BLUNET: ARITHMETIC-FREE INFERENCE WITH BIT-SERIALISED TABLE LOOKUP OPERATION FOR EFFICIENT DEEP NEURAL NETWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Deep neural networks (DNNs) are both computation and memory intensive. Large amounts of costly arithmetic multiply-accumulate (MAC) operations and data movement hinder its application to edge AI where DNN models are required to run on energy-constrained platforms. Table lookup operations have potential advantages over traditional arithmetic multiplication and addition operations in terms of both energy consumption and latency in hardware implementations for DNN design. Moreover, the integration of weights into the table lookup operation eliminates costly weight movements. However, the challenge of using table lookups is in scaling. In particular, the size and lookup times of tables grow exponentially with the fan-in of the tables. In this paper, we propose BLUnet, a table lookup-based DNN model with bit-serialized input to overcome this challenge. Using binarized time series inputs, we successfully solve the fan-in issue of lookup tables. BLUnet not only achieves high efficiency but also the same accuracies as MAC-based neural networks. We experimented with popular models in computer vision applications to confirm this. Our experimental results show that compared to MAC-based baseline designs as well as the state-of-the-art solutions, BLUnet achieves orders of magnitude improvement in energy efficiencies.

1 INTRODUCTION

The success of deep neural networks (DNN) comes at a high cost of computation and storage in computer systems. In particular, it remains a challenge to deploy them in use-case scenarios where computing resources and the energy budget are limited, such as edge AI and the artificial intelligence of things (AIoT). As a result, many optimization methods have been proposed to realize efficient inference, such as *model quantization*. With quantization, the precision of trained weights and activations of a DNN is reduced from full precision (i.e. 32-bit IEEE floating point) to significantly fewer bits. The state-of-the-art works have successfully compressed the weight and activation to 6 to 8 bits with negligible accuracy drop. Model quantization exploits redundancies in DNN models to reduce the size of the models significantly, thereby improving computation and storage efficiency.

Going beyond model-level optimization, another research direction examines the efficiency of the operations involved. AdderNet and *spiking neural networks* (SNN) are two efficient neural network designs, where only additions are used in the computations with costly multiplications eliminated. They are gaining prominence in low-power implementation of DNNs.

Table lookup operation is a standard operation in computing and its hardware counterpart, the *lookup table* (LUT), is the core logic unit in field-programmable gate arrays (FPGAs). When the fan-in is small, table lookup operations have significant advantages over traditional arithmetic multiplication and addition operations for quantized neural networks in terms of both energy consumption and latency in hardware implementation. Moreover, the integration of weights into the table lookup operation will eliminate costly weight movement. These two advantages make the table lookup operation an ideal substitute for the MAC operation in an efficient quantized DNN design. However, as the number of fan-in scales, the advantage of table lookup operation vanishes exponentially. In this work, we propose BLUnet, a table lookup-based quantized DNNs design with bit-serialized input. With the binary time series input, we successfully solve the fan-in issue of lookup tables and

achieve high energy efficiency in inference. We carried out an experiment with popular models in computer vision applications, analyzed the effects of different bit-width, and did a thorough study on the fan-in of LUT. Experimental results show that compared with the state-of-the-art efficient DNN designs, BLUnet achieves orders of magnitude improvement in energy consumption.

2 Related Work

An entire class of techniques generally known as model quantization has been used successfully to reduce the computation overhead, memory storage, and energy consumption of DNN models Ding et al. (2019); Tung et al. (2018); Hubara et al. (2016a;b); Courbariaux et al. (2015); Zhou et al. (2016); Wang et al. (2019c); Yang et al. (2019); Han et al. (2015). In model quantization, the precision of the weights and activations are reduced. However, radical model quantization such as *binarized neural networks* (BNN) achieves high computation efficiency, but suffers severe accuracy drop Hubara et al. (2016b); Lin et al. (2017). Thus, reducing bit widths while ensuring accuracy is an active research topic Zhou et al. (2016); Wang et al. (2019c); Yang et al. (2019); Guo et al. (2016). In addition, many works put forward model compression methods that directly consider the capabilities of the hardware Wang et al. (2019c); Yang et al. (2017). Wang et al. (2019c) explored quantization policies with a given set of energy constraints. The previous studies on model compression effectively reduce the computation cost and data movement by minimizing the bit width of weights and activation.

However, the weight access and costly multiplication are still unavoidable. More importantly, the model compression does not work well on already very compact models. To optimize already compressed models further, the efficiency of the basic operations of DNN were studied. One example is the design of DNNs that have only addition operations because the hardware efficiency of addition is much better than that of multiplication. AdderNet family is a typical exploration in this research direction Chen et al. (2020); You et al. (2020). In AdderNet design, massive multiplications are traded for much cheaper additions. Another class of neural networks that only involve additions is spiking neural networks (SNN) Tavanaei et al. (2019); Sengupta et al. (2019); Wu et al. (2019); Wang et al. (2021).

Table lookup operation is another efficient operation that can be utilized for optimizing DNN designs Umuroglu et al. (2020); Ramanathan et al. (2020); Wang et al. (2019b); Umuroglu et al. (2017). However, the efficiency of the table lookup operation vanishes exponentially with the scaling up of its fan-in. Umuroglu et al. (2020) proposed the co-design of neural networks and circuits. However, it can only support small and customized networks due to the fan-in limitation. Wang et al. (2019b); Umuroglu et al. (2017) use LUT in FPGA to help implement XNOR gate in BNN, which however suffered from severe accuracy drop. Different from previous work, in this paper, we propose BLUnet, a table lookup-based quantized DNNs design with bit-serialized input. With bit-serialised table lookup operation, BLUnet achieves highly efficient inference.

3 LUT-based DNN Design with Bit-serialized Input

In this section, we will first describe how to replace arithmetic MAC operations with table lookup operations, show the much higher efficiency advantage of table lookup operations over MAC operations in hardware implementation for quantized neural networks, and explain why controlling fan-in is the key to efficiency. Then we will propose a table lookup-based DNN design with Bit-serialized input/output to solve the fan-in issues.

