Impact of Instruction Set Architecture on Machine Learning Workloads

Jeung Moon Lee, Hyesoon Kim, Hyojong Kim, Pranith Kumar

1 Background and Motivation

As neural network (NN) models are becoming more sophisticated, the hardware processing the workload is required to process more computationally complex and memory intensive data. However, general-purpose instruction-set architectures (ISAs) (e.g., ARM, MIPS, or x86) are not very energy efficient because of their flexibility in supporting various workloads. New NN-specific architectures (e.g., Cambricon) have been proposed to build energy-efficient hardware [2].

However, many Internet-of-things (IoT) devices or embedded systems are still based on either ARM or x86 processors. In such platforms, machine learning (ML) workloads are just one of their applications and having a separate ML accelerator increases the cost of devices significantly. Hence, ARM or x86 devices are still widely used.

In such IoT or embedded systems, which ISA is a better choice? The paper is trying to answer this question. There have been research on comparisons between x86 and ARM [1], but there has been no study for ML workloads specifically. Hence, in this paper, we would like to evaluate the ISA effect on ML workloads only. To separate the architecture and compiler effect as much as possible, we compare two workloads in a simulation environment. For ML workload, we choose the Darknet benchmark suite because it is based on C++ so that we can evaluate CPU only aspect and it also has many latest NNs. And then, we identify frequently executed functions in four NN models (AlexNet, Cifar, ResNet-18, and VGG-16).

2 Approach and Results

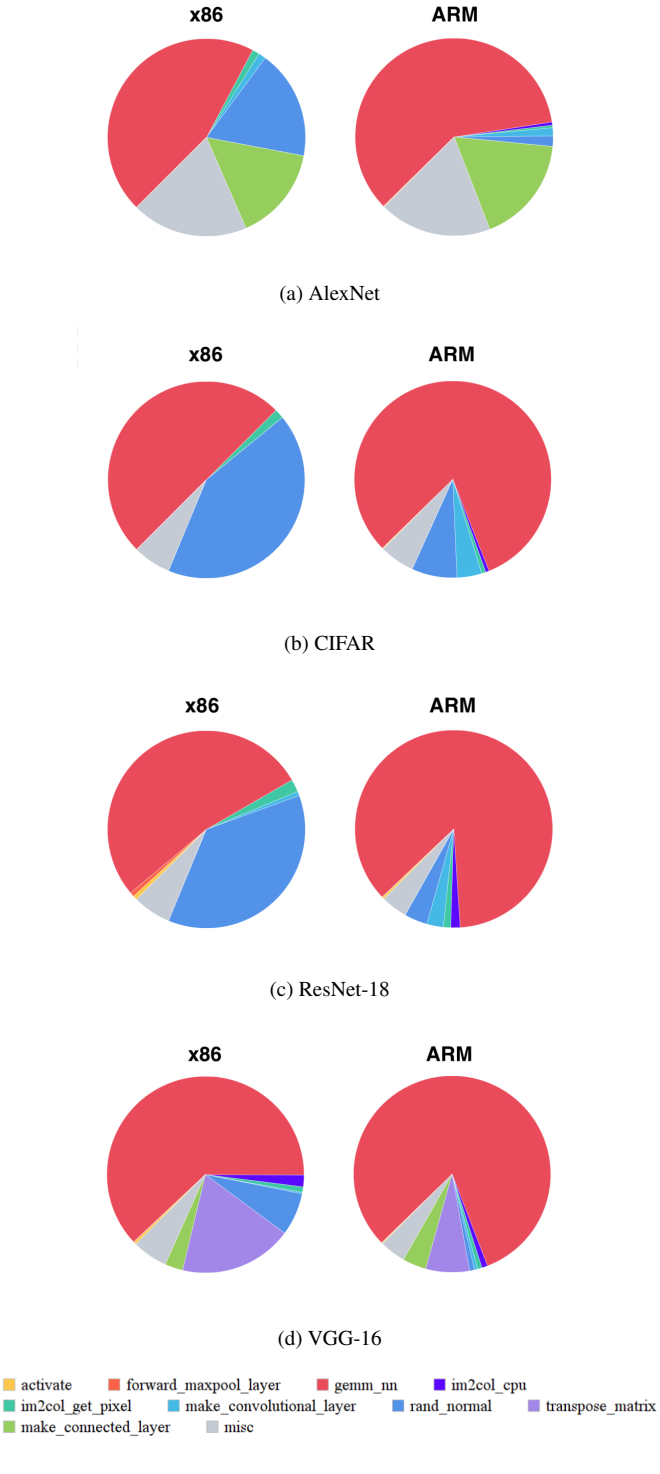
For both ISAs, ARM and x86, we profile time spent on each function in Darknet while running inferences on multiple NNs. The same source code is compiled for both ARM and x86 using gcc. We use gprof for profiling. When running inferences, we use four pre-trained, image classification models: AlexNet, Cifar, ResNet-18, and VGG-16. The results are shown in Figures 1a, 1b, 1c, and 1d. The results are collected from native executions on Xeon processors for x86, and from QEMU executions running on x86 for ARM64.

