DeepliteRT: Computer Vision at the Edge

Saad Ashfaq
saad@deeplite.ai
Alexander Hoffman
alexander.hoffman@deeplite.ai
Saptarshi Mitra
saptarshi@deeplite.ai
Sudhakar Sah
sudhakar@deeplite.ai
MohammadHossein AskariHemmat
mohammad@deeplite.ai
Ehsan Saboori
ehsan@deeplite.ai

Deeplite Inc.
Toronto, Canada

Abstract

The proliferation of edge devices has unlocked unprecedented opportunities for deep learning model deployment in computer vision applications. However, these complex models require considerable power, memory and compute resources that are typically not available on edge platforms. Ultra low-bit quantization presents an attractive solution to this problem by scaling down the model weights and activations from 32-bit to less than 8-bit. We implement highly optimized ultra low-bit convolution operators for ARM-based targets that outperform existing methods by up to 4.34×. Our operator is implemented within Deeplite Runtime (DeepliteRT), an end-to-end solution for the compilation, tuning, and inference of ultra low-bit models on ARM devices. Compiler passes in DeepliteRT automatically convert a fake-quantized model in full precision to a compact ultra low-bit representation, easing the process of quantized model deployment on commodity hardware. We analyze the performance of DeepliteRT on classification and detection models against optimized 32-bit floating-point, 8-bit integer, and 2-bit baselines, achieving significant speedups of up to 2.20×, 2.33× and 2.17×, respectively.

1 Introduction

Deep learning models for computer vision are being extensively deployed in various domains and industries due to substantial improvements in the accuracy of deep convolutional neural networks (CNNs). CNN architectures including VGG [28], ResNet [18], Inception [29], DenseNet [20] and YOLO [27] have demonstrated exceptional performance on image classification and object detection tasks. The widespread adoption of deep learning solutions in computer vision has also coincided with the growth of edge computing [33], promising the potential of bringing machine learning to low-power edge devices. However, the enhancements in CNN model accuracy have come at the expense of increased model complexity leading to high power, compute, memory, and storage requirements, making such models highly impractical for most use cases on resource-constrained edge devices.

Several compression techniques [4] [15] [19] have been explored to tackle this problem with the goal of decreasing model size while maintaining the baseline accuracy. Quantization is one such approach that realizes this goal by reducing the scale of model weights and activations from 32-bit floating-point (FP32) to lower precision representations. In addition to model compression, quantization also offers the benefits of fewer memory accesses, lower latency, and improved energy efficiency. 8-bit integer (INT8) has become the predominant bit-width for quantization and is widely supported in publicly available machine learning frameworks [1] [25] that perform quantization-aware training (QAT) and in open-source inference engines [10] [16] that execute the quantized models on commodity hardware. Recent advances have also been made in ultra low-bit quantization where the model weights and activations are quantized to less than 8 bits of precision. Using methods such as LSQ [11], a 2-bit quantized model can achieve a compression rate of up to 16× with an accuracy drop of less than a few percent relative to the FP32 baseline. Moreover, compute-intensive nodes in the network, including dense and convolution layers, can also utilize inexpensive bitwise operations to perform the dot products on extremely low-bit data. The significant compression and speedup resulting from ultra low-bit quantization make it a compelling choice for CNN deployment on edge devices.

Deep learning workloads on CPU architectures in commodity off-the-shelf edge devices generally utilize Single Instruction, Multiple Data (SIMD) hardware units to perform operations on multiple inputs in parallel. INT8 inference can be easily performed as 8-bit SIMD instructions are available in the instruction set architectures (ISAs) of mainstream CPUs. On the other hand, ultra low-precision models necessitate operations on sub-8-bit data requiring custom kernel implementations since SIMD execution is generally unsupported on less than 8 bits. Moreover, the weights and activations are "fake-quantized" during the forward and backward passes of QAT. This means that the input values are rounded to a discrete set of floating-point values and all computations are still performed in full-precision during the training phase. In the case of INT8 quantization, model weights and activations in FP32 can be easily cast to standard 8-bit integer when exporting the quantized model for inference. However, for ultra low-bit quantization, the conversion to extremely low-precision can not be performed at this stage due to lack of support for sub-8-bit data types on the target platform. Typically, the machine learning framework used for training inserts custom operators for quantized layers such as convolution during model export after QAT. The inference engine then needs to parse these custom operators when loading the model, lower them to the corresponding ultra low-bit kernels based on the quantized layer, and pack the fake-quantized inputs in ultra low-bit data structures. These modifications required in both the training and the inference paths make it extremely challenging to deploy ultra low-bit models on real commodity hardware.

