More is Less

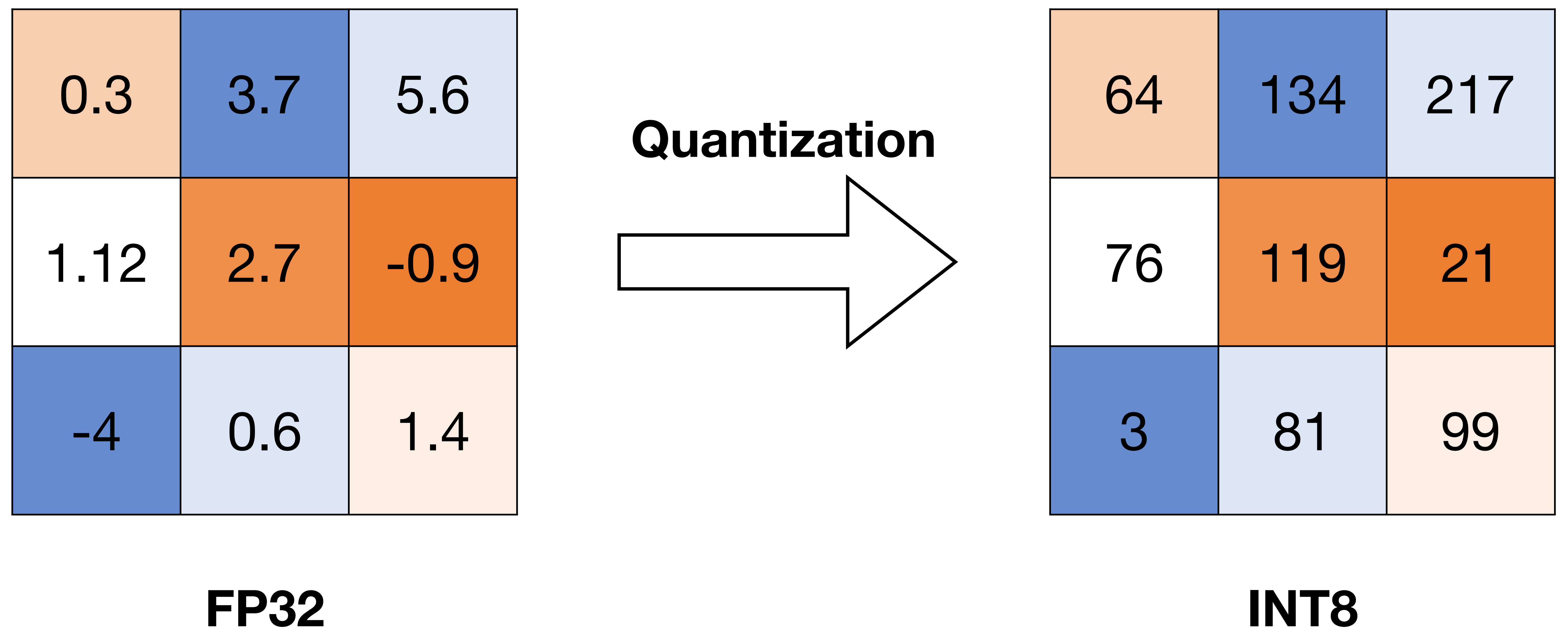
Byte-quantized models are faster than bit-quantized models on the edge

Pengfei Zhang, Chenxia Han, and Eric Lo

The Chinese University of Hong Kong

Quantization

- Store and compute tensors at lower bit-widths

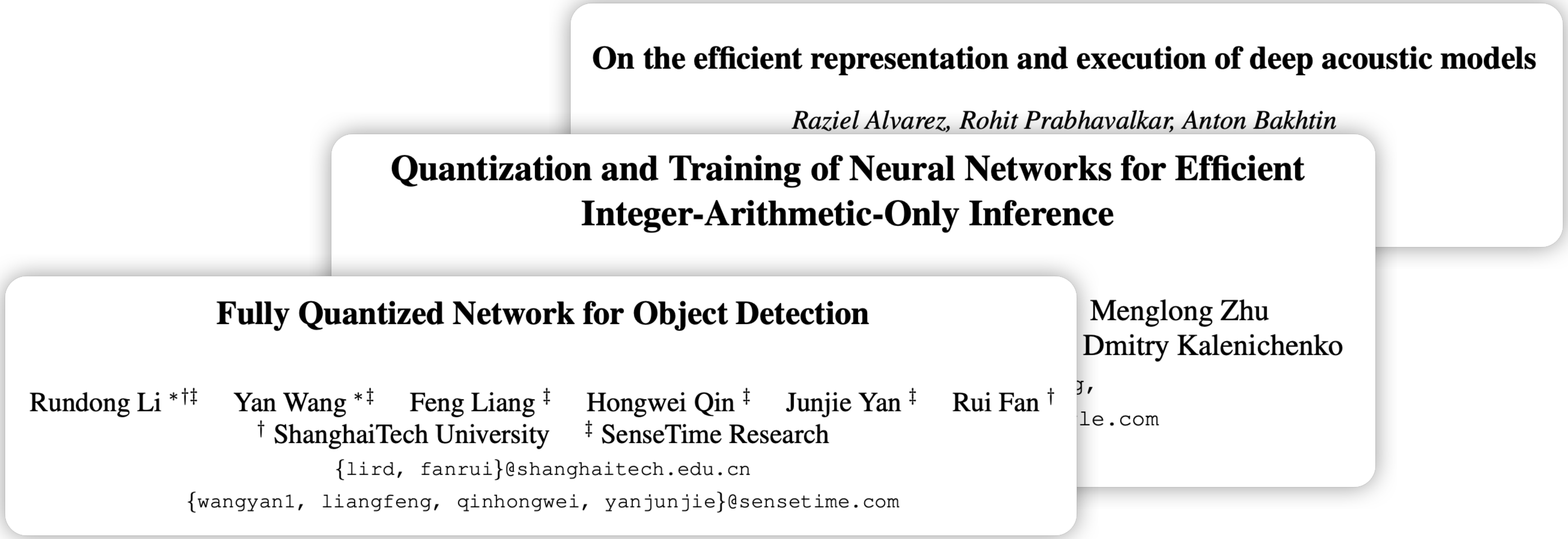


Quantization

- Store and compute tensors at lower bit-widths
  - Reduce model size
  - Speed up inference

Quantization

- Store and compute tensors at lower bit-widths
  - Retain similar accuracy for various tasks



