Zac: Towards Automatic Optimization and Deployment of Quantized Deep Neural Networks on Embedded Devices

(Invited Paper)

Qingcheng Xiao, Yun Liang†
Center for Energy-efficient Computing and Applications, EECS, Peking University
{walkershaw,ericlyun}@pku.edu.cn

ABSTRACT

With the development toward commercial and civil use, the need for deploying Deep neural network (DNN) models on resource-constrained embedded devices is growing. Quantization has a paramount impact on the performance, storage, and energy efficiency. However, to fully realize these benefits, programmers need to manually utilize low precision operations while maintaining accuracy, which is very challenging. Hence, we present a framework `Zac` to automatically optimize and deploy quantized DNN models on embedded devices. In order to do this, `Zac` performs necessary data type conversion and chooses proper data types for the intermediate data. Then it automatically customizes the operations according to the chosen types. Experiments demonstrate that by utilizing quantized models, `Zac` offers up to 19.18X and 25.44X improvement for throughput and energy efficiency compared with full precision designs, respectively. The automatically generated hardware designs from `Zac` achieve comparable performance to the highly optimized state-of-the-art accelerators which are designed manually.

1 INTRODUCTION

DNNs achieve impressive accuracy among a wide range of domains, from optical character recognition to fluid simulation. The widespread adoption of mobile and IoT devices has led DNN applications increasingly deployed onto small and imperceptible embedded devices to improve our daily life. However, the DNN deployment is complicated as embedded devices vary hugely in terms of computation, memory, and bandwidth capabilities. For example, different hardware platforms are not standardized for their supported data types. ARM NEON CPUs support vectorized 16-bit floating point operations in SIMD fashion; NVIDIA GPUs have introduced INT8 (8-bit integer) for deep learning inference; FPGAs support arbitrary precision operations; ASICs also employ dedicated processing units to deal with varying precision requirements [15]. Besides, hardware platforms have diverse memory capacities. For example, NVIDIA TX2 GPU board has 8 GB memory, while Xilinx ZC706 FPGA board only has 2.4 MB on-chip block RAM.

DNN applications, especially when deployed on embedded devices, demand high performance, low latency, and low power consumption. The combination of these requirements calls for novel ways that the hardware is designed and optimized. In this work, we focus on data quantization, an important knob that helps to reduce the computation intensity and memory storage on resource-constrained embedded devices without impacting the accuracy. The benefits of quantization come from several folds. First, an inherent advantage of quantized models is that the operations on low precision data cost less energy and hardware area. Second, the DNN model size scales linearly with the bitwidth of the data type. Low precision data has much shorter bitwidth so that model size is significantly reduced. Smaller models also require less communication and storage in deployment. Third, quantized models contribute to performance improvement. The reduced computational complexity of low precision operations leads to low hardware latency. Finally, the energy, area, storage, and performance benefits also contribute to a better user experience, smaller device sizes, and longer battery life, etc.

While the benefits of quantization are clear, it is challenging to deploy quantized DNN models onto embedded devices. First, both embedded hardware platforms and software DNN models are diversified. Hardware diversification refers to the fact that different hardware platforms are equipped with a different set of quantized data types, arithmetic units, and embedded memory. Therefore, deploying the quantized models onto different hardware requires significant manual effort. The diversity of DNN models adds further complications as different DNN models or different layers of the same DNN models may use different quantization strategy. Second, quantized models only assign data types for weights and activations [9], neglecting intermediate data storing partial results in computations. Even so, the assigned types might not be supported by the hardware. Improper types introduce quantization errors in deployment. Hence, programmers need to customize the original computation by taking advantage of the available data type and low precision computation on the hardware without hurting the application accuracy.