To address these shortcomings in ultra low-bit pipelines, we introduce Deeplite Runtime (DeepliteRT), an end-to-end inference solution based on the TVM machine learning compiler stack [7], that offers state-of-the-art performance and framework-agnostic deployment of ultra low-bit models on ARM CPUs. We implement an ultra low-bit convolution operator that improves upon the performance of the TVM bit-serial kernel [8] [9] by up to 4.34×. We provide 32-bit ARMv7 and 64-bit ARMv8 bit-serial kernels making ultra low-bit CNN inference possible on globally pervasive ARM-based edge devices. We define compiler passes to automatically convert standard convolution layers into ultra low-bit operators and to efficiently pack full-precision data into compact ultra low-bit representations. These passes

Table 1: 2-bit accuracy on ImageNet with different QAT methods [5] [32] [22] [6] [11].

Model	**Top-1 Accuracy@32-bit**	**Top-1 Accuracy@2-bit**				
		PACT (2018)	LQ-NET (2018)	QIL (2019)	PACT-SAWB (2019)	LSQ (2020)
ResNet18	70.5%	64.4%	65.2%	65.7%	67.0%	67.9%
ResNet50	76.9%	72.2%	71.5%		74.2%	74.6%

enable fake ultra low-bit quantized models trained with various ML frameworks to be executed on ARM CPUs without any additional changes in the training and inference paths. With support for mixed precision inference, layers in the network that are sensitive to quantization can be kept at higher precision (FP32, INT8, etc.) while insensitive layers can be reduced to ultra low-bit in order to minimize the accuracy drop resulting from quantizing all layers in the model. To summarize, this paper makes the following contributions:

- We implement high performance bit-serial convolution kernels that achieve a speedup of up to 4.34× over existing ultra low-bit methods on ARM-based platforms.
- We present DeepliteRT, a compiler and runtime package for ultra low-bit inference on ARM CPUs. DeepliteRT automates the process of converting fake-quantized convolution layers from different machine learning frameworks used for quantization-aware training into ultra low-bit convolution kernels. Quantized models can be exported with the weights and activations still in full-precision without the need for custom operator definitions as compiler passes in DeepliteRT can handle the necessary casting, layout transforms and operator conversions during compilation. DeepliteRT provides a framework-agnostic end-to-end solution for ultra low-bit CNN deployment on edge devices eliminating the need to modify any code in the inference or runtime path.
- We perform a comprehensive evaluation of DeepliteRT on classification and detection models for both ARMv7 and ARMv8 targets, achieving significant performance improvements of up to 2.20×, 2.33× and 2.17× over highly optimized FP32, INT8 and ultra low-bit baselines, respectively.

2 Related Work

2.1 Ultra Low-bit Quantization

Quantization methods can be broadly categorized into uniform and non-uniform as well as quantization-aware training (QAT) and post-training quantization (PTQ). Uniform quantization refers to the case where the floating-point weights are quantized to integer values with a linear scaling from the integer to floating-point domain. The benefit of these methods is that operations can be performed in the integer domain and quickly converted to the floating-point domain via multiplication of a scaling factor. Non-uniform quantization removes this restriction, allowing for more flexibility in the mapping from floating-point to integer data.

QAT quantizes weights and activations while training the model to better simulate the model's performance after quantized deployment. PTQ methods train a full-precision model without regard for quantization, and then quantize the model with minimal access to the training dataset. State-of-the-art ultra low-bit quantization methods, shown in Table 1, make use of QAT to offset the loss of precision when reducing precision to less than 8 bits. LSQ

[11] is a simple yet effective quantization method which takes advantage of both uniform quantization and QAT to quantize models to as low as 2 bits with minimal accuracy degradation. For example, ResNet18 quantized to 2 bits with LSQ only incurs a 2.4% drop in accuracy relative to full-precision, but offers 16× compression per quantized layer.

2.2 Ultra Low-bit Inference

Most previous works on sub-8-bit inference on CPU architectures utilize the bit-serial method [8] [9] for dot product computation. Considering binary vectors with unipolar (unsigned) encoding where each input value is either 0 or 1, the bit-serial dot product is given by Eq. (1a). A bit-wise AND operation gives the element-wise product of the binary inputs and the popcount operation, that counts the number of bits set to 1, performs the accumulation. The binary case can easily be extended to larger bit-widths by slicing the inputs into binary vectors and performing a summation of the bit-serial dot products over all possible bit-sliced combinations. The corresponding equation for an M-bit weight and an N-bit activation vector is given in Eq. (1b) where operations are performed across bit-planes (w_m and a_n).