3.1 Implementation of MAC in Quantized Models with Table Lookup Operation

We will start the description of BLUnet with the description of a specific LUT operation representing the computation of the following:

$$y = f(w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5 + b). \tag{1}$$

This is the typical computation of a neuron with six dendrites. Here w_i are the 32-bit weights, and b is the bias. Note that the weights and bias are constants for trained network models. For

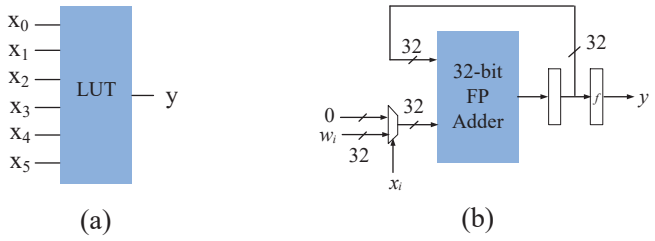

Figure 1: Two different ways of summing 6 inputs: (a) 6-input LUT and (b) 32-bit FP adder for a 6-input neuron with binarized input and output.

reasons that will be clear later, the 6-input LUT as in Fig. 1(a) uses one-bit activation x_i and an activation function f that will produce a one-bit output y. Because x_i and y are binary, a size 2^6 LUT suffices. Fig. 1(b) shows what would be needed to implement Eq. 1 using a full 32-bit adder in a serial manner. We can of course also implement Eq. 1 using a tree of full 32-bit adders but that would add to the resource cost. Table 1 shows the energy, latency, and area cost of both options. The comparison is conducted on the 45nm process node. The data for a 32-bit FP adder is obtained from Nathan et al. (2013). The data for a 6-input LUT is scaled from Abusultan & Khatri (2014). It is clear that the LUT implementation is orders of magnitude more efficient than one using arithmetic logic. In case the activations x_i and output y consist of multiple bits, the advantages disappear very quickly as the table will need to grow exponentially, albeit the computation can still theoretically be done. Using the LUT directly as described leads, however, to a binarized network that will perform poorly in terms of accuracy. In the next section, we will show how we can use Eq. 1 and the small 6-input LUT to build a complete DNN with bit-serialized input.

3.2 Table Lookup-based DNN Design with Bit-serialized Input/Output

To overcome the major obstacle of large fan-in, we propose the table lookup-based BLUnet design with serialized binary input/output. In the rest of this section, we first introduce how to build the neuron in the DNN model with table lookup operation at the bit level, i.e., the BLUnet neuron. Then, the construction of a DNN model with individual neurons is presented.

3.2.1 BLU Neuron Computation

To show how small fan-in table lookup operators are used in our proposed BLUnet, we shall use two figures with a simple example to demonstrate the computation of a neuron with and without timing information separately. Eq. 2 shows a normal computation for a neuron with four 2-bit activations a^i and one 2-bit output y. The 2-bit activation $a_i = a_i^1 a_i^0$ effectively represent $a_i^1 * 2^1 + a_i^0 * 2^0$. Negative a_i is represented by its two's complement code.

$$y = w_0(a_0^1 * 2^1 + a_0^0 * 2^0) + w_1(a_1^1 * 2^1 + a_1^0 * 2^0) + w_2(a_2^1 * 2^1 + a_2^0 * 2^0) + w_3(a_3^1 * 2^1 + a_3^0 * 2^0) \quad (2)$$

Next, we will use a table lookup operation to replace the computation in Eq. 2 to get an (approximate/quantized) two-bit results. Let's use a concrete example: assuming there are four binary activations, $a_0 = 11_2$, $a_1 = 10_2$, $a_2 = 01_2$, $a_3 = 01_2$, respectively, while the four 32-bit FP weights are $w_0 = 1.13$, $w_1 = 0.92$, $w_2 = 0.87$, and $w_3 = 0.23$ (see Fig. 2(b)). We then serialize the bits into binary time series inputs. The computation conducted in the LUTs at the first stage at the first time step is the multiplication of the first input bit and the corresponding weight w followed by an

Table 1: Costs of the 6-input neuron (with activation quantized to binary digit) implementation with six 32-bit float point addition VS 6-input LUT in terms of energy, latency, and area.

Name	Six 32-bit FP additions		6-input LUT
Energy	46.32 pJ	(9154x)	5.06 fJ
Latency	26.7 ns	(162x)	165.26ps
Area	4804 μm^2	(1855x)	2.59 μm^2

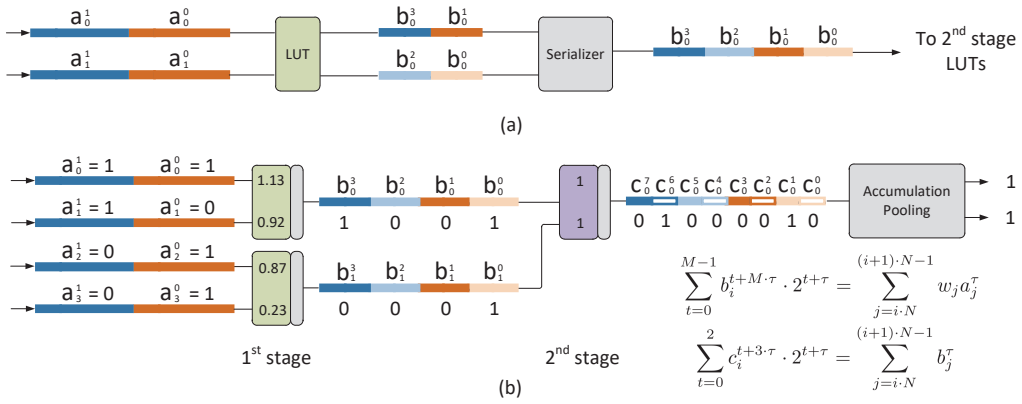

Figure 2: Computation of a simple 4 input neuron implemented by LUT-2 with timing information to show the timing relation

quantization function to quantize the result to a two-bit integer. For example:

$$b_0^1 \cdot 2^1 + b_0^0 \cdot 2^0 = f_q(w_0 a_0^0 + w_1 a_1^0) \tag{3}$$

where f_q is the quantization function that is integrated into the LUT. In the example, we have $\lfloor (1 \times 1.13) + (0 \times 0.92) \rfloor = \lfloor 1.13 \rfloor = 1 = 01_2$.