In this paper, we propose a framework `Zac` to automate the quantization optimization when deploying the DNN models onto embedded devices. `Zac` first defines a set of succinct and powerful abstraction for DNN computations using tensor expressions which are arithmetic expressions with tensors (multi-dimensional data arrays) as their operands [2, 19]. For common DNN computations like convolutions and matrix multiplications, we build a library in `Zac` using our tensor expressions for fast development. For given DNN models written in tensor expressions, `Zac` automatically generates optimized hardware designs for a diverse set of hardware platforms, including FPGAs, GPUs, and CPUs. We develop three quantization-specific optimization passes in `Zac`. First, `Zac` converts customized data types of quantized models to native data types of the target platform. Then, it determines the data types for all intermediate data through range and precision analysis based on symbolic execution. Last, it customizes the operation implementations for different operand bitwidths and target platforms. Through these passes, `Zac` ultimately generates source code for a given quantized model.

The key contributions of this paper include:

- We propose `Zac`, a framework that automatically optimizes and deploys quantized DNN models on embedded devices.

† Corresponding author.

978-1-7281-2350-9/19/$31.00

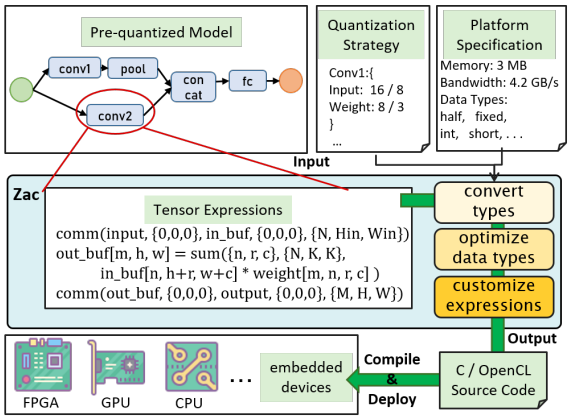

Figure 1: The workflow of Zac.

- We propose tensor expressions to serve as a unified cross-platform programming interface for DNNs.
- We develop three quantization-specific passes to optimize tensor expressions for utilizing quantized models.

Experiments using various DNNs are conducted to evaluate Zac. Our automatically generated designs achieve comparable performance to the state-of-the-art highly optimized DNN accelerators which are designed manually. In addition, for text recognition application [16], Zac offers up to 19.18X and 25.44X improvement for throughput and energy efficiency compared to the full precision designs, respectively.

2 WORKFLOW OF ZAC

Figure 1 depicts the workflow of Zac. The inputs of Zac consist of three parts: a DNN model, a quantization strategy, and a platform specification. The model is represented as a computational graph where nodes represent DNN layers and edges represent data dependencies among layers. For each node, it is defined in tensor expressions. Zac employs tensor expressions as a unified programming interface for abstracting and representing computation and transformation for heterogeneous embedded platforms. For commonly used layers, we build an expression library in Zac for fast implementation. For new layers, users are able to define them with tensor expressions easily.

Based on these expressions, Zac performs quantization-specific optimizations. First, Zac performs type conversions between various data types. Then, it chooses data types for the intermediate data within each layer. Last, Zac customizes the implementation for each layer according to the employed data types and the target platform. By these quantization-specific optimizations, the implementations achieve high performance while maintaining accuracy. As Figure 1 illustrates, optimizations are performed under the guidance of the quantization strategy and platform specification. The quantization strategy specifies the data types for all weights and activations.

Finally, the transformed tensor expressions are then lowered to abstract syntax trees based on which source code are generated with the assist of toolchains like LLVM. Different source code would be generated for various embedded devices, such as CUDA or OpenCL for GPUs, and C-based HLS for FPGAs. Then the source code could be compiled and deployed. It's notable that Zac is orthogonal to other tools performing computational graph transformations, such as TVM [2] and TensorRT.

3 TENSOR EXPRESSIONS

Here, we define a concise set of tensor expressions that express tensor computation in DNN models in an abstract way. Zac will automatically map the tensor expressions onto low-level bit-level implementations, providing high programming productivity, good platform portability and high performance. Though a number of previous works [2, 19] also employ tensor expressions as an intermediate representation, they support neither quantization features nor fine-grained control over computations. Since the performance bottleneck of a DNN layer could be either computation or communication, we address both aspects. Especially, high-performance computation usually owes to efficient operation implementation, high parallelism, and good data locality. Based on this insight, we define expressions in Table 1 to generate more efficient designs.