$$\vec{w} \cdot \vec{a} = popcount(\vec{w} \ \& \ \vec{a}) \tag{1a}$$

$$\vec{w} \cdot \vec{a} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} (popcount(\vec{w_m} \ \& \ \vec{a_n})) << (n+m) \tag{1b}$$

This bit-serial approach is implemented within TVM for dense and convolution layers in [8] and [9] with an average speedup of 1.9× for a 2-bit ResNet18 network over an optimized FP32 baseline on the ARM Cortex-A53 CPU in the Raspberry Pi 3B. Riptide [13] also uses the bit-serial kernels in TVM along with fusion, vectorization and tiling optimizations for binary networks to achieve considerable latency improvements over full-precision models on the Cortex-A53. Bitflow [17] presents another bit-serial implementation of a binary VGG network for Intel CPUs that is even faster than the corresponding full-precision CNN tested on a high-performance GPU. There have also been initiatives in this space that are not based on the bit-serial method including ULPPACK [30], BiQGEMM [21] and DeepGEMM [14].

3 Bit-serial Convolution

3.1 Bitpacking

Binary quantization approaches [26] can result in an unacceptable accuracy loss due to the use of a single bit for weight and activation values. To counter this, the bit-serial method can be extended to multiple bits by slicing the input weights and activation into separate bitplanes depending on the bit-width. This is illustrated in Fig. 1 for the 2A2W configuration (2 bits for activations and 2 bits for weights). Each value in the input data is first broken down into its constituent bits, creating bitplanes at every bit position. A bitplane holds the corresponding bit from different input values; for instance, bitplane 0 for weights stores the least significant bits across the weight values. Bitplanes can be compactly stored into standard data types such as 8-bit unsigned integers through the process of bitpacking. Assuming unipolar encoding for the 2-bit weights and activations, the bit-serial dot product can then be computed using Eq. (1b) producing the same result as a standard dot product as shown in Fig. 1. Based on our experiments, the bitpacking operation is not a major bottleneck consuming only 2-4% of the overall execution time in the bit-serial computation.

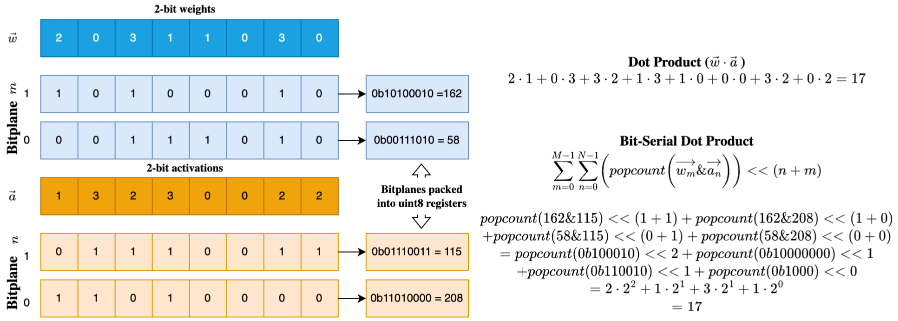

2-bit weights
$\vec{w}$
2 0 3 1 1 0 3 0
Bitplane m
1
1 0 1 0 0 0 1 0
0b10100010 =162
0
0 0 1 1 1 0 1 0
0b00111010 = 58
2-bit activations
$\vec{a}$
1 3 2 3 0 0 2 2
Bitplanes packed into uint8 registers
Bitplane n
1
0 1 1 1 0 0 1 1
0b01110011 = 115
0
1 1 0 1 0 0 0 0
0b11010000 = 208
Dot Product ($\vec{w} \cdot \vec{a}$)
$2 \cdot 1 + 0 \cdot 3 + 3 \cdot 2 + 1 \cdot 3 + 1 \cdot 0 + 0 \cdot 0 + 3 \cdot 2 + 0 \cdot 2 = 17$
Bit-Serial Dot Product
$\sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \left(popcount \left(\overrightarrow{w_m} \& \overrightarrow{a_n} \right) \right) << (n+m)$
$popcount(162\&115) << (1+1) + popcount(162\&208) << (1+0)$
$+popcount(58\&115) << (0+1) + popcount(58\&208) << (0+0)$
$= popcount(0b100010) << 2 + popcount(0b10000000) << 1$
$+popcount(0b110010) << 1 + popcount(0b1000) << 0$
$= 2 \cdot 2^2 + 1 \cdot 2^1 + 3 \cdot 2^1 + 1 \cdot 2^0$
$= 17$

Figure 1: Input weight and activation values are sliced into bitplanes and bitpacked within unsigned 8-bit integers enabling dot product calculation using bitwise operations.