Based on the profiled data, we select nine functions that take most time in running the inference as follows: activate(), forward_maxpool_layer(), gemm_nn(), im2col_cpu(), im2col_get_pixel(), make_connected_layer(), make_convolutional_layer(), rand_normal(), and transpose_matrix(). The detailed results are summarized in Table 1.

	% Time	activate	fwd_layer	gemm	im2col_cpu	im2col_pixel	conn_layr	conv_layr	randn	transpose	misc
a64	Alexnet	0.06	0.12	59.89	0.53	0.41	17.59	1.34	1.7	0	18.36
	CIFAR	0.14	0.05	81.3	0.57	0.68	0	4.07	7.44	0	5.75
	Resnet18	0.34	0.08	86.05	1.52	1.19	0	2.71	3.78	0	4.33
	VGG16	0.14	0.07	81.45	0.94	0.74	3.88	0.52	0.79	7.17	4.3
	AVG	0.17	0.08	77.1725	0.89	0.755	5.3675	2.16	3.4275	1.7925	8.185
x86	Alexnet	0	0	45.24	0	1.19	15.48	1.19	17.86	0	19.04
	CIFAR	0	0	48.49	0	1.52	0	3.03	40.91	0	6.05
	Resnet18	0.69	0.69	52.78	0	2.08	0	0.69	36.81	0	6.26
	VGG16	0.4	0.11	62.06	1.93	0.91	2.94	0.23	7.02	18.57	5.83
	AVG	0.2725	0.2	52.1425	0.4825	1.425	4.605	1.285	25.65	4.6425	9.295

Table 1: Percentage of time spent in neural network inference.

We see that most time is spent on running gemm_nn(), a function for general matrix-to-matrix multiplication. Both ISAs spend relatively same portion of time for each function except for rand_normal(), a function that generates a random variable with normal distribution. We



(a) AlexNet

(b) CIFAR

(c) ResNet-18

(d) VGG-16

Fig. 1: Darknet profile results on x86 vs. ARM

observe that more time is spent in running rand_normal() in x86 than in ARM.

For the nine functions, we create traces that contain instructions for executing the particular function. We use Pin to generate x86 traces and qsim to generate ARM traces. As seen in Figure 2, x86 has more number of static instructions than ARM. This is because x86 requires more instructions for register spills due to fewer number of registers.

		activate	fwd_layer	gemm	im2col_cpu	im2col_pixel	conn_layr	conv_layr	randn	transpose
a64	arithmetic	0	33	43	24	4	17	35	2	8
	logic	3	18	25	8	2	23	22	1	5
	branch	6	16	22	8	6	51	62	10	8
	floating point	3	1	8	0	0	3	11	7	0
	move	19	82	59	73	7	204	268	32	28
	vector	0	2	2	0	0	1	1	0	0
	misc.	4	4	5	0	0	13	24	6	1
x86	arithmetic	2	36	26	21	9	12	32	3	7
	logic	6	26	34	9	6	37	42	6	9
	branch	8	18	20	9	8	56	64	12	10
	floating point	6	2	15	0	0	3	11	7	0
	move	12	119	105	73	29	228	309	15	36
	vector	0	3	6	0	0	2	2	0	0
	misc.	7	6	8	6	3	11	21	6	4

Table 2: Number of instructions for ARM and x86 in each function

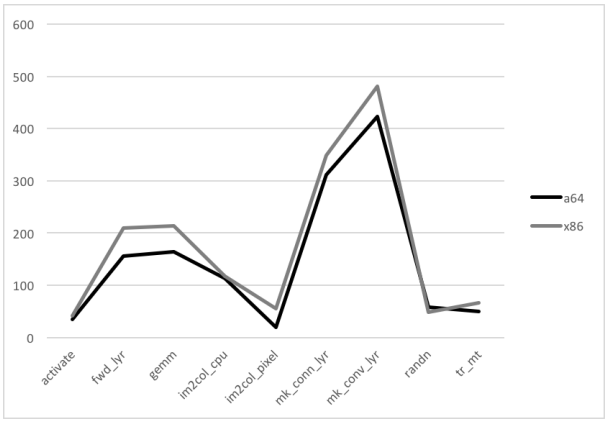


Fig. 2: Number of instructions for ARM and x86 in each function

We also analyze the types of instructions in each ISA. We categorize instructions into seven types: arithmetic, logic, branch, floating point, move, vector, and miscellaneous. The detailed instructions are described in Table 3 and their distributions are shown in Figure 3. In general, both ISAs show similar instruction group distributions which make sense since they are eventually running the same applications. Floating points are significant in activate, gemm_nn and rand_normal. Surprisingly, in terms of static instructions, move instruction categories are the majority of all instructions. Vector instructions are shown only in a few benchmarks, which should be improved by compilers since many applications have data parallelism.