For the computation in the second time step, the computation is:

$$b_0^3 \cdot 2^2 + b_0^2 \cdot 2^1 = f_q(w_0 a_0^1 + w_1 a_1^1). \tag{4}$$

Again referring to Fig. 2(b), we have $\lfloor (1 \times 1.13) + (1 \times 0.92) \rfloor = \lfloor 2.05 \rfloor = 2 = 10_2$. This completes the result of the first stage of this LUT yielding the serialized results of $b_0 = 1001_2$. The relationship between a and b can be generalized into Eq. 5.

$$\sum_{t=0}^{M-1} b_i^{t+M \cdot \tau} \cdot 2^{t+\tau} = f_q(\sum_{j=i \cdot N}^{(i+1) \cdot N-1} w_j a_j^\tau) \tag{5}$$

Using the serialized output of the first stage, the computation in the LUTs of the second stage at first time step is:

$$c_0^1 \cdot 2^1 + c_0^0 \cdot 2^0 = (b_0^0 + b_1^0) \tag{6}$$

This is followed by:

$$c_0^3 \cdot 2^2 + c_0^2 \cdot 2^1 = (b_0^1 + b_1^1) \tag{7}$$

So the sum of $b_0 = 1001_2$ and $b_1 = 0001_2$ in the example would yield $(01|00|00|10)_2$. Note that the carry-in of each bit position is *not* incorporated in the next bit position but forms part of the serialized bit stream. Using our example, the first output pair of 10_2 in $(01|00|00|10)_2$ is effectively the carry and sum bits of that bit position.

The relationship between b and c can be generalized into Eq. 8.

$$\sum_{t=0}^{2} c_i^{t+3 \cdot \tau} \cdot 2^{t+\tau} = \sum_{j=i \cdot N}^{(i+1) \cdot N-1} b_j^\tau \tag{8}$$

Finally, the serialized result is sent into an 'accumulation pooling' layer (see below) in a pipeline manner to finally generate a quantized integer output for this neuron.

Fig. 3 shows a more generalized design with N^x inputs. The more inputs there are, the more stages BLUnet will have. Therefore, BLU neuron would have more computing stages than traditional neurons. However, thanks to the much smaller latency of table lookup operations than traditional arithmetic operations as shown in Table 1, the increased number of computing stage would not affect the latency efficiency of BLU neuron. The serialized data are sent into a single '*accumulation*

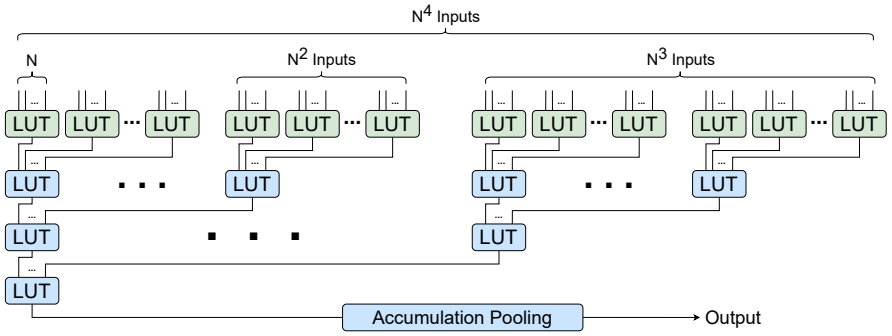

Figure 3: Computation of a n-input neuron implemented by LUT-6 without timing information to show the structure and connection of LUTs

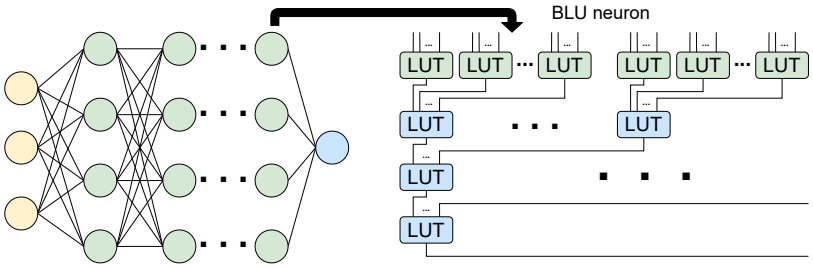

Figure 4: DNN model design with BLU neuron

pooling' layer in a pipelined manner to finally generate a quantized integer output for this neuron. The function of accumulation pooling layer is given by:

$$y = f_q(\sum^{i} c_0^i \cdot 2^{\lfloor i/M \rfloor + (i \bmod M)}) \tag{9}$$

where y is the multi-bit quantized output of the neuron and f_q is the quantization function, which can be expressed in Eq. 10, where β is the scaling factor necessary for reducing the input to f_q to the target bitwidth.

$$f_q(v) = \lfloor v/2^{\beta} \rfloor \tag{10}$$

3.2.2 DNN Model Design with BLU Neuron

Fig. 4 shows a multi-layer perceptron (MLP) style neural network and how to build the corresponding BLUnet with BLU neuron. We can see that as long as the model can be represented as a collection of neuron computations, the BLU neuron can be used to replace the normal neuron in the DNN models including *convolutional neural networks* (CNNs) and MLP to realize efficient inference. Therefore, BLUnet could be widely applied to any DNN model-based applications. To implement BLUnet for inference, we just need to pre-calculate the LUT outputs to all possible input bit combinations, given the weights from the model. This computation is straightforward because of the limited number of possibilities.