3.2 Optimized bit-serial dot product

Eq. (1b) assumes a unipolar encoding scheme with unsigned values for both weights and activations. Recent works [11] [6] typically employ a hybrid unipolar-bipolar scheme with unipolar activations and bipolar (signed) weights producing quantized models with higher accuracy. The `nn.bitserial_conv2d` operator in TVM implements a convolution kernel for this hybrid scheme that calculates the bit-serial dot product as shown in Eq. (2), providing an open-source SOTA baseline for comparison with our work.

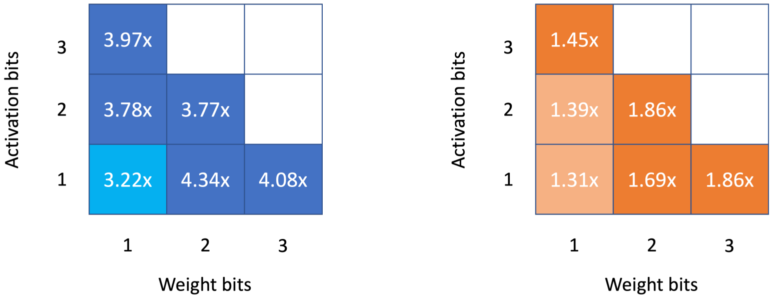
$$\vec{w} \cdot \vec{a} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} (popcount(\vec{w_m} \;\&\; \vec{a_n}) - popcount(\neg \vec{w_m} \;\&\; \vec{a_n})) << (n+m) \tag{2}$$

Compared to the purely unipolar case in Eq. (1b), this version doubles the number of popcount instructions adding considerable latency to the dot product calculations. Moreover, the weights can not take on the value 0 since this bipolar scheme distributes the quantization levels around 0. For example, in the case of 2 bits, each weight value will lie in the discrete set {-3, -1, 1, 3}. Such a representation introduces error when quantizing zero values, which is particularly harmful for common operations such as zero-padding and ReLU [24].

To address these drawbacks, we propose a novel bit-serial computation method in Eq. (3) for the hybrid scheme. Our approach reduces the number of popcount operations per dot product to one. It also requires the same number of overall instructions as the unipolar variant except for the most significant weight bit which has a slight overhead due to a constant multiplication. Our scheme also enables zero mapping of the signed weight values. For instance, 2-bit weights now fall in the set {-2, -1, 0, 1} providing compatibility with high accuracy quantization techniques such as LSQ that require zero mapping for the weights. This bit-serial dot product is the building block of our bit-serial convolution operator `dlrt_bitserial_conv2d`. With optimizations in kernel and data vectorization, loop reordering, and parallelization, `dlrt_bitserial_conv2d` achieves substantial performance uplifts over TVM's `nn.bitserial_conv2d` as shown in Fig. 2.

$$\vec{w} \cdot \vec{a} = \begin{cases} -1 \times \sum_{n=0}^{N-1} (popcount(\vec{w}_{M-1} \;\&\; \vec{a_n})) << (n+m), & \text{if } m = M-1 \\ \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} (popcount(\vec{w_m} \;\&\; \vec{a_n})) << (n+m), & \text{otherwise} \end{cases} \tag{3}$$

As opposed to the `nn.bitserial_conv2d` kernel that is only defined for ARMv7,



(a) Speedup on second layer of ResNet18 across different bit-widths.

(b) Speedup on ResNet18 model across different bit-widths.

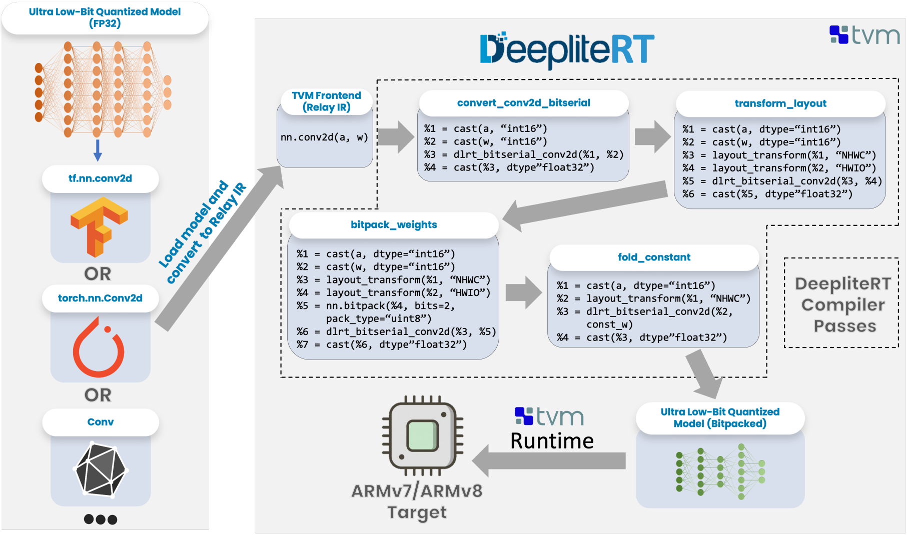
Figure 2: Operator level and end-to-end speedups of `dlrt_bitserial_conv2d` over TVM's `nn.bitserial_conv2d` on the Raspberry Pi 4B running in 32-bit mode.

