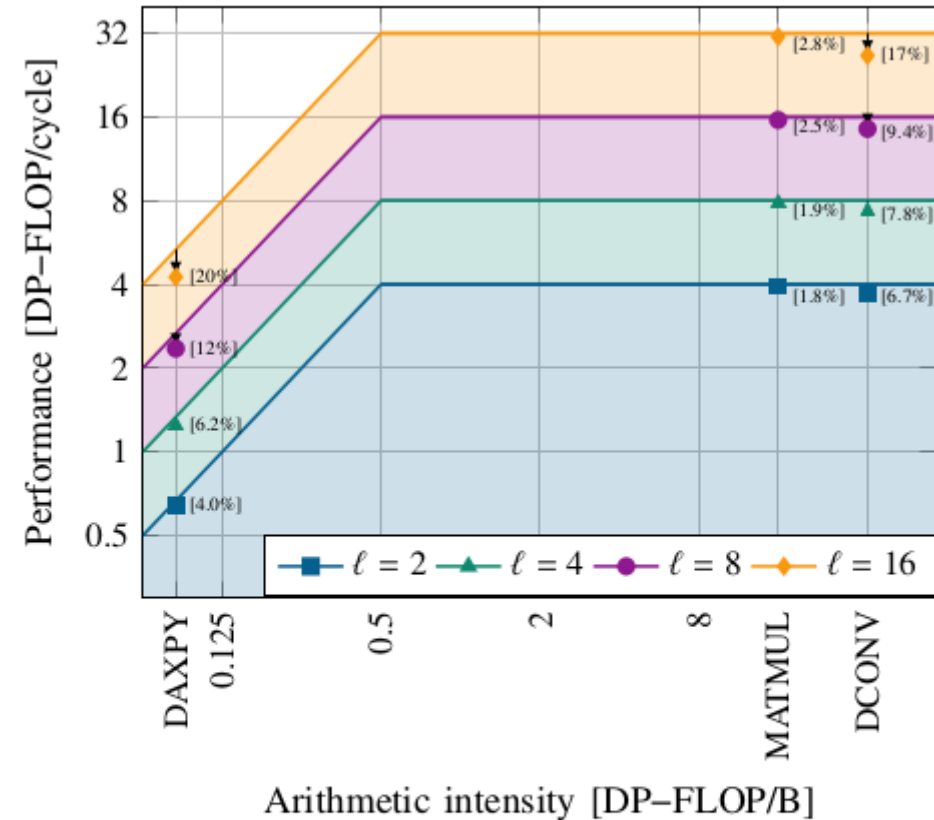


Fig. 6. Performance results for the three considered benchmarks, with different number of lanes ℓ . AXPY uses vectors of length 256, the MATMUL is between matrices of size 256×256 , and CONV uses GoogLeNet's sizes. The numbers between brackets indicate the performance loss, with respect to the theoretically achievable peak performance.

Performance with Different Lane Widths

TABLE III
POST-PLACE-AND-ROUTE ARCHITECTURAL COMPARISON BETWEEN SEVERAL ARA INSTANCES IN GLOBALFOUNDRIES 22FDX FD-SOI TECHNOLOGY IN TERMS OF PERFORMANCE, POWER CONSUMPTION, AND ENERGY EFFICIENCY.

Figure of merit	Instance											
	$\ell = 2$			$\ell = 4$			$\ell = 8$			$\ell = 16$		
Clock (nominal) [GHz]	1.25			1.25			1.17			1.04		
Clock (worst-case) [GHz]	0.92			0.93			0.87			0.78		
Area [kGE]	2228			3434			5902			10735		
<i>Area per lane</i> [kGE]	1114			858			738			671		
Kernel	<i>matmul</i> ^a	<i>dconv</i> ^b	<i>daxpy</i> ^c	<i>matmul</i>	<i>dconv</i>	<i>daxpy</i>	<i>matmul</i>	<i>dconv</i>	<i>daxpy</i>	<i>matmul</i>	<i>dconv</i>	<i>daxpy</i>
Performance [DP–GFLOPS]	4.91	4.66	0.82	9.80	9.22	1.56	18.2	16.9	2.80	32.4	27.7	4.44
Core power [mW]	138	130	68.2	259	239	113	456	420	183	794	676	280
Leakage [mW]	7.2			11.2			21.1			31.4		
Ariane/Ara [mW]	22/116	22/108	20/48	27/232	29/210	25/88	28/428	29/391	24/159	31/763	31/646	25/255
<i>Core power per lane</i> [mW]	69	65	34	65	60	28	57	54	23	50	42	15
Efficiency [DP–GFLOPS/W]	35.6	35.8	12.0	37.8	38.6	13.8	39.9	40.2	15.3	40.8	41.0	15.9

^aDouble precision floating point 256×256 matrix multiplication. ^bDouble precision floating point tensor convolution with sizes from the first layer of GoogLeNet. Input size is $3 \times 112 \times 112$ and kernel size is $64 \times 3 \times 7 \times 7$. ^cDouble precision AXPY of vectors with length 256.

Vector Architectures contd ...