On the efficient representation and execution of deep acoustic models

Raziel Alvarez, Rohit Prabhavalkar, Anton Bakhtin

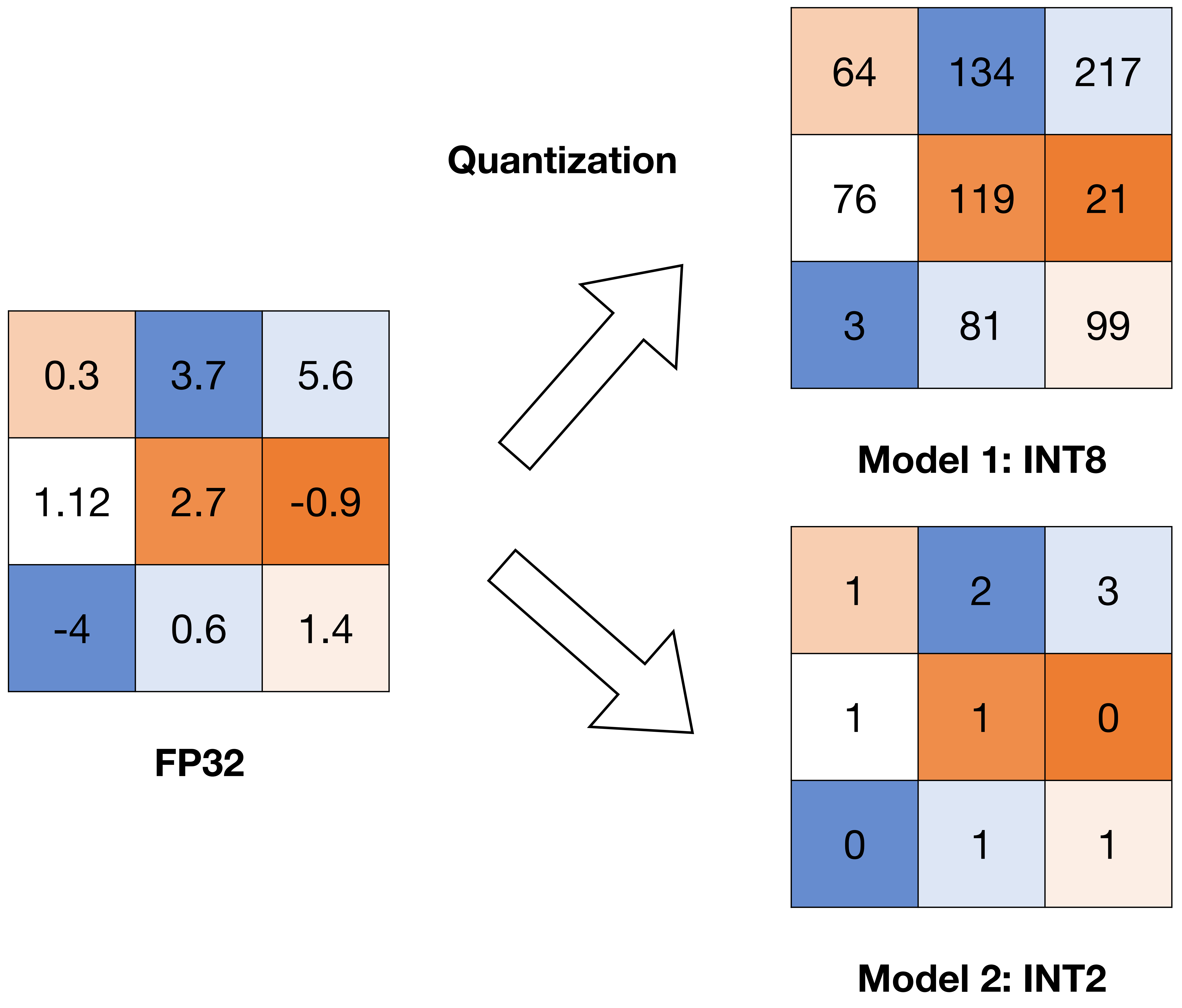
Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference

Menglong Zhu
Dmitry Kalenichenko

Fully Quantized Network for Object Detection

Rundong Li [*†‡] Yan Wang [*‡] Feng Liang [‡] Hongwei Qin [‡] Junjie Yan [‡] Rui Fan [†]
[†] ShanghaiTech University [‡] SenseTime Research
{lird, fanrui}@shanghaitech.edu.cn
{wangyan1, liangfeng, qinhongwei, yanjunjie}@sensetime.com

Bit vs. Byte Quantization



Does Model 2 run faster than Model 1?

On specialized hardware, YES.
But on ARM, NO.

Existing Works

Bit-width	Package	Instructions
INT{1-7}	Riptide	bitwise OPs, popcount bitglue, bitpack
INT8	TFLite	gemm(im2col)
	QNNPACK	SMLAL

Existing Works

Bit-width	Package	Instructions
INT{1-7}	Riptide	bitwise OPs, popcount bitglue, bitpack
INT8	TFLite	gemm(im2col)
	QNNPACK	SMLAL

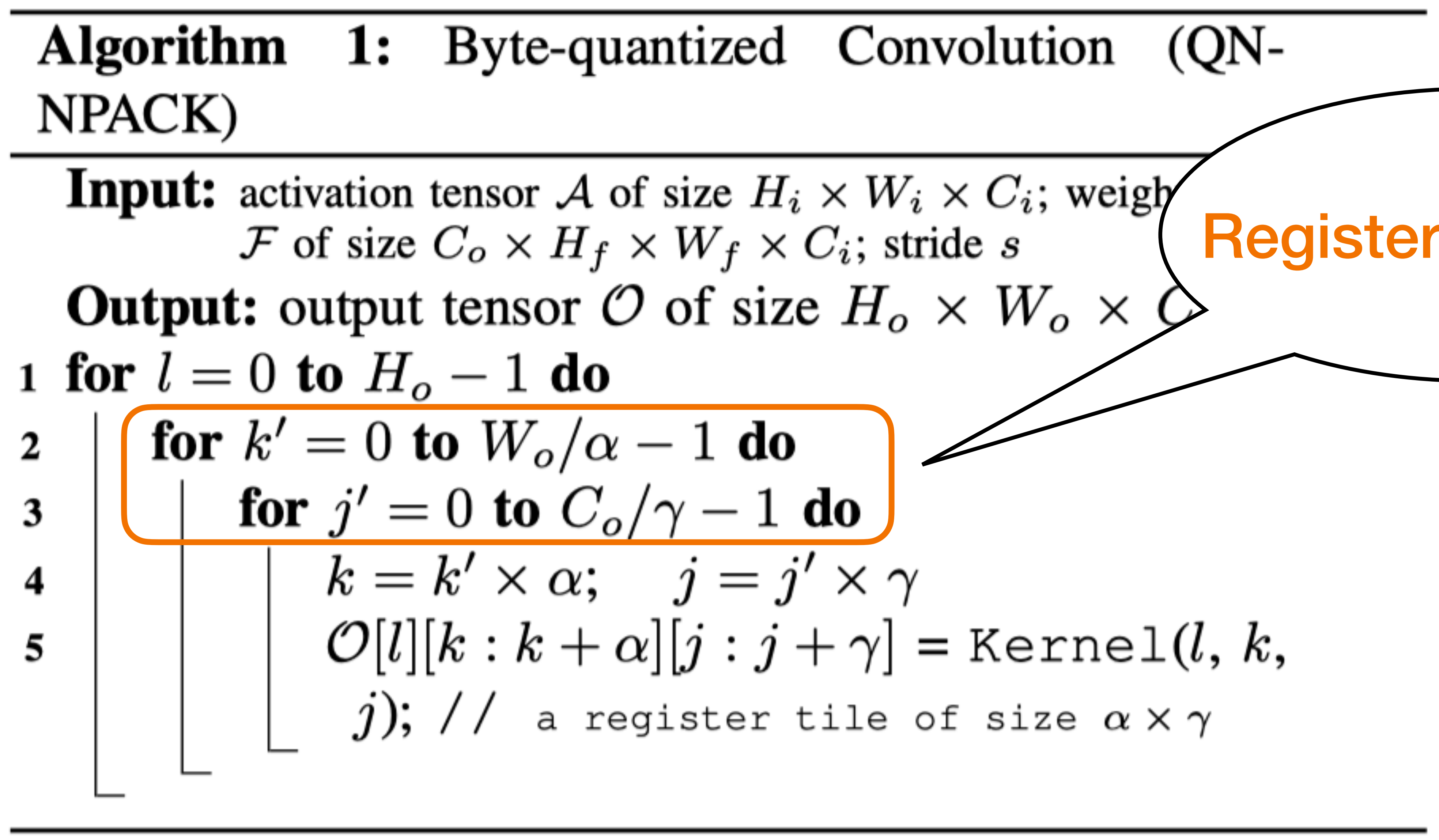
Overhead

Existing Works

Bit-width	Package	Instructions
INT{1-7}	Riptide	bitwise OPs, popcount bitglue, bitpack
INT8	TFLite	gemm(im2col)
	QNNPACK	SMLAL