Due to the high efficiency of table lookup operations, BLUnet is well suited for hardware implementation of specific accelerators using application specific integrated circuits (ASIC). However, the aim of this paper is to introduce BLUnet and the overall efficient design idea. Specific accelerator design for BLUnet will have to consider the idiosyncrasies of the hardware platform chosen, such as hardware architecture, process node, etc., and specific hardware design optimizations will have to be made to obtain the best results, which is more suitable to hardware venues and beyond the scope of this paper. These issues are orthogonal to the work described here. Like other papers describing efficient neural network design such as AdderNet, we will use the number of operations along with their cost in terms of energy and latency as metrics to assess the efficiency of the model designs.

3.3 IMPLEMENTATION AND TRAINING FOR BLUNET

We implemented BLUnet in Pytorch for the accuracy results by rewriting the functions for the convolutional and linear layer. The traditional neuron layers, either `nn.Conv2d` or `nn.Linear`, will output a three dimensional tensor A, with shape (C, W, H), where C, W and H stand for the number of channels, width and height of the activation tensor. We extended them so that they can output a four-dimensional tensor A' whose shape is (C, W, H, L). where the new dimensional size L stands for the number of first-stage LUTs implemented during the computation. The relationship between the element $\alpha_{c,w,h}$ in tensors A and the element $\alpha'_{c,w,h,l}$ in tensor A' follows Eq. 11.

$$\alpha_{c,w,h} = \sum_{l}^{L} \alpha'_{c,w,h,l} \tag{11}$$

After obtaining tensor A' using the new function, we quantize each element in the tensor into M bits using a mask, and then calculate tensor A by Eq. 11. This is the output of each layer in BLUnet. It should be noted that the value of L equals the number of weights associated to $\alpha_{c,w,h}$ over N, the fan-in of the LUT. Please check Appendix for more implementation details.

Another advantage of BLUnet lies in its training procedure. Different from previous efficient design with non-traditional operators, such as AdderNet, which needs to retrain the model from scratch, BLUnet could utilize the pre-trained weights for conventional models due to the fact that BLUnet based model design has a similar model structure with the conventional model. To ease the training effort, in this work, we adopt weights from the pre-training model from a package in Pytorch `TORCHVISION.MODELS`. We then do gradient decent with a 1/10 learning rate for extra 20 epochs. Experiment setting is detailed in Section 4. Code would be made open source after the acceptance.

4 EXPERIMENTS

In this section, we verify the effectiveness of BLUnet on various kinds of models and compare it with the traditional model design. We also compare our work with state-of-the-art work to further show its advantages. Then, we explore the relation between LUT type and the energy, latency, and area of BLUnet. At last, we compare the results of a BLUnet prototype on FPGA with other low energy solutions on AISC and FPGA to show the advantage of BLUnet for extremely low energy applications.

4.1 EXPERIMENT SETUP

We built BLUnet with the method described in Section 3.3 and fine-tuned the models from the well-trained checkpoints until the accuracy is recovered to its maximum point. The fine-tune learning rate is set 10% of the initial learning rate for the training the model, and it is halved every 4 epochs. The fine-tuning process can be done in 20 epochs. The experiment is implemented on the workstation using NVIDIA's 2080Ti graphic cards. With the trained weights, we build a simulator to simulate the inference process and calculate the number of table lookup operations required to realize the inference. Combined with Xilinx XPE toolkit Xilinx (2018), the energy, latency results are obtained.

We test several popular models including LeNet-5, VGG-16, MobileNet, ResNet-18 and ResNet-34 on popular datasets such as MNIST, CIFAR10, CIFAR-100 and ImageNet. We then compare the accuracy of BLUnet with the baseline accuracy, which is got from conventional models with full-precision weights and activations (FP32). To make the estimation aligned to the real device, we estimate the energy consumption and latency of the LUT operation using the data from the Xilinx XPE toolkit Xilinx (2018) for commercial FPGA devices. Unless specified, we assume LUT-6 is used ($N = 6$), which is the most popular LUT type used today in modern FPGA. The notation 'BLUnet-M' corresponds to the BLUnet where each activation is quantized to M bits. We use $LUT - N \times M$ to form an N-input, M-output LUT group.

Table 2: Comparison BLUnet with traditional computation on different models.