First, expressions should be aware of different data types so that their implementations can be customized. We introduce the φ function to dictate the expressions that are sensitive to data types. Computations represented by these expressions can be reformed by the quantization-driven expression optimization in Section 4.3.

Second, in addition to computation, specifying communications is necessary. For platforms that handle data transfer explicitly, such as GPUs and FPGAs, elaborating communications is essential to performance. Thus, we introduce the communication expression comm(src, s_idx, dst, d_idx, dims) which specifies the source to read, destination to write, and the total transfer size.

Third, parallelism information is vital to achieving high performance. Distributing workloads to multiple threads and pipelining related tasks are two main sources of parallelism. To indicate parallelism information, thread-level and task-level parallelism expressions are employed. At thread level, map(idx, n) and reduce(idx) expressions are used to distribute workloads of a loop to n threads and collect results. At task level, annotations including <@stage = name . . . > and <@pipeline = name . . . > are used to group expressions as a task stage and assign the task to a pipeline.

Last, loop transformations such as tiling and interchange are common data locality optimizations used in DNN layers. Performing loop transformations directly on expressions changes the corresponding indexes, complicating the original expressions. To perform transformations easily, we decouple them from other expressions by introducing transformation expressions. Tile expression tile(idx, inner_bnd) factorizes the iteration space of the loop with iterator idx. Fuse expression fuse(idx1, idx2, idx) coalesces iteration space of loops with iterators $idx1$ and $idx2$ into a new loop with iterator idx. Interchange expression reorder(idx1, idx2) reverses the scheduling orders of loops with iterators $idx1$ and $idx2$. In addition, common mathematical functions are treated as built-in expressions, such as *sum*, *sigmoid*, and others.

With these expressions, we build a library in Zac for common DNN computations, such as convolution and matrix multiplication. Listing 1 gives a pipelined tiled convolutional layer written in tensor expression. The input consists of N feature maps and each map has size $H_{in} \times W_{in}$. Input feature maps convolve with one $K \times K \times N$ weight block at stride S to produce one output feature map with size $H_{out} \times W_{out}$. Totally, M weight blocks are convolved to generate M output feature maps. As the annotations dictate, the *conv* pipeline is composed of three stages: *load*, *core*, and *store*. In *load* and *store*

Table 1: Proposed tensor expressions.

Type	Syntax	Description
Customization	expr.φ()	The implementation of expressions dictated by φ would be customized
Communication	comm(src, s_idx, dst, d_idx, dims)	Transfer a data block with size $dims$ from tensor src to dst. The read starts at index s_idx and the write starts at index d_idx.
Thread Parallelism	map(idx, n)	Distribute workloads of the loop with iterator idx to n threads
	reduce(idx)	Collect the results of all threads for the loop with iterator idx
Task Parallelism	<@stage = name ...>	Mark a group of expressions as a pipeline stage
	<@pipeline = name ...>	Mark a group of stages as a pipeline
Loop Transformation	tile(idx, inner_bnd)	Factorize the loop with iterator idx into two loops and the boundary of inner loop is $inner\_bnd$
	fuse(idx1, idx2, idx)	Coalesce the loops with iterators $idx1$ and $idx2$ into a new loop with iterator idx
	reorder(idx1, idx2)	Reverse the scheduling order of loops with iterators $idx1$ and $idx2$

stages, communication expressions are used to transfer inputs and outputs. All expressions are tiled in three dimensions with tile size set as 32. In the core stage, core_expr is dictated by φ so that its implementation would be customized latter.