we implement both 32-bit and 64-bit `dlrt_bitserial_conv2d` kernels enabling deployment on a broader range of 32-bit ARMv7 and 64-bit ARMv8 platforms.

4 DeepliteRT

Machine learning frameworks used for ultra low-precision QAT such as PyTorch [25] and TensorFlow [1] produce quantized models with extra operators relative to the full-precision network to handle the quantization and dequantization of model weights and activations. Assuming uniform quantization, these operators including addition, subtraction, division, multiplication, clipping and rounding are generally used to convert the floating-point data to integer before quantized layers and integer data back to floating-point after quantized layers. Inference engines such as ONNX Runtime [10] offer native support for these operators as they act on standard data types (FP32, INT16, INT8, etc.). However, quantized nodes such as convolution and dense layers are typically fake-quantized during QAT, restricting the weights and activations to a discrete set but still storing them in FP32. To realize ultra low-bit deployment on target hardware, custom operators and attributes for these layers have to be added by the ML framework which need to be then parsed and lowered to corresponding low-level kernels by the inference engine. These modifications in the ML and runtime frameworks require some level of expertise in both training and inference domains. Moreover, the changes made for one ML framework are not portable to a different framework, making quantized ultra low-bit model deployment inaccessible to most practitioners.

DeepliteRT is an inference solution that defines custom compiler passes in the TVM machine learning compiler to transform fake-quantized models into compact ultra low-precision networks. ML practitioners can perform QAT in any framework of choice and simply compile the resulting fake-quantized model with DeepliteRT for easy deployment on ARM-based targets with TVM runtime. DeepliteRT includes our optimized bit-serial convolution operator detailed in Section 3.2. It also supports mixed precison deployment allowing quantization-sensitive layers to be kept at higher precision and insensitive layers at ultra low-precision. With high-level APIs in both Python and C++, DeepliteRT can be easily integrated in applications on edge devices for quantized model compilation, tuning and inference.



```
nn.conv2d(a, w)
```

```
%1 = cast(a, “int16”)
%2 = cast(w, “int16”)
%3 = dlrt_bitserial_conv2d(%1, %2)
%4 = cast(%3, dtype”float32”)
```

```
%1 = cast(a, dtype=“int16”)
%2 = cast(w, dtype=“int16”)
%3 = layout_transform(%1, “NHWC”)
%4 = layout_transform(%2, “HWIO”)
%5 = dlrt_bitserial_conv2d(%3, %4)
%6 = cast(%5, dtype”float32”)
```

```
%1 = cast(a, dtype=“int16”)
%2 = cast(w, dtype=“int16”)
%3 = layout_transform(%1, “NHWC”)
%4 = layout_transform(%2, “HWIO”)
%5 = nn.bitpack(%4, bits=2,
     pack_type=“uint8”)
%6 = dlrt_bitserial_conv2d(%3, %5)
%7 = cast(%6, dtype”float32”)
```

```
%1 = cast(a, dtype=“int16”)
%2 = layout_transform(%1, “NHWC”)
%3 = dlrt_bitserial_conv2d(%2,
     const_w)
%4 = cast(%3, dtype”float32”)
```

Figure 3: DeepliteRT converts fake-quantized convolution layers from models in different formats to optimized ultra low-bit convolution operators through a series of compiler passes. The passes replace `nn.conv2d` with `dlrt_bitserial_conv2d`, bitpack the weights in ultra low-bit, and cast and transform the layouts of data as required. The resulting compiled model can be deployed on ARMv7 and ARMv8 CPUs via TVM runtime.