Dataset / Model	Metrics	BLUnet-3	BLUnet-4	BLUnet-5	Base.
MNIST / LeNet-5	Top-1 accuracy	98.47	99.12	99.17	98.97
	Number of MAC Ops (10^6)	NA	NA	NA	0.4
	Number of LUT-6 Ops (10^6)	1.5	1.9	2.4	NA
	Energy per Op. (fJ)	4.5	5.1	5.6	82k
	Latency (ns)	28	38	47	674
Dataset / Model	Metrics	BLUnet-6	BLUnet-7	BLUnet-8	Base.
CIFAR-10 / ResNet-18	Top-1 accuracy	94.97	95.12	95.18	95.3
	Number of MAC Ops (10^9)	NA	NA	NA	0.6
	Number of LUT-6 Ops (10^9)	3.7	4.2	4.8	NA
	Energy per Op. (fJ)	0.5	0.6	0.6	7k
	Latency (ns)	878	1024	1170	3222
CIFAR-10 / VGG-16	Top-1 accuracy	93.46	93.65	93.70	93.83
	Number of MAC Ops (10^9)	NA	NA	NA	0.3
	Number of LUT-6 Ops (10^9)	2.1	2.4	2.7	NA
	Energy per Op. (fJ)	7.3	7.7	8.1	147k
	Latency (ns)	779	909	1039	2137
CIFAR-10 / MobileNet	Top-1 accuracy	90.00	90.97	91.38	91.44
	Number of MAC Ops (10^9)	NA	NA	NA	0.05
	Number of LUT-6 Ops (10^9)	0.3	0.4	0.4	NA
	Energy per Op. (fJ)	26.1	28	29	560k
	Latency (ns)	339	396	452	3641
CIFAR-100 / ResNet-18	Top-1 accuracy	76.62	76.94	77.14	77.50
	Number of MAC Ops (10^9)	NA	NA	NA	0.3
	Number of LUT-6 Ops (10^9)	3.7	4.2	4.8	NA
	Energy per Op. (fJ)	0.5	0.6	0.6	7k
	Latency (ns)	878	1024	1170	3222
CIFAR-100 / VGG-16	Top-1 accuracy	72.65	73.60	73.87	74.33
	Number of MAC Ops (10^9)	NA	NA	NA	0.3
	Number of LUT-6 Ops (10^9)	2.1	2.4	2.7	NA
	Energy per Op. (fJ)	7.3	7.7	8.1	147k
	Latency (ns)	779	909	1039	2137
CIFAR-100 / MobileNet	Top-1 accuracy	61.12	63.69	64.27	64.53
	Number of MAC Ops (10^9)	NA	NA	NA	0.3
	Number of LUT-6 Ops (10^9)	0.3	0.4	0.4	NA
	Energy per Op. (fJ)	26.1	28	29	560k
	Latency (ns)	339	396	452	3641
ImageNet / ResNet-18	Top-1 accuracy	66.88	69.37	70.03	69.76
	Number of MAC Ops (10^9)	NA	NA	NA	2
	Number of LUT-6 Ops (10^9)	12	13.8	15.6	NA
	Energy per Op. (fJ)	4.2	4.4	4.6	83k
	Latency (ns)	883	1031	1178	3235
ImageNet / ResNet-34	Top-1 accuracy	71.50	73.19	73.73	73.30
	Number of MAC Ops (10^9)	NA	NA	NA	4
	Number of LUT-6 Ops (10^9)	24.2	27.9	31.5	NA
	Energy per Op. (fJ)	4.2	4.4	4.6	83k
	Latency (ns)	1847	2155	2463	5604

4.2 Results of Different Models

We compare BLUnet with the traditional computation method on several popular models in terms of accuracy, latency, number of operations, and energy consumption against baseline models that use full 32-bit floating-point weights and activation, necessitating the use of floating-point multiply-accumulate operations. The energy consumption is estimated by counting the number of LUT operations and MAC operations. Latency is estimated under the estimation that maximum parallelism

Table 3: Comparison between the BLUnet and the state-of-the-art work. A negative value for 'Acc. Gain' represents a drop in accuracy.

ResNet-18 on ImageNet	Top-1	Top-5	Acc. Gain	Energy (mJ)	Norm. Ene.
AdderNet Chen et al. (2020)	69.8	89.1	0.04	28.01	400
Joint Sparsity Choi et al. (2018a)	67.8	NA	-1.96	25.50	364
FAQ McKinstry et al. (2018)	69.82	89.1	0.06	2.16	31
UNIQ Baskin et al. (2021)	67.02	NA	-2.74	4.32	62
QIL Jung et al. (2019)	67.02	NA	-0.56	1.22	17
LQ-Nets Zhang et al. (2018)	68.2	87.9	-1.56	1.22	17
ABC-Net Lin et al. (2017)	65.0	85.9	-4.76	3.38	48
Group-Net Zhuang et al. (2018)	69.2	88.5	-0.56	1.08	15
RegularizationChoi et al. (2018b)	69.2	88.5	-2.46	2.16	31
BLUnet-8	**70.03**	**89.43**	**0.27**	**0.07**	**1**
ResNet-34 on ImageNet	Top-1	Top-5	Acc. Gain	Energy (mJ)	Norm. Ene.
NISP Yu et al. (2018)	72.38	90.53	-0.92	81.12	579
FAQ McKinstry et al. (2018)	73.31	91.32	0.01	4.36	31
UNIQ Baskin et al. (2021)	71.09	NA	-2.21	8.73	63
QIL Jung et al. (2019)	73.1	NA	-0.2	2.45	18
LQ-Nets Zhang et al. (2018)	71.9	90.2	-1.4	2.45	18
ABC-Net Lin et al. (2017)	68.4	88.2	-4.9	6.82	49
BLUnet-8	**73.73**	**91.6**	**0.43**	**0.14**	**1**

are applied in classical accelerators, that is, all the multiplication operations in each neuron can be executed in parallel.

From Table 2, we can see that BLUnet is effective in reducing the energy consumption and latency of inference, while only incurring a small accuracy drop. For the large dataset, ImageNet, on the ResNet-18 and ResNet-34 models, our BLUnet-8 design shows a significant improvement over the respective baselines on energy efficiency, with no drop on top-1 accuracy. The BLUnet-7 design has an even better energy efficiency but at the cost of 0.39% and 0.11% top-1 accuracy drop. For smaller datasets such as CIFAR-10, BLUnet-8 can also achieve improvement on energy efficiency with only 0.13% of accuracy drop using the VGG-16 model. In addition to VGG and ResNet, BLUnet also performs well on the compact MobileNet model.

The experimental results show that although each LUT has a quantization error, the accumulated quantization error along many LUTs does not impact the accuracy of the models significantly. This can be attributed to the fact that the quantization error of each LUT is uniformly distributed with a mean close to zero. A positive quantization error from one LUT may be canceled out by a negative quantization error in another LUT. Therefore, the total accumulated quantization error is negligible.

4.3 Comparison to State-of-the-Art

Table 3 compares BLUnet with the state-of-the-art works in terms of accuracy and energy efficiency. The work listed in the table encompasses quantization and the energy reduction method that simplifies the model into an adder-only format. The energy consumption of the previous work is based on the number of operations required and the exact energy consumed by the corresponding operation type and bitwidth. From the table, we can see that BLUnet outperforms all others in terms of energy reduction, achieving high accuracy. With better top-1 accuracy, BLUnet can achieve at least $10\times$ further reduction compared to previous works.