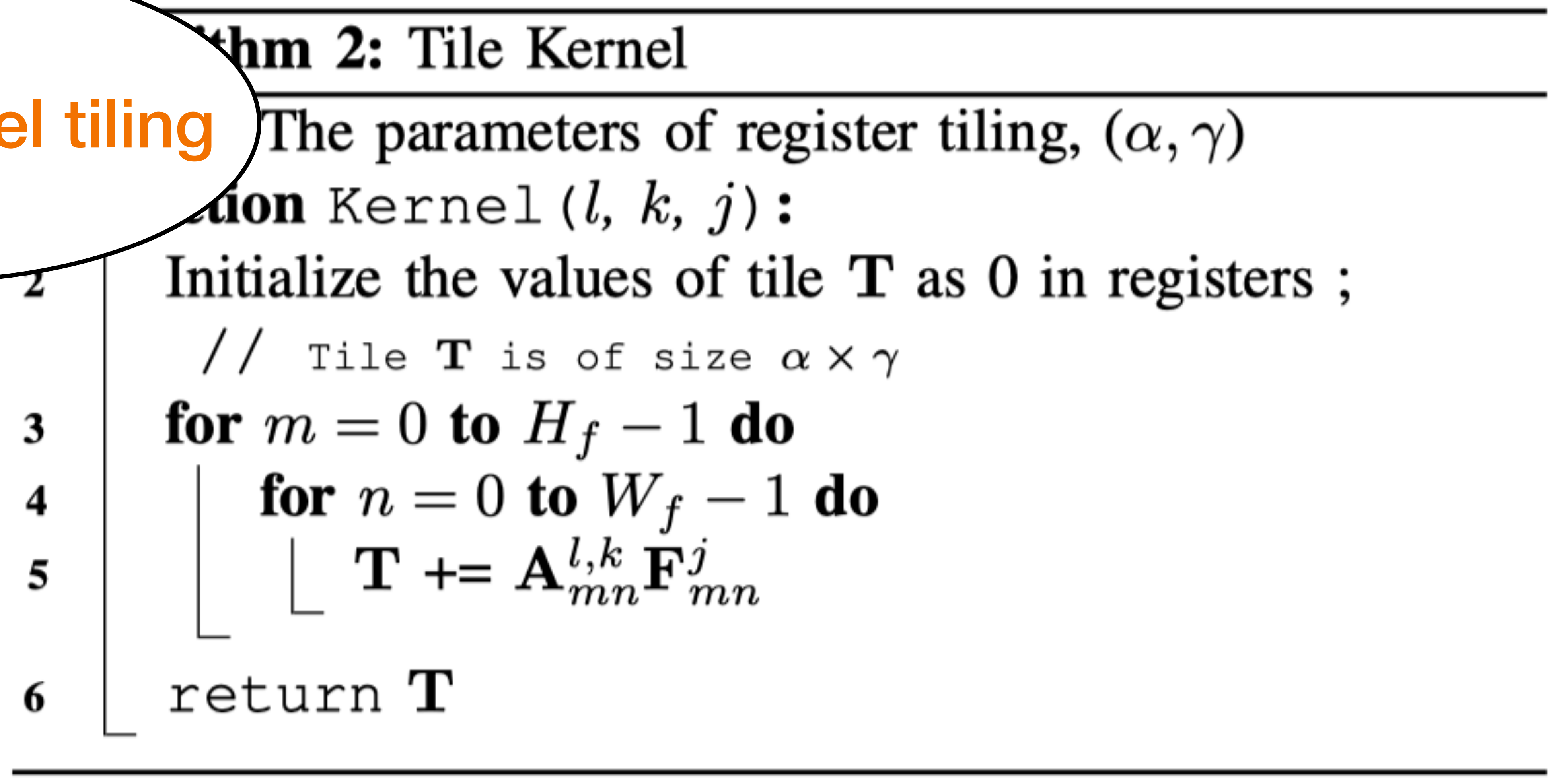
Serious cache misses
Underutilization of SIMD registers
Low parallelism

QNNPack



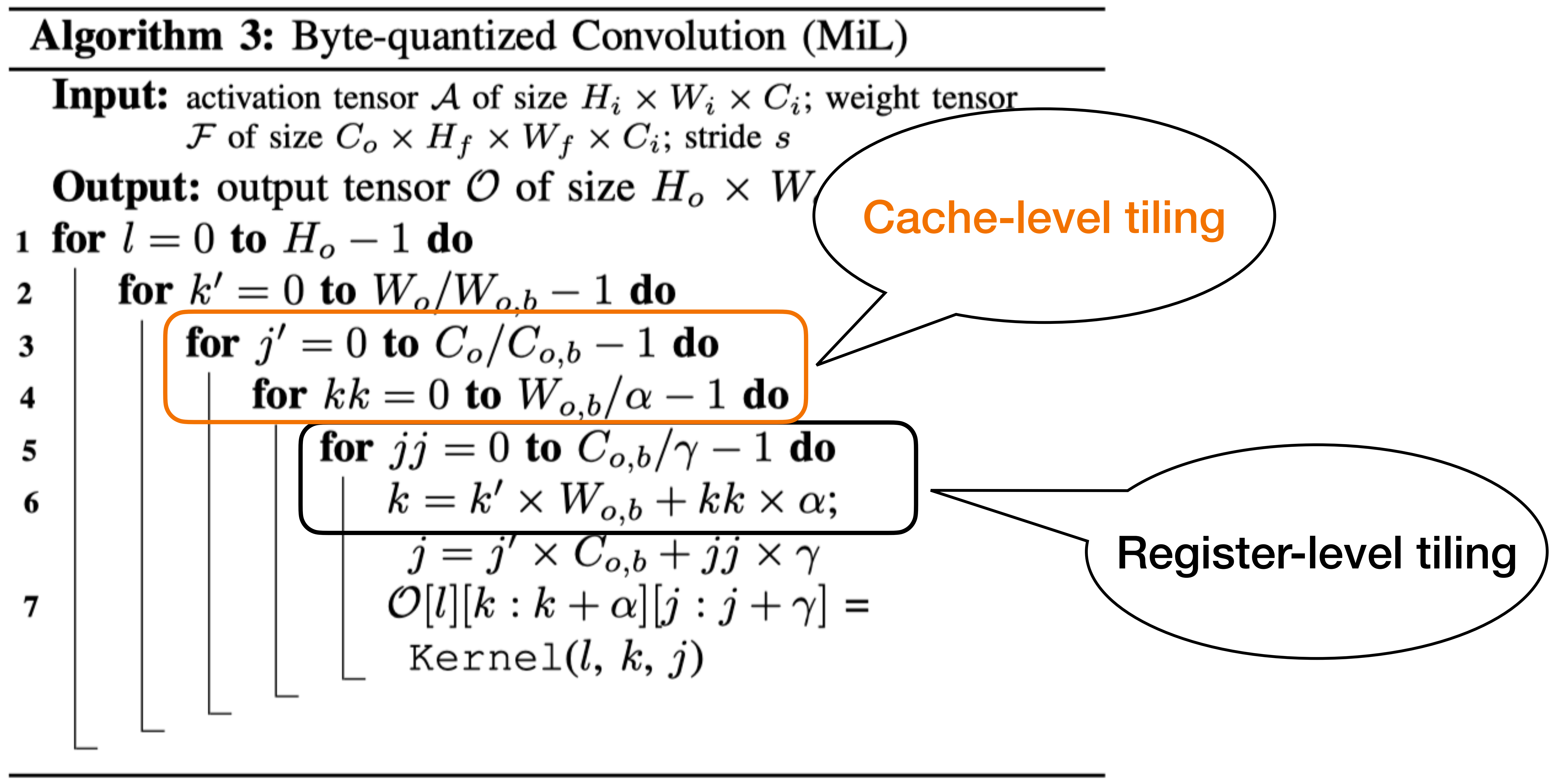
```
Algorithm 1: Byte-quantized Convolution (QN-NPACK)
Input: activation tensor $\mathcal{A}$ of size $H_i \times W_i \times C_i$; weigh
       $\mathcal{F}$ of size $C_o \times H_f \times W_f \times C_i$; stride $s$
Output: output tensor $\mathcal{O}$ of size $H_o \times W_o \times C$
1 for $l = 0$ to $H_o - 1$ do
2     for $k' = 0$ to $W_o/\alpha - 1$ do
3         for $j' = 0$ to $C_o/\gamma - 1$ do
4             $k = k' \times \alpha$;    $j = j' \times \gamma$
5             $\mathcal{O}[l][k : k+\alpha][j : j+\gamma]$ = Kernel($l$, $k$,
               $j$); // a register tile of size $\alpha \times \gamma$
```