4.1 Compiler passes

`nn.conv2d` is the operator for 2D convolution in TVM's Relay IR. Convolution layers from models trained with different ML frameworks are internally converted into `nn.conv2d` by the appropriate frontend. For instance, `tf.nn.conv2d` from a TensorFlow model, `torch.nn.Conv2d` from a PyTorch model and `Conv` from an ONNX model are all translated to `nn.conv2d`. We define a sequence of compiler passes in DeepliteRT to convert a fake quantized convolution layer represented by `nn.conv2d` in Relay IR into our optimized bit-serial convolution operator `dlrt_bitserial_conv2d` as shown in Fig. 3.

convert_conv2d_bitserial: This custom pass converts `nn.conv2d` nodes for quantized layers into `dlrt_bitserial_conv2d` nodes in the IR. It also casts the input weights and activations into integer and the resulting convolution output back to floating-point.

transform_layout: This pass is invoked to change the layout for activations to NHWC and the layout for weights to HWIO as required by the low-level `dlrt_bitserial_conv2d` kernel. The transformation is only performed if the activations and/or weights are not already in the required layouts.

bitpack_weights: This custom pass adds `nn.bitpack` operators in the Relay IR for the bitpacking of weights during compilation prior to bit-serial convolution. The bitpacking of activations is handled by the `dlrt_bitserial_conv2d` operator during inference since the activation values are not available offline.

fold_constant: This pass is used to perform all the computations on weights during compilation as they are compile-time constants. The result of casting the weights to integer, transforming their layout and bitpacking them is then simply passed as a constant to the `dlrt_bitserial_conv2d` operator.

Table 2: End-to-end latencies (ms) and speedups of DeepliteRT 2A2W over TVM FP32, ONNX Runtime INT8 and TVM bit-serial 2A2W baselines.

Model	**Raspberry Pi 4B - 32-bit ARMv7**				**Raspberry Pi 4B - 64-bit ARMv8**			
	FP32	INT8	2A2W	2A2W (Ours)	FP32	INT8	2A2W	2A2W (Ours)
ResNet18	149.29	145.44	130.92	70.32	110.94	91.13	123.28	67.13
ResNet50	433.19	326.49	311.8	196.79	315.03	203.56	295.96	197.91
ResNet101	-	558.47	487.96	325.37	545.01	378.27	471.71	319.09
VGG19	-	1399	1003	654.69	-	922.28	962.65	636.79
InceptionV3	312.82	245.16	357.77	165.05	218.18	151.55	340.82	164.62
DenseNet121	387.98	589.03	296.27	252.65	302.50	261.94	269.91	227.05
VGG16-SSD300	1671	2310	1780	1190	1547	1462	1631	1060
YOLOv5s	219.72	197.27	135.64	100.32	169.93	113.5	130.03	97.49
Average speedup	1.89×	1.91×	1.58×	-	1.54×	1.20×	1.56×	-
Minimum speedup	1.40×	1.49×	1.17×	-	1.32×	0.92×	1.19×	-
Maximum speedup	2.20×	2.33×	2.17×	-	1.71×	1.45×	2.07×	-

4.2 Mixed precision support

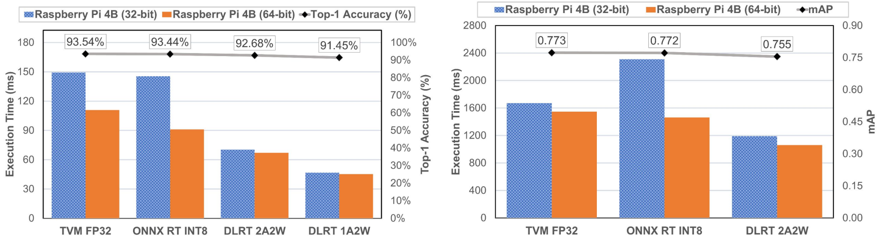
In the default case, DeepliteRT converts all convolution layers except the first to bit-serial operators using the specified bit-width. However, quantizing all the layers to ultra low-bit can result in severe accuracy degradation. This can be countered with mixed precision quantization by choosing different precisions across layers using methods such as HAWQ-V3 [31] for accuracy preservation. DeepliteRT provides mixed precision inference by accepting a configuration file as input that specifies the quantization parameters per layer including activation bit-width, weight bit-width and encoding scheme. This per-layer information is passed to the **convert_conv2d_bitserial** pass to selectively offload convolution layers to ultra low-bit with the provided bit-widths and keep other layers in full-precision as required.

5 Evaluation

We evaluate classification and detection models on a Rasbperry Pi 4B (4×ARM Cortex-A72@1.8GHZ) device with 32-bit and 64-bit operating systems to enable ARMv7 and ARMv8 execution. We select TVM FP32 for the full-precision baseline as it significantly outperformed FP32 kernels in ONNX Runtime and TensorFlow Lite [16] in our experiments. TVM does not offer an optimized INT8 operator so we choose ONNX Runtime for INT8 experiments due to its high performance 8-bit kernels. Finally, we use the TVM 2A2W configuration based on the `nn.bitserial_conv2d` operator for ultra low-bit experiments; we also port this operator to ARMv8 to establish the 64-bit 2A2W baseline. All models deployed with TVM and DeepliteRT were tuned using AutoTVM [3] with 1500 trials.