The main reason behind the performance of BLUnet is the novel idea of using bit serialization to solve the fan-in problem of LUTs. This enables us to use very small LUTs to perform the computation of very complex functions. Finally, the accumulated quantization errors along many LUTs cancel each other out, resulting in very little accuracy drop compared to the baseline models.

4.4 Study on LUT Type for Energy, Latency, and Area

Fig. 5 shows design tradeoffs in BLUnet. Today, the most popular LUT type is LUT-6, which has 6 inputs and 2^6 possible combinations. Another two LUT-types are LUT-4 and LUT-5, which have

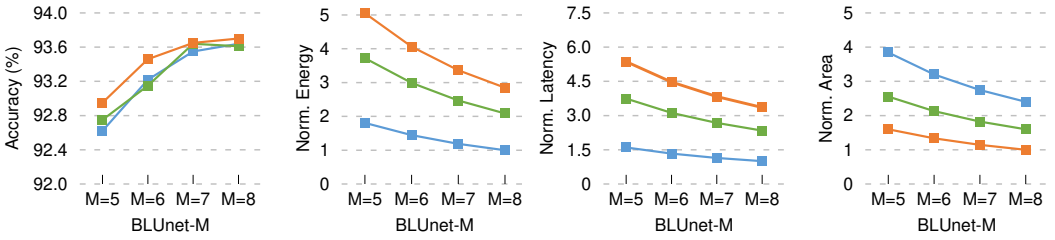

Figure 5: Accuracy, energy improvement, latency improvement and area improvement of BLUnet, for VGG-16 on CIFAR-10 dataset

four and five inputs, respectively, are also common. Theoretically, implementing BLUnet using LUT-4 and LUT-5 will increase the total number of LUTs and lower the accuracy because of the increase in accumulated quantization errors. However, LUT-4 and LUT-5 have higher area efficiency than LUT-6. Therefore, it is worth exploring especially for resource-constrained scenarios.

From Fig. 5, we can see that on average, LUT-4 and LUT-5 will cause a 0.18% and 0.15% accuracy drop, respectively. However, compared with the LUT-6 based BLUnet, the LUT-4 based one can achieve higher area efficiency than LUT-5. This is because the area of LUT-4 is only a quarter of that of LUT-6. Although the LUT-4 based BLUnet uses more LUTs than the LUT-6 based one, its overall area efficiency is still higher. Therefore, if 0.18% accuracy drop is acceptable, then LUT-4 based BLUnet is a better choice for area efficiency.

Another observation is that quantizing the activations into still fewer bits can improve the energy/area efficiency and latency. For example, if we quantize activation by 7 bits (BLUnet-7) instead of 8 bits (BLUnet-8), we can achieve a comprehensive higher efficiency in energy consumption, latency, and area overhead at the cost of 0.04% accuracy drop. This is because by lowering the precision of the activation, we can use fewer LUTs in the network.

4.5 Performance on real edge silicon

In order to demonstrate the effectiveness of BLUnet on real edge hardware with low energy constraints such as wearable devices, we design and implement a BLUnet prototype on FPGA. For a fair comparison with similar accuracy, we used a two-layer MLP with a hidden layer containing 216 neurons. We compare it with other low energy ASIC/FPGA implementations for edge vision with MNIST. According to the results in Table 4, compared with other low energy solutions, with similar accuracy, BLUnet achieves the best energy and throughput efficiency.

Table 4: Comparison between a prototype of BLUnet and other low energy classifiers on real hardware in terms of accuracy, energy per image, and images per second.

Approach	**Accuracy**	**Energy/image**	**Images/Sec.**	**Tech.**
TrueNorth (ASIC) Esser et al. (2015)	95.0%	4000nJ	1 x 10^3	28nm
Shenjing (Simulation) Wang et al. (2019a)	96.11%	2020nJ	5.95 x 10^4	28nm
BNN (FPGA) Umuroglu et al. (2017)	95.83%	591nJ	1.24 x 10^7	28nm
Tianjic (ASIC) Pei et al. (2019)	**96.59%**	3.8 x 10^4nJ	40	28nm
BLUnet (FPGA) (This work)	96.55%	**21.97 nJ**	**1.25 x** 10^8	28nm

5 Conclusion

In this work, we proposed BLUnet, a table lookup-based DNNs. By serializing the bits of activation, we successfully solve the 'show stopping' fan-in issue of lookup tables (LUT). BLUnet achieves high efficiency compared with MAC-based DNN models due to the advantages offered by the table lookup operation over traditional arithmetic (floating-point) multiply-accumulate (MAC) operation in hardware implementations. We conducted experiments on BLUnet with popular models and datasets. Our experimental results show that BLUnet achieves orders of magnitude higher improvement compared with not only MAC-based baseline designs but also the state-of-the-art efficient DNN designs that employ a wide range of optimization techniques.

REPRODUCIBILITY STATEMENT

We provide experimental instructions in the appendix to help the reproducibility. Source code would be made open source after the acceptance.

REFERENCES

Monther Abusultan and Sunil P Khatri. Look-up table design for deep sub-threshold through full-supply operation. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pp. 259–266. IEEE, 2014.

Chaim Baskin, Natan Liss, Eli Schwartz, Evgenii Zheltonozhskii, Raja Giryes, Alex M Bronstein, and Avi Mendelson. Uniq: Uniform noise injection for non-uniform quantization of neural networks. *ACM Transactions on Computer Systems (TOCS)*, 37(1–4):1–15, 2021.

Hanting Chen, Yunhe Wang, Chunjing Xu, Boxin Shi, Chao Xu, Qi Tian, and Chang Xu. Addernet: Do we really need multiplications in deep learning? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1468–1477, 2020.

Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. Compression of deep convolutional neural networks under joint sparsity constraints. *arXiv preprint arXiv:1805.08303*, 2018a.

Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. Learning low precision deep neural networks through regularization. *arXiv preprint arXiv:1809.00095*, 2, 2018b.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pp. 3123–3131, 2015.