Register-level tiling



```
thm 2: Tile Kernel
The parameters of register tiling, $(\alpha, \gamma)$
tion Kernel($l$, $k$, $j$):
2     Initialize the values of tile $\mathbf{T}$ as 0 in registers ;
       // Tile $\mathbf{T}$ is of size $\alpha \times \gamma$
3     for $m = 0$ to $H_f - 1$ do
4         for $n = 0$ to $W_f - 1$ do
5             $\mathbf{T}$ += $\mathbf{A}^{l,k}_{mn}\mathbf{F}^{j}_{mn}$
6     return $\mathbf{T}$
```

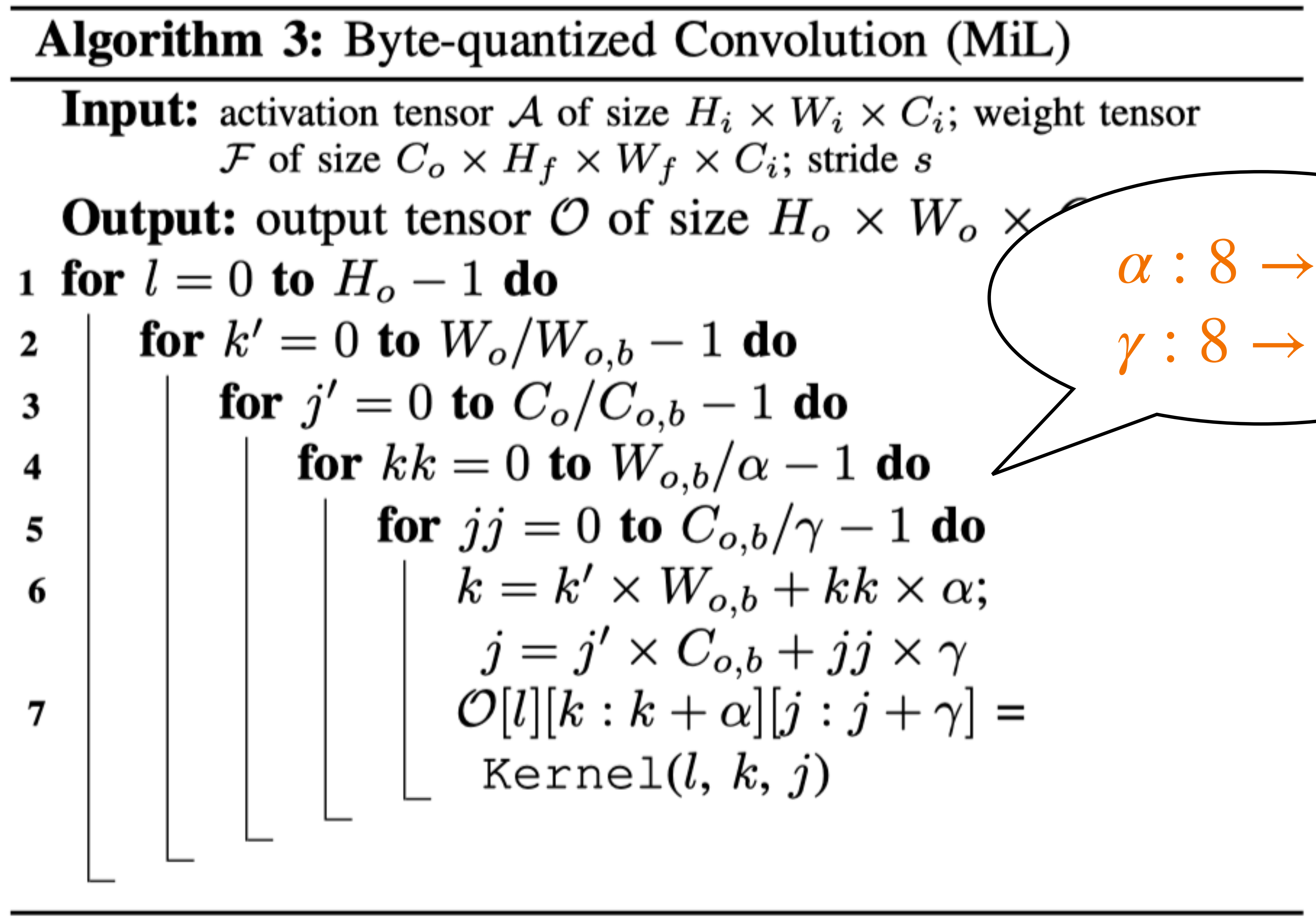
Problem 1: cache miss



Algorithm 3: Byte-quantized Convolution (MiL)

```
Input: activation tensor $\mathcal{A}$ of size $H_i \times W_i \times C_i$; weight tensor
       $\mathcal{F}$ of size $C_o \times H_f \times W_f \times C_i$; stride $s$
Output: output tensor $\mathcal{O}$ of size $H_o \times W$
1 for $l = 0$ to $H_o - 1$ do
2     for $k' = 0$ to $W_o/W_{o,b} - 1$ do
3         for $j' = 0$ to $C_o/C_{o,b} - 1$ do
4             for $kk = 0$ to $W_{o,b}/\alpha - 1$ do
5                 for $jj = 0$ to $C_{o,b}/\gamma - 1$ do
6                     $k = k' \times W_{o,b} + kk \times \alpha$;
                      $j = j' \times C_{o,b} + jj \times \gamma$
7                     $\mathcal{O}[l][k : k + \alpha][j : j + \gamma]$ =
                      Kernel($l$, $k$, $j$)
```