5.1 End-to-end performance

Table 2 reports the end-to-end latencies and speedups for classification and detection models. The average, minimum and maximum numbers represent the speedups realized with DeepliteRT over the TVM FP32, ONNX Runtime INT8 or TVM 2A2W results in the same column. Some results for ResNet101 and VGG19 at FP32 are missing in the table as the device runs out of memory when loading full-precision model parameters. Interestingly, even though the TVM 2A2W configuration offers similar level of performance in 32-bit and 64-bit modes, it does not remain competitive in the latter case due to substantial performance



(a) ResNet18 on VWW.

(b) VGG16-SSD300 on VOC.

Figure 4: Trade off between ultra low-bit model accuracy and performance.

uplifts for the FP32 and INT8 baselines with the ARMv8 ISA. In contrast, DeepliteRT offers leading performance for both ARMv7 and ARMv8 targets. On average, DeepliteRT realizes speedups of 1.89×, 1.91× and 1.58× in 32-bit mode and 1.54×, 1.20× and 1.56× in 64-bit mode over TVM FP32, ONNX Runtime INT8 and TVM 2A2W, respectively.

Table 3: DeepliteRT latency (ms) on ResNet50 with mixed precision configurations.

52 FP32	**26 FP32 + 26 2A2W**	**52 2A2W**	**26 2A2W + 26 1A2W**	**52 1A2W**
433.19	314.69	196.79	180.37	134.26

5.2 Model accuracy and mixed precision

SOTA for ultra low-bit quantization has progressed at a rapid pace as shown in Table 1. We study the accuracy-performance tradeoff of ultra low-bit quantization using LSQ for a classification and detection model in Fig. 4. ResNet18 trained on the VWW dataset [7] only incurs accuracy drops of 0.86% and 2.09% relative to the FP32 baseline with performance uplifts of up to 2.12× and 3.19× at 2A2W and 1A2W, respectively. Similarly, VGG16-SSD300 [23] trained on the VOC dataset [12] only sees a 0.18 loss in mAP at 2A2W while realizing a speedup of up to 1.46×. The minor accuracy dips, substantial latency improvements and huge savings in model size make ultra low-bit networks an ideal fit for edge deployment. Moreover, mixed precision inference with DeepliteRT enables practitioners to easily explore this tradeoff between accuracy and performance, as illustrated in Table 3 for ResNet50, by varying the number of layers in FP32, 2A2W and 1A2W. An appropriate quantization configuration can be chosen based on model accuracy and latency measurements from the target.

6 Conclusion

We present an end-to-end inference solution in DeepliteRT for ML framework-agnostic deployment of ultra low-bit quantized models on 32-bit ARMv7 and 64-bit ARMv8 platforms. It implements compiler passes for the automatic conversion of fake-quantized networks in full-precision to compact representations in ultra low-bit, eliminating the need for custom modifications in the training and runtime components to enable inference at ultra low-precision. Using high-performance bit-serial convolution kernels, DeepliteRT outperforms highly optimized floating-point, integer, and ultra low-bit baselines on image classification and object detection models by up to 2.20×, 2.33× and 2.17×, respectively.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018. URL http://arxiv.org/abs/1802.04799.

[3] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *CoRR*, abs/1805.08166, 2018. URL http://arxiv.org/abs/1805.08166.

[4] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *CoRR*, abs/1710.09282, 2017. URL http://arxiv.org/abs/1710.09282.

[5] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. PACT: parameterized clipping activation for quantized neural networks. *CoRR*, abs/1805.06085, 2018. URL http://arxiv.org/abs/1805.06085.

[6] Jungwook Choi, Swagath Venkataramani, Vijayalakshmi (Viji) Srinivasan, Kailash Gopalakrishnan, Zhuo Wang, and Pierce Chuang. Accurate and efficient 2-bit quantized neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 348–359, 2019. URL https://proceedings.mlsys.org/paper_files/paper/2019/file/006f52e9102a8d3be2fe5614f42ba989-Paper.pdf.

[7] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. Visual wake words dataset, 2019.

[8] Meghan Cowan, Thierry Moreau, Tianqi Chen, and Luis Ceze. Automating generation of low precision deep learning operators. *CoRR*, abs/1810.11066, 2018. URL http://arxiv.org/abs/1810.11066.