Caiwen Ding et al. REQ-YOLO: A Resource-Aware, Efficient Quantization Framework for Object Detection on FPGAs. In *FPGA*, pp. 33–42. ACM, 2019.

Steve K Esser, Rathinakumar Appuswamy, Paul Merolla, John V Arthur, and Dharmendra S Modha. Backpropagation for energy-efficient neuromorphic computing. In *Advances in Neural Information Processing Systems*, pp. 1117–1125, 2015.

Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *NIPS*, 2016.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061*, 2016a.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran ElYaniv, and Yoshua Bengio. Binarized neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pp. 4114–4122, 2016b.

Sangil Jung, Changyong Son, Seohyung Lee, Jinwoo Son, Jae-Joon Han, Youngjun Kwak, Sung Ju Hwang, and Changkyu Choi. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4350–4359, 2019.

Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *Advances in Neural Information Processing Systems*, pp. 344–352, 2017.

Jeffrey L McKinstry, Steven K Esser, Rathinakumar Appuswamy, Deepika Bablani, John V Arthur, Izzet B Yildiz, and Dharmendra S Modha. Discovering low-precision networks close to full-precision networks for efficient embedded inference. *arXiv preprint arXiv:1809.04191*, 2018.

Ralph Nathan, Bryan Anthonio, Shih-Lien Lu, Helia Naeimi, Daniel J Sorin, and Xiaobai Sun. Recycled error bits: Energy-efficient architectural support for higher precision floating point. *arXiv preprint arXiv:1309.7321*, 2013.

Jing Pei, Lei Deng, Sen Song, Mingguo Zhao, Youhui Zhang, Shuang Wu, Guanrui Wang, Zhe Zou, Zhenzhi Wu, Wei He, et al. Towards artificial general intelligence with hybrid tianjic chip architecture. *Nature*, 572(7767):106–111, 2019.

Akshay Krishna Ramanathan, Gurpreet S Kalsi, Srivatsa Srinivasa, Tarun Makesh Chandran, Kamlesh R Pillai, Om J Omer, Vijaykrishnan Narayanan, and Sreenivas Subramoney. Look-up table based energy efficient processing in cache support for neural network acceleration. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 88–101. IEEE, 2020.

Abhronil Sengupta, Yuting Ye, Robert Wang, Chiao Liu, and Kaushik Roy. Going deeper in spiking neural networks: Vgg and residual architectures. *Frontiers in neuroscience*, 13:95, 2019.

Amirhossein Tavanaei, Masoud Ghodrati, Saeed Reza Kheradpisheh, Timothée Masquelier, and Anthony Maida. Deep learning in spiking neural networks. *Neural Networks*, 111:47–63, 2019.

F. Tung et al. CLIP-Q: Deep Network Compression Learning by In-parallel Pruning-Quantization. In *CVPR*, pp. 7873–7882, June 2018.

Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74, 2017.

Yaman Umuroglu, Yash Akhauri, Nicholas James Fraser, and Michaela Blott. Logicnets: Co-designed neural networks and circuits for extreme-throughput applications. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 291–297. IEEE, 2020.

Bo Wang, Jun Zhou, Weng-Fai Wong, and Li-Shiuan Peh. Shenjing: A low power reconfigurable neuromorphic accelerator with partial-sum and spike networks-on-chip. *arXiv preprint arXiv:1911.10741*, 2019a.

Erwei Wang, James J Davis, Peter YK Cheung, and George A Constantinides. Lutnet: Rethinking inference in fpga soft logic. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 26–34. IEEE, 2019b.

Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8612–8620, 2019c.

Zhehui Wang, Xiaozhe Gu, Rick Goh, Joey Tianyi Zhou, and Tao Luo. Efficient spiking neural networks with radix encoding. *arXiv preprint arXiv:2105.06943*, 2021.

Yujie Wu, Lei Deng, Guoqi Li, Jun Zhu, Yuan Xie, and Luping Shi. Direct training for spiking neural networks: Faster, larger, better. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 1311–1318, 2019.

Xilinx. Vivado Design Suite User Guide. *Technical Publication*, 2018.

Jiwei Yang, Xu Shen, Jun Xing, Xinmei Tian, Houqiang Li, Bing Deng, Jianqiang Huang, and Xiansheng Hua. Quantization networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 7308–7316, 2019.

Tien-Ju Yang et al. Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning. In *CVPR*, pp. 5687–5695, 2017.

Haoran You, Xiaohan Chen, Yongan Zhang, Chaojian Li, Sicheng Li, Zihao Liu, Zhangyang Wang, and Yingyan Lin. Shiftaddnet: A hardware-inspired deep network. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 2771–2783. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/1cf44d7975e6c86cffa70cae95b5fbb2-Paper.pdf.

Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S Davis. Nisp: Pruning networks using neuron importance score propagation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 9194–9203, 2018.

Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pp. 365–382, 2018.

Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

Bohan Zhuang, Chunhua Shen, and Ian Reid. Training compact neural networks with binary weights and low precision activations. *arXiv preprint arXiv:1808.02631*, 2018.

A APPENDIX: EXPERIMENTAL INSTRUCTIONS

We implement BLUnet by re-writing neuron layers in Pytorch platform. The convolutional layer and the linear layer use different methods, which will be discussed in two parts.

A.1 CONVOLUTIONAL LAYER

For `nn.conv2d`, the key idea in this algorithm is to decompose the original convolutional computation into many groups so that we can implement the quantization error among LUTs. Therefore, instead of computing the traditional convolutional layers with only one group, we compute a convolutional layer with `ch_in` groups, where `ch_in` is the number of input channels. As a result, there is one input channel in each group. Our algorithm is to make sure that within every group, the only input channel will do convolution with all weights it involves with in the original computation algorithm. It then lists all results as multiple output channels for further processing.

In Fig. 6. we show how our implementation works using a simple example. In this example, the input activation size is 2x2x3, and the filter size is 1x1x3x2, so the output activation size is 2x2x2.