Listing 1: Example of a pipelined tiled convolutional layer.

```
#all inner loop boundaries for tiling are set to 32
<@pipeline=conv <@stage=load
ld_expr: comm(input, {0, 0, 0}, in_buf, {0, 0, 0}, {N, Hin, Win})
ld_expr.tile(i, 32).tile(j, 32).tile(k, 32)
> <@stage=core
core_expr: out_buf[m, h, w] =
sum({n, r, c}, {N, K, K}, in_buf[n, h + r, w + c] × weight[m, n, r, c])
#sum(indexes, boundaries, body)
core_expr.tile(m, 32).tile(h, 32).tile(w, 32).φ()
> <@stage=store
st_expr: comm(out_buf, {0, 0, 0}, output, {0, 0, 0}, {M, H, W})
st_expr.tile(i, 32).tile(j, 32).tile(k, 32) > >
```

4 OPTIMIZATION PASSES

Once layers are defined in tensor expressions, Zac implements quantized models layer by layer. Zac needs to determine data types for all tensors along with the operation implementations. To do this, Zac first performs the type conversions for weights and activations and optimizes data type for intermediate data. Then it customizes the implementations of expressions according to data types.

4.1 Type Conversion

When deploying DNNs, the target hardware platform may only support limited data types. Zac treats all unsupported types as customized types. A customized type differs from native types in that it requires a translation function that explicitly translates a binary string to the represented value. Given a quantized model, only the data types of weight and activation tensors are assigned. Even so, these data types could be customized types. Thus, to serve all kinds of quantized models and target platforms, Zac first determines the data type mapping for these tensors and performs type conversion.

Assume v is a weight or activation tensor, T_v is the type assigned by the quantized model for v, and PT represents all native types of the target platform. If $T_v \in PT$, Zac keeps the type. Otherwise, Zac uses T_v as the storage type and finds another type $T'_v \in PT$ as the compute type. When using T'_v to approximate T_v, data values represented by these two types might differ, which raises an error err. We evaluate err as the sum of range error rng_err and precision error $prcs\_err$ as below:

$$\begin{aligned} rng\_err(T_v, T'_v) &= \max\{0, val_{max}(T'_v) - val_{max}(T_v), \\ &\qquad val_{min}(T_v) - val_{min}(T'_v)\} \\ prcs\_err(T_v, T'_v) &= \max\{0, prcs(T'_v) - prcs(T_v)\} \\ err(T_v, T'_v) &= rng\_err(T_v, T'_v) + prcs\_err(T_v, T'_v) \end{aligned} \tag{1}$$

where $val_{max}(x)$ and $val_{min}(x)$ are the maximal and minimal value represented by the type x, respectively, and $prcs(x)$ is the maximal precision of x. Accordingly, Zac chooses the type with minimal bitwidth as T'_v while err is kept within a threshold.

Then Zac inserts conversion expressions into the original expression group wherever v is accessed. Assume the translation function of T_v is $f()$, and $T'_v()$ is employed as an implicit translation function. Before any expression that reads v, a decoding expression $v' = (T'_v)f(v)$ is inserted and the expression reads v' instead. Similarly, an encoding expression $v = f^{-1}(v')$ is inserted. Listing 2 gives a type conversion example of a *ReLU* layer. Assume the compute type T'_v is the standard 16-bit floating point and the storage type T_v is a customized type with its translate function f being *atanh* [12]. In this case, data are decoded by $(half)atanh()$ and then encoded by $tanh()$ around relu_expr.

Listing 2: Example of type conversion.

```
#after inserting encoding/decoding expressions
decode_expr: in_tmp[i] = (half)atanh(in_data[i])
relu_expr:   result[i] = max(0, in_tmp[i])
encode_expr: out_data[i] = tanh(result[i])
#after merging
merged_expr: out_data[i] = max(0, in_data[i])
```

Furthermore, inserting conversion expressions might introduce redundant computations. Especially, for activation layers where an element-wise function $g()$ are applied, redundant conversions lead to significant overhead. Hence, Zac further omits unnecessary conversions if the element-wise function $g()$ and the translate function $f()$ of the customized type commute. As Listing 2 shows, since *ReLU* and *atanh* commute, conversion expressions are omitted.

4.2 Data Type Optimization

In addition to weight and activation tensors, Zac also optimizes intermediate data tensors when deploying quantized models. To choose proper data types for intermediate tensors, Zac adopts a twofold approach as illustrated in Figure 2.