[9] Meghan Cowan, Thierry Moreau, Tianqi Chen, James Bornholt, and Luis Ceze. Automatic generation of high-performance quantized machine learning kernels. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 305–316, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370479. doi: 10.1145/3368826.3377912. URL https://doi.org/10.1145/3368826.3377912.

[10] ONNX Runtime developers. Onnx runtime. https://onnxruntime.ai/, 2021. Version: 1.15.0.

[11] Steven K. Esser, Jeffrey L. McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S. Modha. Learned step size quantization. *CoRR*, abs/1902.08153, 2019. URL http://arxiv.org/abs/1902.08153.

[12] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes challenge: A retrospective. *International Journal of Computer Vision*, 111(1):98–136, January 2015.

[13] Joshua Fromm, Meghan Cowan, Matthai Philipose, Luis Ceze, and Shwetak Patel. Riptide: Fast end-to-end binarized neural networks. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 379–389, 2020. URL https://proceedings.mlsys.org/paper_files/paper/2020/file/2a79ea27c279e471f4d180b08d62b00a-Paper.pdf.

[14] Darshan C. Ganji, Saad Ashfaq, Ehsan Saboori, Sudhakar Sah, Saptarshi Mitra, MohammadHossein AskariHemmat, Alexander Hoffman, Ahmed Hassanien, and Mathieu Léonardon. Deepgemm: Accelerated ultra low-precision inference on cpu architectures using lookup tables. *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2023. URL https://arxiv.org/abs/2304.09049.

[15] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *CoRR*, abs/2103.13630, 2021. URL https://arxiv.org/abs/2103.13630.

[16] Google. Tensorflow lite. https://www.tensorflow.org/lite, 2022. Version: 1.15.0.

[17] Anton Gulenko, Alexander Acker, Florian Schmidt, Sören Becker, and Odej Kao. Bitflow: An in situ stream processing framework. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, pages 182–187, 2020. doi: 10.1109/ACSOS-C51401.2020.00053.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

[19] Yang He and Lingao Xiao. Structured pruning for deep convolutional neural networks: A survey, 2023. URL https://arxiv.org/abs/2303.00566.

[20] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016. URL http://arxiv.org/abs/1608.06993.

[21] Yongkweon Jeon, Baeseong Park, Se Jung Kwon, Byeongwook Kim, Jeongin Yun, and Dongsoo Lee. Biqgemm: Matrix multiplication with lookup table for binary-coding-based quantized dnns. *CoRR*, abs/2005.09904, 2020. URL https://arxiv.org/abs/2005.09904.

[22] Sangil Jung, Changyong Son, Seohyung Lee, JinWoo Son, Youngjun Kwak, Jae-Joon Han, and Changkyu Choi. Joint training of low-precision neural network with quantization interval parameters. *CoRR*, abs/1808.05779, 2018. URL http://arxiv.org/abs/1808.05779.

[23] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single shot MultiBox detector. In *Computer Vision – ECCV 2016*, pages 21–37. Springer International Publishing, 2016. doi: 10.1007/978-3-319-46448-0_2. URL http://arxiv.org/abs/1512.02325.

[24] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. *CoRR*, abs/2106.08295, 2021. URL https://arxiv.org/abs/2106.08295.

[25] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learni
pdf.

[26] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. Binary neural networks: A survey. *CoRR*, abs/2004.03333, 2020. URL https://arxiv.org/abs/2004.03333.

[27] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018. URL http://arxiv.org/abs/1804.02767.

[28] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[29] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015. URL http://arxiv.org/abs/1512.00567.

[30] Jaeyeon Won, Jeyeon Si, Sam Son, Tae Jun Ham, and Jae W. Lee. Ulppack: Fast sub-8-bit matrix multiply on commodity simd hardware. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 52–63, 2022. URL https://proceedings.mlsys.org/paper_files/paper/2022/file/14bfa6bb14875e45bba028a21ed38046-Paper.pdf.

[31] Zhewei Yao, Zhen Dong, Zhangcheng Zheng, Amir Gholami, Jiali Yu, Eric Tan, Leyuan Wang, Qijing Huang, Yida Wang, Michael W. Mahoney, and Kurt Keutzer. HAWQV3: dyadic neural network quantization. *CoRR*, abs/2011.10680, 2020. URL https://arxiv.org/abs/2011.10680.

[32] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. *CoRR*, abs/1807.10029, 2018. URL http://arxiv.org/abs/1807.10029.

[33] Guangxu Zhu, Dongzhu Liu, Yuqing Du, Changsheng You, Jun Zhang, and Kaibin Huang. Toward an intelligent edge: Wireless communication meets machine learning. *IEEE Communications Magazine*, 58(1):19–25, 2020. doi: 10.1109/MCOM.001.1900103.