Before the main algorithm, we obtain the dimension of the original weights from the following code.

```
ch_out, ch_in, kernel_x, kernel_y = self.weight.shape[0],
self.weight.shape[1], self.weight.shape[2], self.weight.shape[3]
```

A.1.1 BASIC ALGORITHM

To make it easy to understand, we first show how it works in a 1×1 kernel convolution layer.

In the original computation loop, all MAC operations along all the input channels are accumulated to one single sum. In our new algorithm, we do not sum these MAC operation results over all input channels. Instead, the result from each input channel is separately calculated. In the new algorithm, the number of the output channel is `ch_out* ch_in`, which means that for each input channel, there are `ch_out` output channels.

To implements this, we need to covert the weight matrix by repeating some weights. This can be done by transposing and reshaping the original weight matrix.

```
weight_expend = torch.transpose(self.weight, 0, 1).reshape(ch_in *
ch_out, -1, kernel_x, kernel_y)
```

The next step is the key to this algorithm. We do the convolution using the converted weight matrix.

```
activation_expand = F.conv2d(input = input, weight = weight_expend,
bias = None, stride = self.stride, padding = self.padding,
dilation = self.dilation, groups = self.in_channels)
```

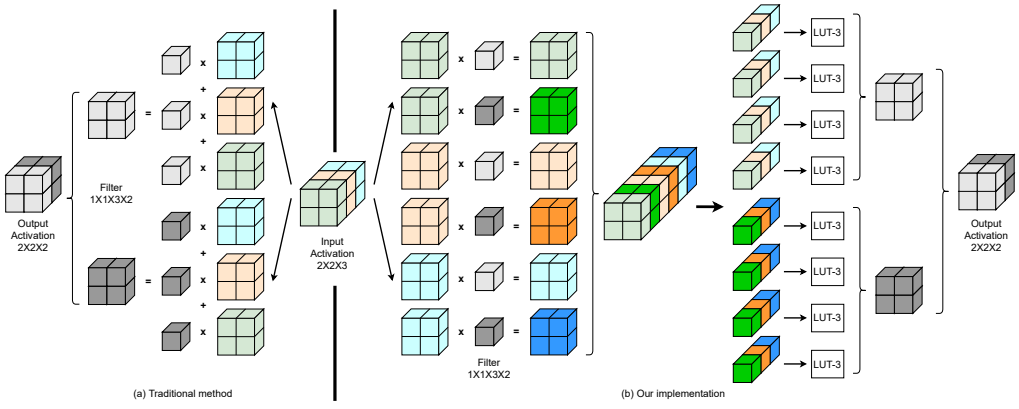

Figure 6: Comparison between the traditional method and our implementation.

Then we need to read out the batch size, and activation size for further processing.

```
batch_num, act_x, act_y = activation_expand.shape[0],
activation_expand.shape[2], activation_expand.shape[3]
```

Now for every six MAC outputs, we use LUT-6 to realize the function, which can realize the six MAC operations at a single time.

```
activation_regrouped = activation_expand.view(batch_num, ch_in, -1,
act_x, act_y)
```

```
activation_regrouped = torch.sum(activation_regrouped.view(batch_num,
-1, 6, ch_out, act_x, act]_y), 2)
```

In this step, we apply the quantization function to the tensor in the last step. This is to realize the quantizing function of each LUT due to the reason that the LUT output would not be a full precision value. Here the parameter $res\_activation$ indicates the quantization bits.

```
activation_regrouped_quantized = QuantizeThrough.apply(activation_regrouped,
res_activation)
```

Finally, we implement the following addition function with LUT-based operations.

```
activation = torch.sum(activation_regrouped_quantized, 1)
```

A.1.2 Extension to Generally Cases

In the previous part, we discuss the method on 1×1 convolutional layer. In this part, we will use an example to briefly introduce the method for $N \times N$ convolutional layers. For instance, in a 3×3layer, each output result is the accumulation of 9 MAC operations. Since 9 is greater than 6, the fan-in of the LUT, we need to divide the 9 MAC operations into 3 groups by using a mask and further decompose the output results.

```
weight_mask = torch.tensor([[[1, 1, 1], [0, 0, 0], [0, 0, 0]],
[[0, 0, 0], [1, 1, 1], [0, 0, 0]], [[0, 0, 0], [0, 0, 0], [1, 1,
1]]]).view(1, 1, 3, 3 ,3).expand(ch_in, ch_out, -1, -1, -1)
```

We then use the masked weight to do convolutions. As a result, the output channels will be further extended by 3.

```
activation_expand = F.conv2d(input = input, weight =
weight_masked.reshape(ch_in * ch_out * 3, -1, kernel_x, kernel_y),
bias = None, stride = self.stride, padding = self.padding,
dilation = self.dilation, groups = self.in_channels)
```

Because in this example, each group contains three MAC operations, we combine any two of them to get the result with six MAC operations.

```
activation_regrouped = torch.transpose(activation_regrouped, 1,
2).reshape(batch_num, ch_out, -1, 2, act_x, act_y)
```

The later steps would follow the basic algorithm.

A.2 Linear Layer

For the linear layer, we transform the computation into 1D convolutional layers. The basic idea is similar to what we do for convolutional layers. We decompose the computation into several groups. Each group includes six MAC operations. The reason why the 1D convolutional layer is used is that it can help us to compute the result in parallel.

The first thing in this algorithm is to convert weight and input activation in order to fit into 1D convolutional layer.

```
input_expand = input.view(batch_num, -1, 6)
```

```
weight_expend = torch.transpose(weight.view(element_out, -1, 6), 0,
1).reshape(-1, 1, 6)
```

Afterwards, we use the standard 1D convolutional function to compute the results in parallel.

```
activation_expand = F.conv1d(input = input_expand, weight =
weight_expend, bias = None, stride = 1, padding = 0, dilation =
1, groups = input_expand.shape[1])
```

Finally, we apply the quantization function to the output tensor and sum results, and implement the following addition function with LUT-based operations.