Problem 2: underutilized registers

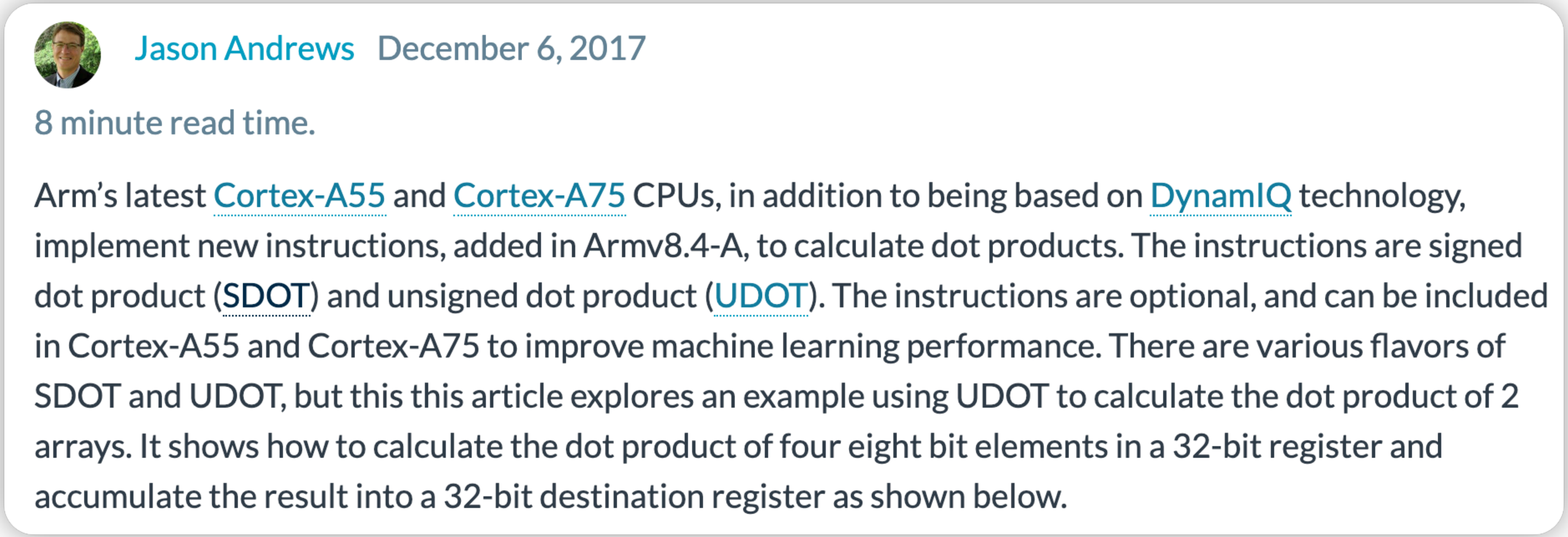


Algorithm 3: Byte-quantized Convolution (MiL)

```
Input: activation tensor $\mathcal{A}$ of size $H_i \times W_i \times C_i$; weight tensor
       $\mathcal{F}$ of size $C_o \times H_f \times W_f \times C_i$; stride $s$
Output: output tensor $\mathcal{O}$ of size $H_o \times W_o \times$
1 for $l = 0$ to $H_o - 1$ do
2     for $k' = 0$ to $W_o/W_{o,b} - 1$ do
3         for $j' = 0$ to $C_o/C_{o,b} - 1$ do
4             for $kk = 0$ to $W_{o,b}/\alpha - 1$ do
5                 for $jj = 0$ to $C_{o,b}/\gamma - 1$ do
6                     $k = k' \times W_{o,b} + kk \times \alpha$;
                      $j = j' \times C_{o,b} + jj \times \gamma$
7                     $\mathcal{O}[l][k : k + \alpha][j : j + \gamma]$ =
                      Kernel($l$, $k$, $j$)
```

$\alpha : 8 \rightarrow 12$
$\gamma : 8 \rightarrow 8$

Problem 3: low parallelism



Jason Andrews December 6, 2017

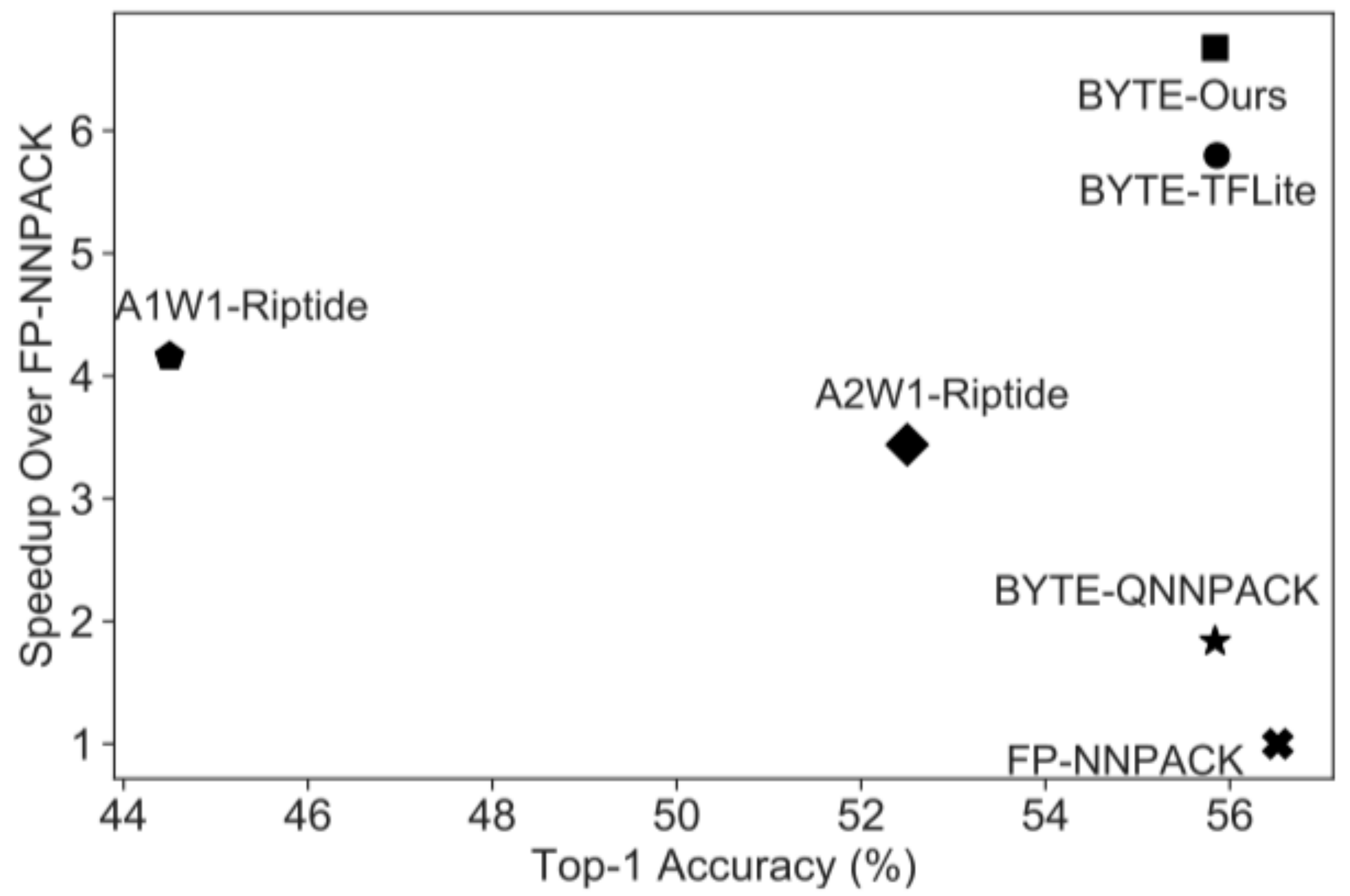
8 minute read time.

Arm’s latest Cortex-A55 and Cortex-A75 CPUs, in addition to being based on DynamIQ technology, implement new instructions, added in Armv8.4-A, to calculate dot products. The instructions are signed dot product (SDOT) and unsigned dot product (UDOT). The instructions are optional, and can be included in Cortex-A55 and Cortex-A75 to improve machine learning performance. There are various flavors of SDOT and UDOT, but this this article explores an example using UDOT to calculate the dot product of 2 arrays. It shows how to calculate the dot product of four eight bit elements in a 32-bit register and accumulate the result into a 32-bit destination register as shown below.