3. SIMT : Single instruction to multiple independent threads running in parallel

Ex: NVIDIA Volta GPU

While this model is efficient to handle data intensive workloads, it consumes substantial power

4. Vector Thread: A compromise between SIMD and MIMD.

Supports Loops with cross iteration dependencies

Main difference between SIMT and VT is that in the latter the vector instructions reside in another thread, and scalar book-keeping instructions can potentially run concurrently with scalar ones.

This division alleviates the problem of SIMT threads running redundant scalar instructions that must be later coalesced in hardware

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.

Microarchitecture

Table I. An overview to Cambricon instructions.

Instruction Type		Examples	Operands
Control		jump, conditional branch	register (scalar value), immediate
Data Transfer	Matrix	matrix load/store/move	register (matrix address/size, scalar value), immediate
	Vector	vector load/store/move	register (vector address/size, scalar value), immediate
	Scalar	scalar load/store/move	register (scalar value), immediate
Computational	Matrix	matrix multiply vector, vector multiply matrix, matrix multiply scalar, outer product, matrix add matrix, matrix subtract matrix	register (matrix/vector address/size, scalar value)
	Vector	vector elementary arithmetics (add, subtract, multiply, divide), vector transcendental functions (exponential, logarithmic), dot product, random vector generator, maximum/minimum of a vector	register (vector address/size, scalar value)
	Scalar	scalar elementary arithmetics, scalar transcendental functions	register (scalar value), immediate
Logical	Vector	vector compare (greater than, equal), vector logical operations (and, or, inverter), vector greater than merge	register (vector address/size, scalar)
	Scalar	scalar compare, scalar logical operations	register (scalar), immediate

Cambricon Performance Results

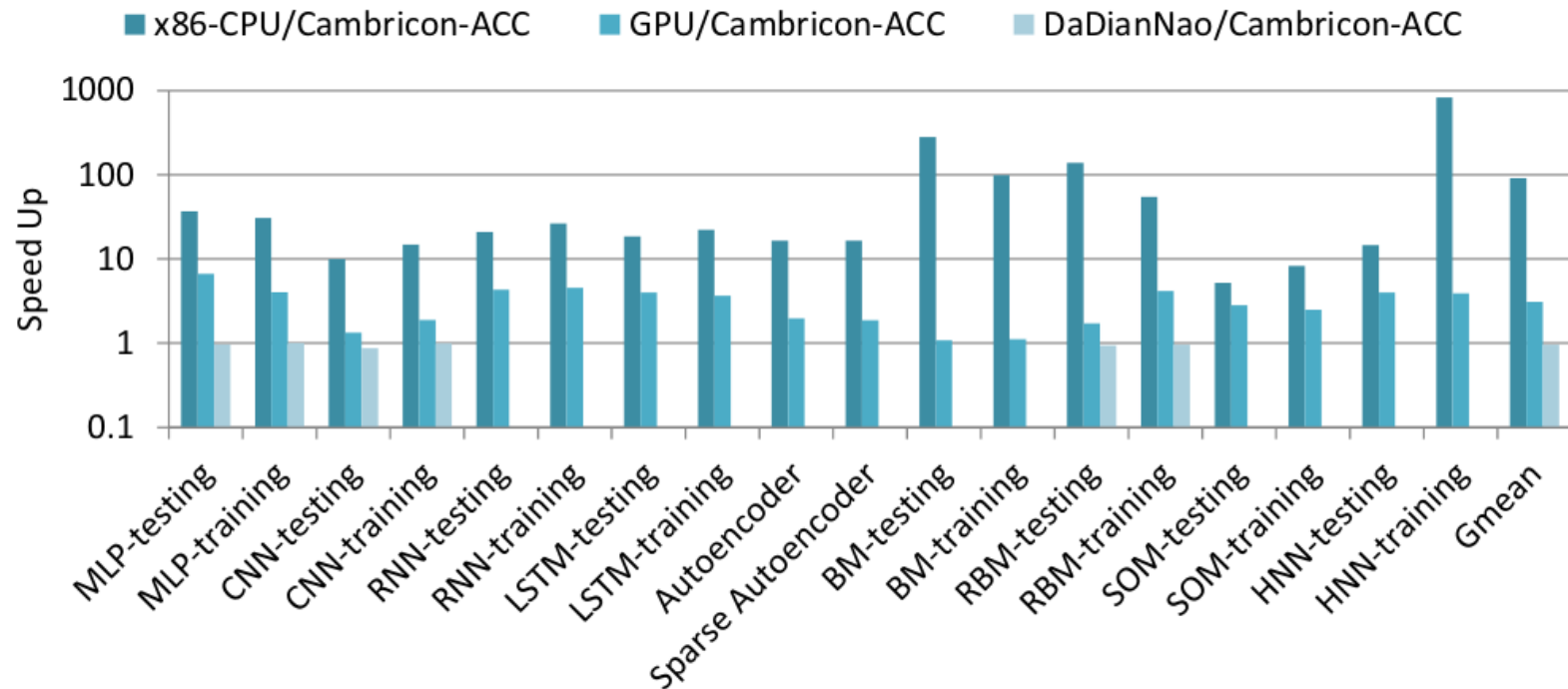


Figure 12. The speedup of Cambricon-ACC against x86-CPU, GPU, and DaDianNao.