a64	Arithmetic	add, sub, msub, mul, sdiv, madd, smull, udiv
	Logic	and, neg, lsl, lsr, orr, fcmpe, cmp, tst, fcmp
	Branch	b.hi, b.le, ret, b, b.eq, b.gt, b.ne, cbz, b.ls, cbnz, bl, b.ge, tbnz, b.mi, b.cc
	Floating Point	fmadd, fmul, fsub, fmaxnm, fdiv, fsqrt, fadd
	Move	ldr, mov, movk, msr, ldp, stp, str, mvn, fmov
	Vector	dup(v), fmla(v), add(v), fmov(v)
	Misc.	sbfiz, nop, sxtw, ubfx, adrp, fcvt, scvtf, csel, cset
x86	Arithmetic	add, sub, addl, imul, idiv, imul, div, idivl
	Logic	pxor, xor, and, neg, not, shl, shr, or, sar, cmp, test, cmpl, comisd
	Branch	ja, jbe, jmp, callq, retq, jle, jmpq, jne, jb, je, jg, jge, js
	Floating Point	addsd, comiss, mulsd, subss, maxss, addss, mulss, divsd, sqrtsd
	Move	mov, pop, push, cmova, lea, movl, movslq, xchg, movq, movsd, movss
	Vector	addps, movups, movlps, shufps, movaps, mulps
	Misc	nopl, nopw, nop, rep, cltq, cltd, cvtsd2ss, cvtss2sd, cvtss2sdl, maxss, setg, setl, setae, cvtsi2sd, cvtsi2sdl

Table 3: Categorization of instructions into seven types

We then simulate the generated traces using an architecture simulator, Macsim. In this paper, we focus on cache hit rate differences for x86 and ARM. As shown in Figure 4, both ISAs show high L1 hit rates but

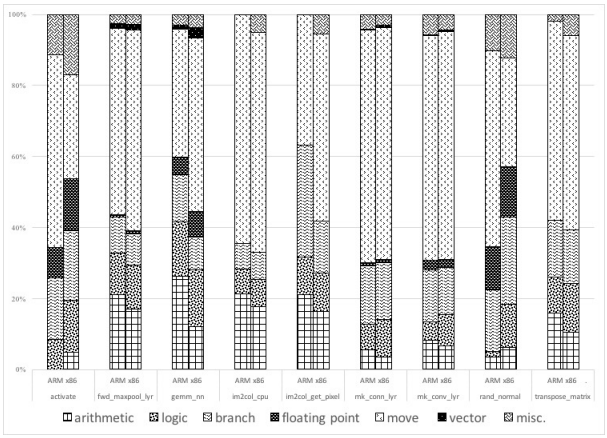
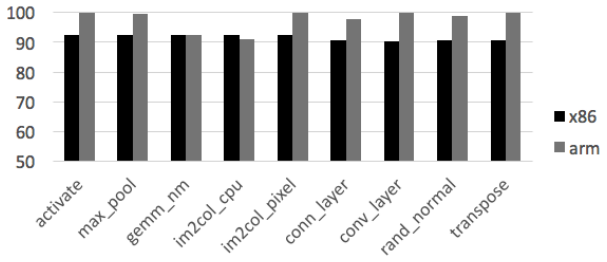
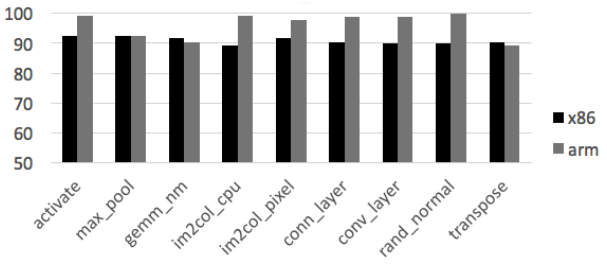


Fig. 3: Groups of Instructions

the hit rates are slightly different between x86 and ARM. On the other hand, L2 hit rate for all benchmarks in both ISAs are 0 meaning the working set size is much larger than L2. Although working set size is heavily dependent on the applications, not ISA, due to the register file usage patterns and ISA effects, they generate slightly different memory working set size which leads to different L1 hit rates. We will further investigate the differences in our future work.



(a) AlexNet



(b) VGG-16

Fig. 4: L1 Hit Rate for x86 vs. ARM

3 Conclusions and Future Works

In this paper, we analyzed the ISA effects on machine learning workloads. Due to ISA differences especially the number of registers, the instruction mixtures vary slightly but arithmetic, floating points, and vector instructions show similar trends in both ISAs. We found that the compiler produces different execution profile and L1 hit rate for functions calls which should be investigated further. We will also include other performance results such as instructions per cycle(IPC) and branch prediction accuracy on the simulator in the final version of the poster. In our future work, we will also evaluate GPU versions of the same functions to evaluate all three different ISAs.

REFERENCES

[1] E. Blem, K. Sankaralingam, and J. Menon. A detailed analysis of contemporary arm and x86 architectures. Technical report, UW-Madison, 2013.

[2] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen. Cambricon: an instruction set architecture for neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 393–405, 2016.