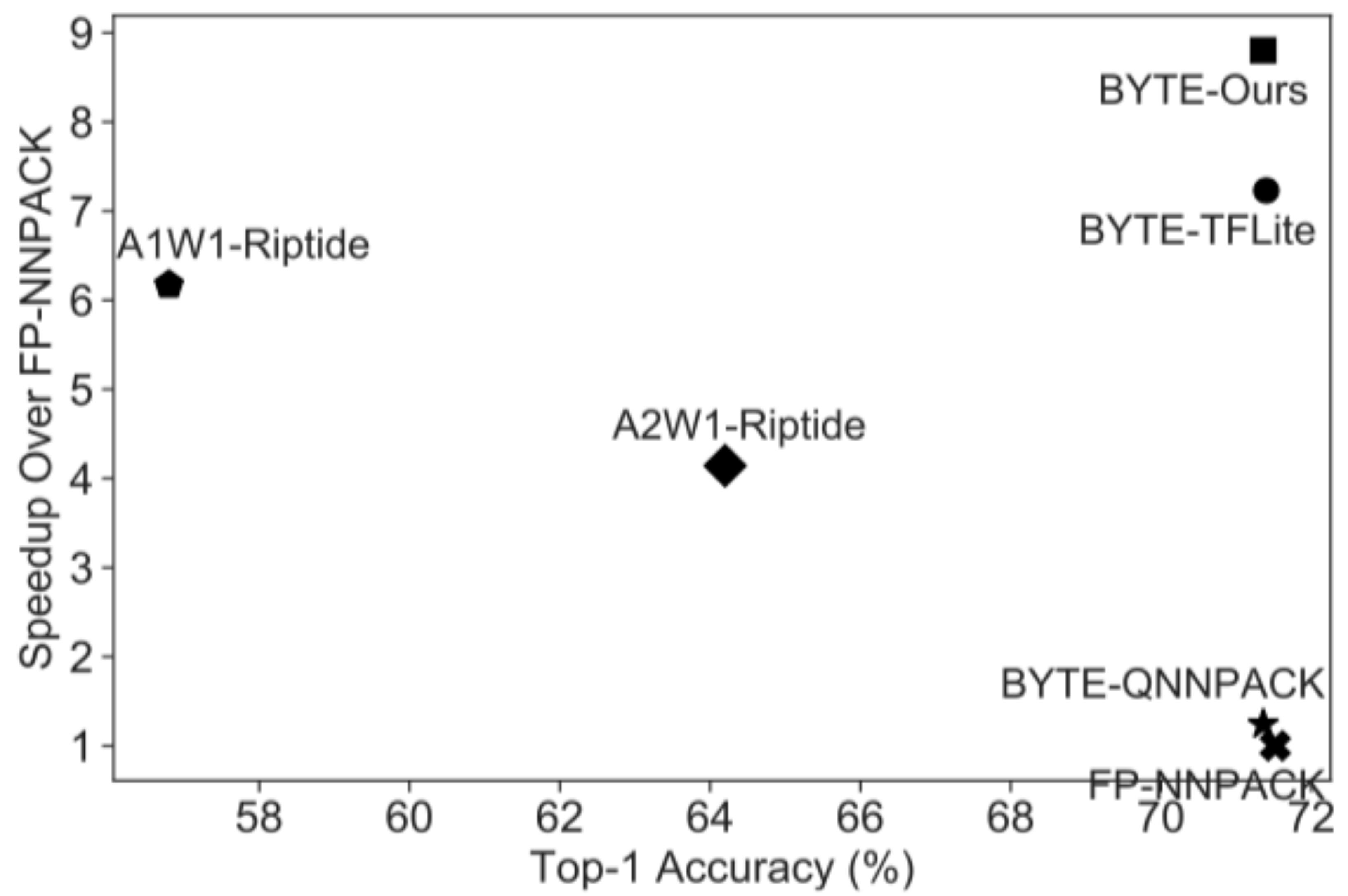
Experiments

- FP-NNPACK
- BYTE-TFLite
- BYTE-QNNPACK
- BYTE-Ours
- AxWy-Riptide

Experiments

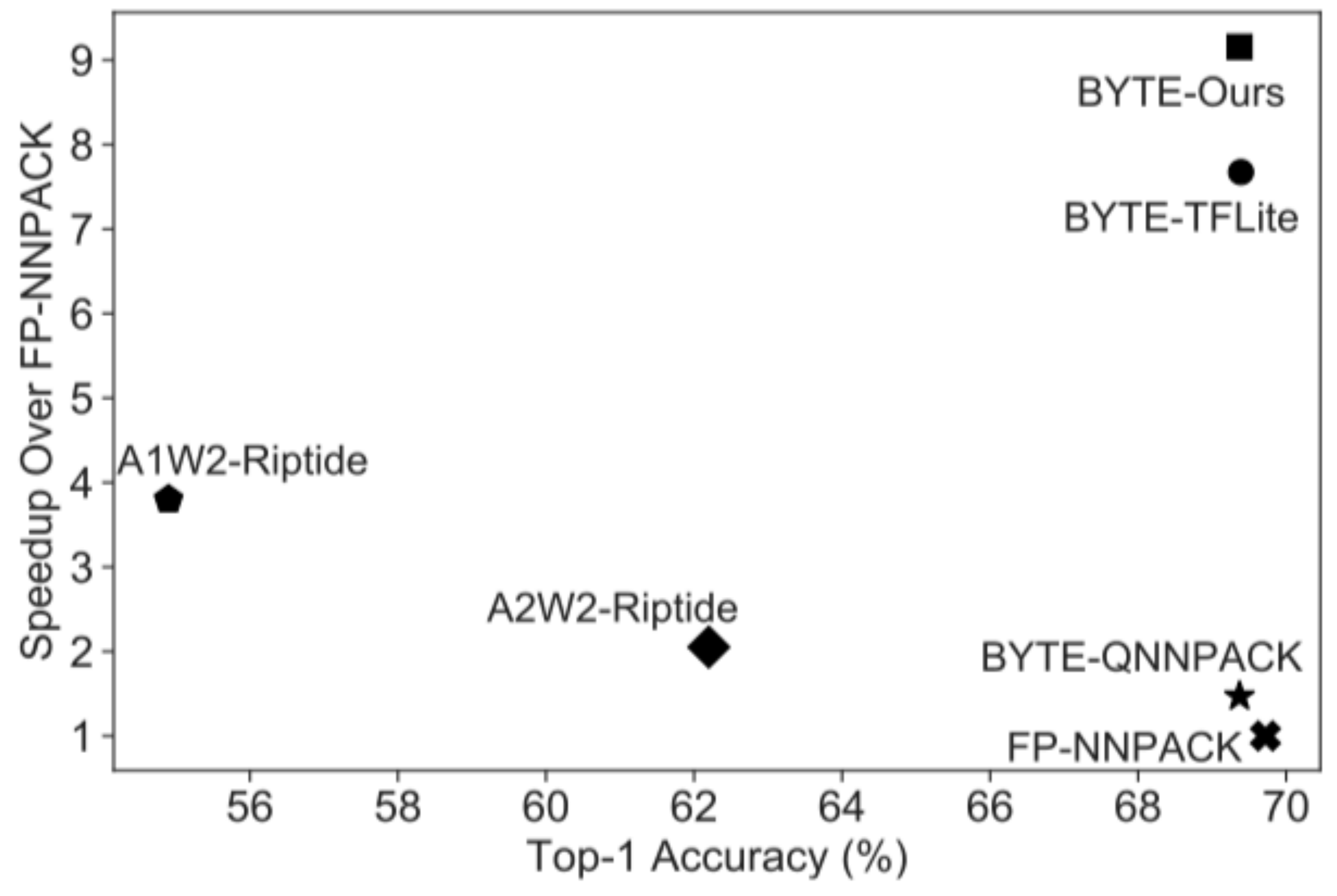


(a) AlexNet

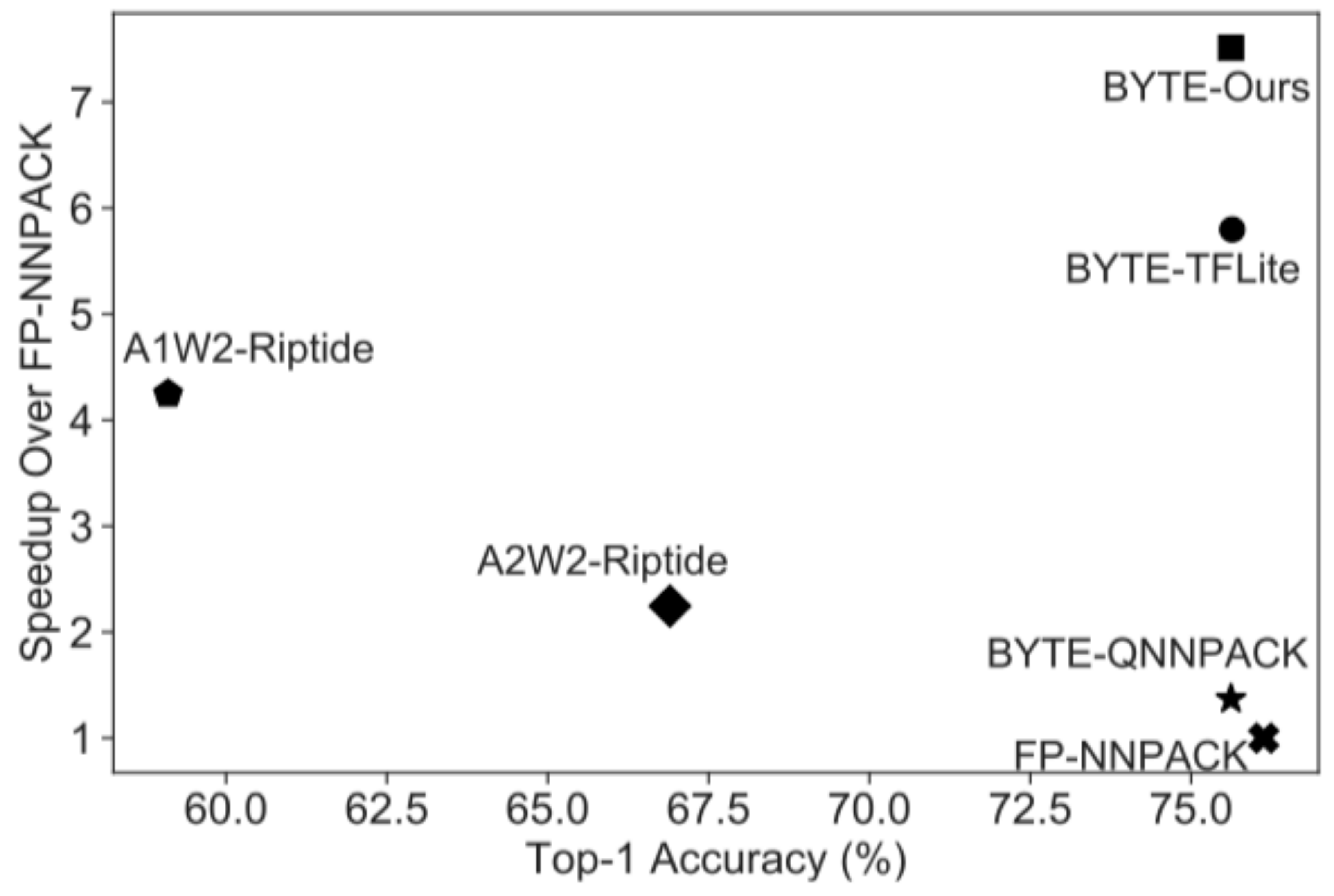


(b) VGGNet

Experiments

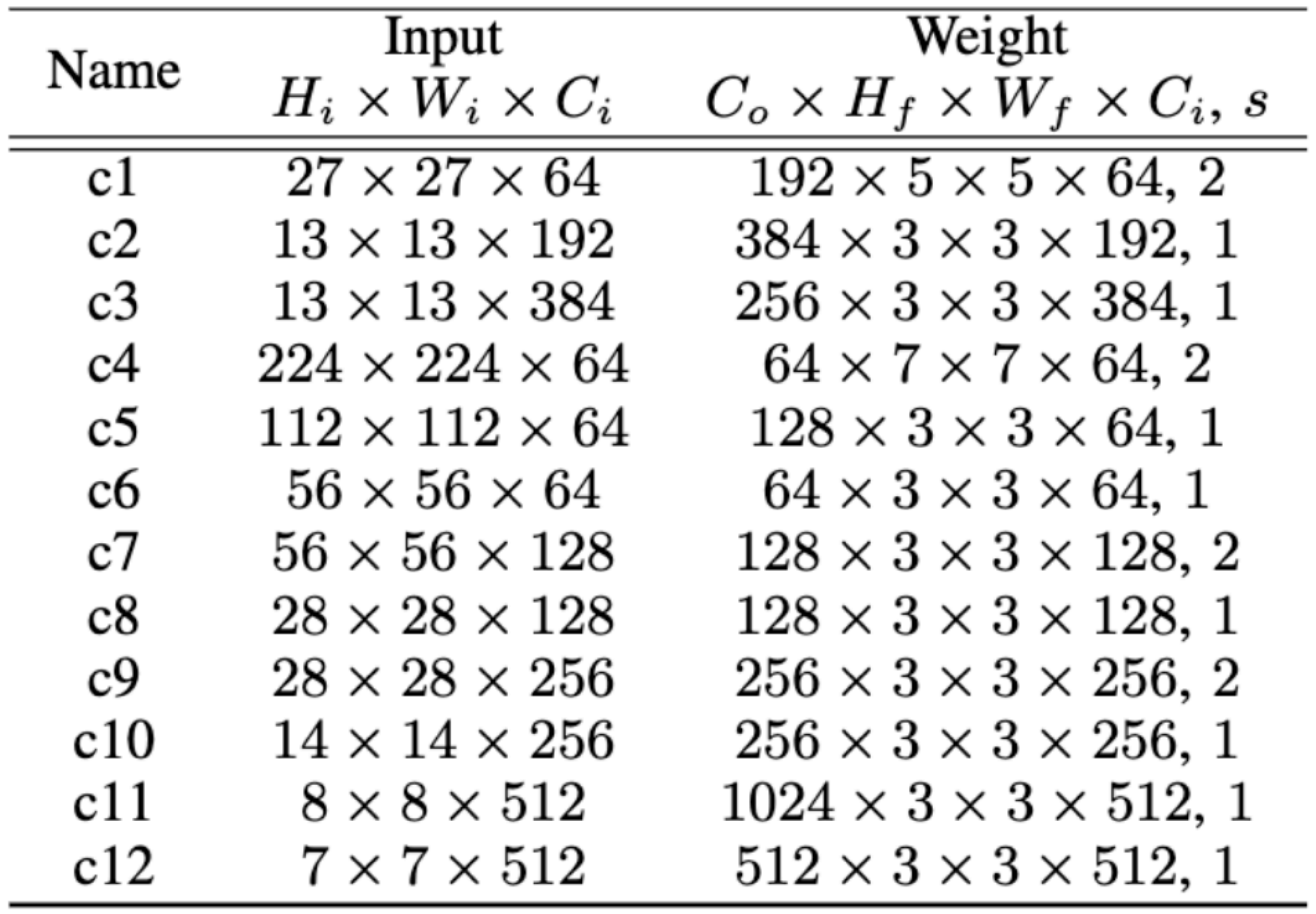


(c) ResNet18



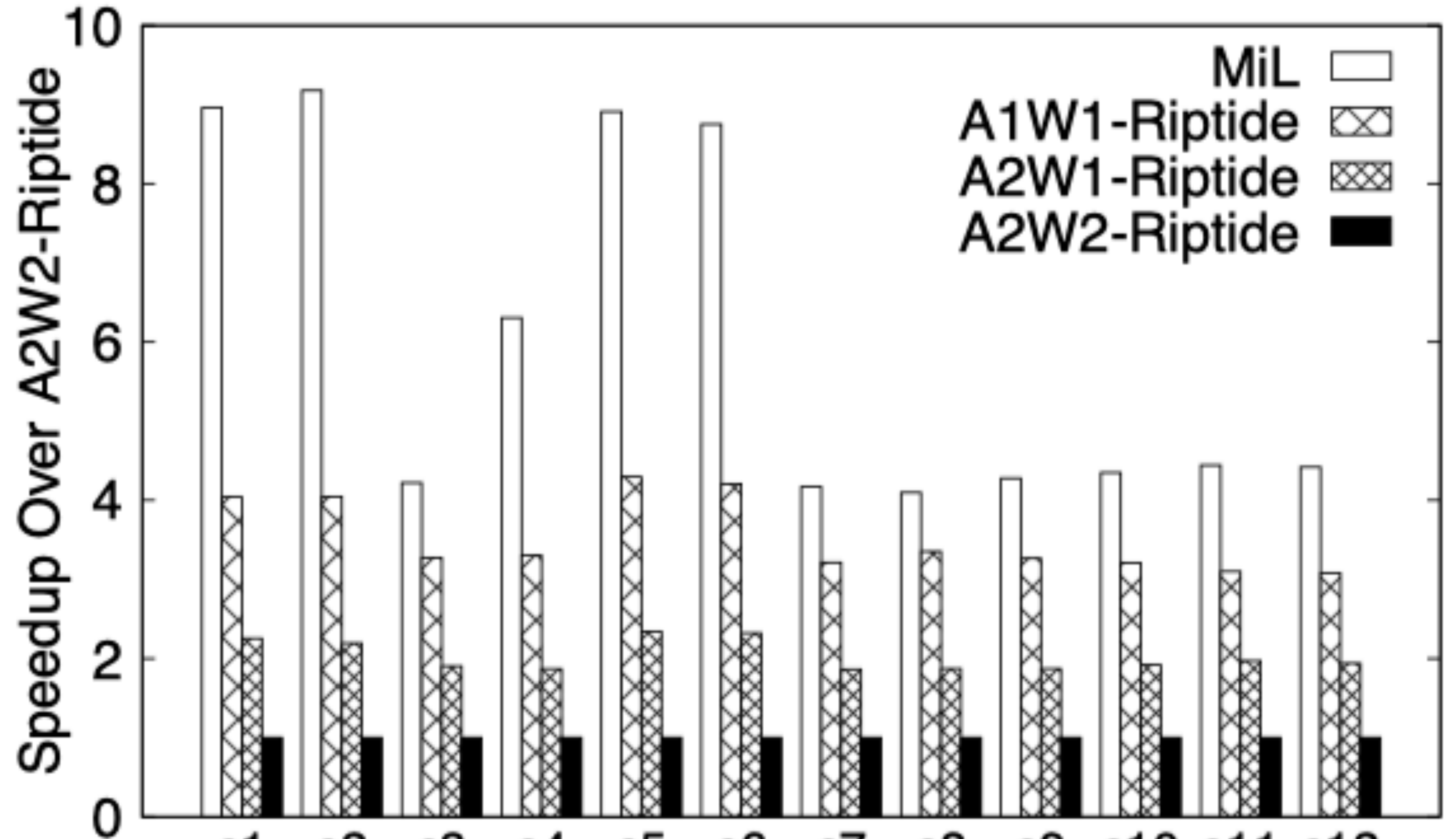
(c) ResNet50

Experiments

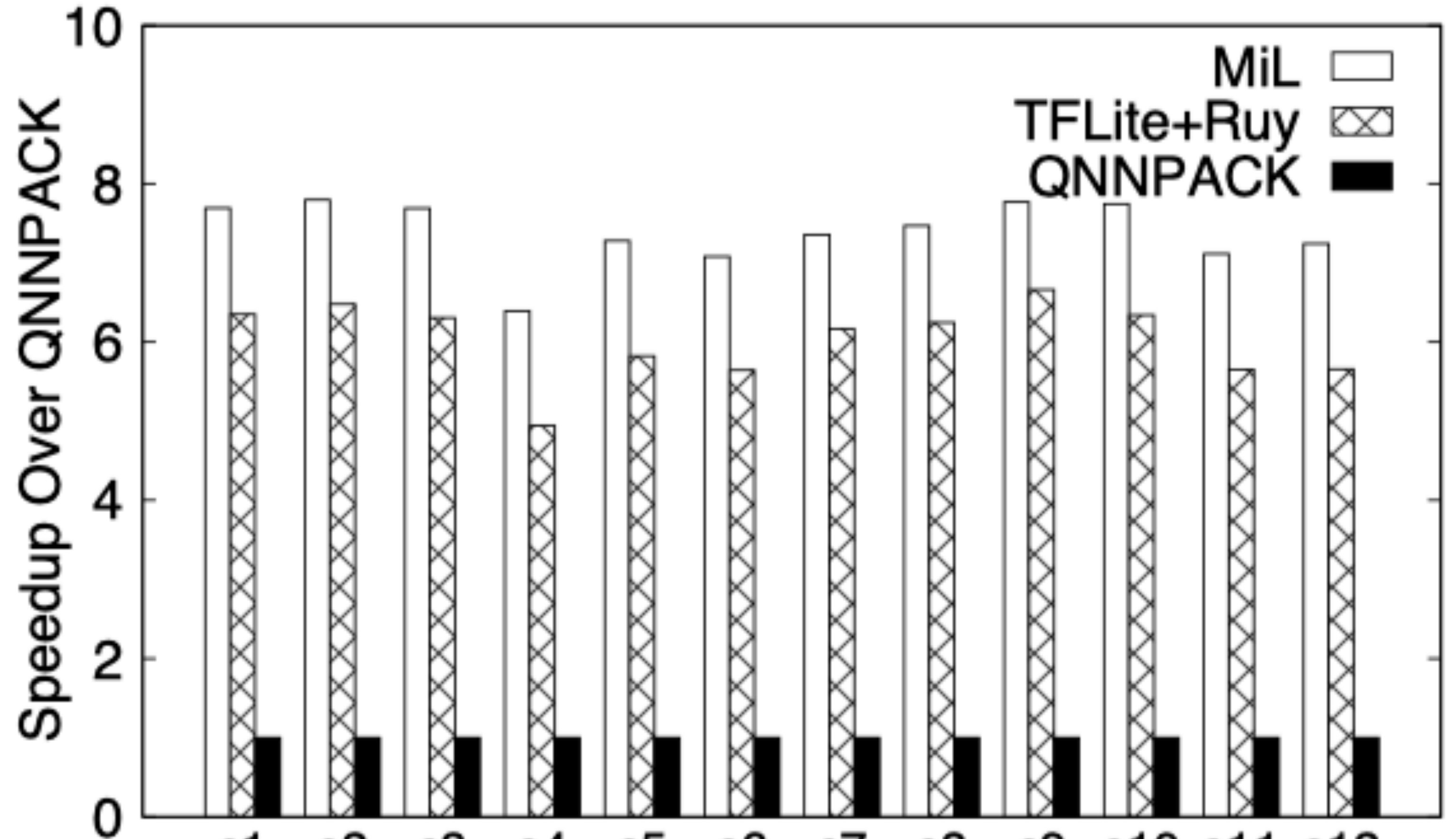


Name	Input $H_i \times W_i \times C_i$	Weight $C_o \times H_f \times W_f \times C_i,\ s$
c1	$27 \times 27 \times 64$	$192 \times 5 \times 5 \times 64,\ 2$
c2	$13 \times 13 \times 192$	$384 \times 3 \times 3 \times 192,\ 1$
c3	$13 \times 13 \times 384$	$256 \times 3 \times 3 \times 384,\ 1$
c4	$224 \times 224 \times 64$	$64 \times 7 \times 7 \times 64,\ 2$
c5	$112 \times 112 \times 64$	$128 \times 3 \times 3 \times 64,\ 1$
c6	$56 \times 56 \times 64$	$64 \times 3 \times 3 \times 64,\ 1$
c7	$56 \times 56 \times 128$	$128 \times 3 \times 3 \times 128,\ 2$
c8	$28 \times 28 \times 128$	$128 \times 3 \times 3 \times 128,\ 1$
c9	$28 \times 28 \times 256$	$256 \times 3 \times 3 \times 256,\ 2$
c10	$14 \times 14 \times 256$	$256 \times 3 \times 3 \times 256,\ 1$
c11	$8 \times 8 \times 512$	$1024 \times 3 \times 3 \times 512,\ 1$
c12	$7 \times 7 \times 512$	$512 \times 3 \times 3 \times 512,\ 1$

Experiments



(a) Compare with Bit-quantized Conv



(b) Compare with Byte-quantized Conv

Conclusions

- Byte-quantized model runs faster than bit-quantized one on ARM
- DotProd offers a new opportunity on ARM
- For byte-quantized models, MiL beats TFLite and QNNPack