As shown in Equation 1, a data type might introduce both range errors and precision errors compared with the true value. To address these errors, Zac first derives the value range and required precision for each intermediate tensor by performing range analysis and precision analysis, respectively. Assume that for a tensor v, the obtained value range is $[a, b]$ and the required precision is p. Zac chooses a native type T_v with the minimal bitwidth to meet the following constraint:

$$\begin{aligned} val_{min}(T_v) \le a \le b \le val_{max}(T_v) \\ p \le prcs(T_v) \end{aligned}$$

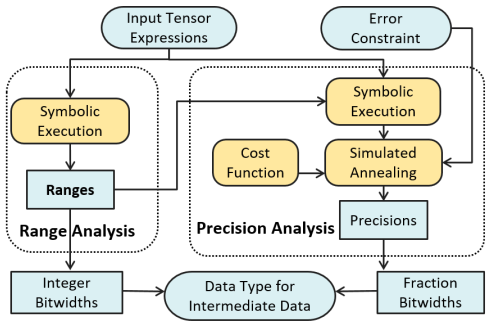

Figure 2: The twofold approach to optimizing data type.

In another word, T_v covers the value range of v and meets the required precision.

More concretely, for range analysis, Zac performs symbolic execution on the tensor expression group. Though tensor expressions have defined tensor-level operations, Zac further finds all involved elements for computing each intermediate data element. Zac derives this element-wise dependency by analyzing tensor indexes. Take the core_expr in Listing 1 for example. Each element in out_buf depends on $N \times K \times K$ elements from both in_buf and $weight$. Then, to calculate the range interval for an intermediate data element, interval arithmetic [7] is performed on the range intervals of all involved elements. For all common operations, we have defined the corresponding interval arithmetic in Zac. For example, the result interval of multiplying two intervals $[a, b]$ and $[c, d]$ is:

$$[a, b] \times [c, d] = [\min\{ac, ad, bc, bd\}, \max\{ac, ad, bc, bd\}]$$

Different from data-flow range analysis [14], this symbolic execution method is both loop-boundary sensitive and array sensitive. Hence, for DNN applications to which loops and tensors are essential, Zac gives more accurate range intervals.

Once the value ranges are obtained, Zac analyzes the required precisions for all intermediate tensors. Faced with various precision combinations, Zac employs the Adaptive Simulated Annealing (ASA) algorithm to find a precision solution. For each solution, the precision error of each tensor is propagated through tensor expressions. We estimate the precision errors according to the worst-case quantization error models proposed in [8]. A solution is only valid if the estimated error of output tensor is under an error constraint. Zac employs ASA algorithm to find the history best solution in a given number of iterations. The solution cost is determined by users, such as estimated latency or required resources.

4.3 Expression Customization

After determining data types for all tensors, Zac is able to customize their computations to bit-level implementations. Among all expressions, Zac focuses on the most compute-intensive ones dictated by φ. For each operation, multiple implementations are possible as the platform and operand bitwidth change. Zac compares the latency or resource usage of various implementations and chooses the one with minimal cost. Here, we demonstrate the customization of three operations that dominate the most computations in DNNs: *Sum*, *Product*, and *Activation*. *Sum* and *Product* are combined to define RNN gates, fully connected, and convolutional layers. Table 2 compares the resource usage of different implementations for these operations on FPGA platforms.

Table 2: Resource consumption of different implementations.

Operation	Implementation	Operand Bitwidth				Resource Usage[1]	
		1 - 2	3 - 7	8 - 16	FP32	DSP	LUT
Sum	ADD		√	√	√	0-2	8-432
	bitwise operations	√				0	10
Product	MULT		√	√	√	0-3	8-322
	bitwise operations	√				0	1 - 3
	conditional result	√				0-2	8-432
Activation	arithmetic operations		√	√	√	-	-
	lookup table	√	√	√		-	-

Sum. Its default implementation is accomplished by adders. If the input data are constrained to one or two bits, *Sum* could be implemented by bitwise operations. For example, in XNOR-Net [13] where 1-bit is used to represent {1, -1}, *Sum* result of the input binary could be obtained as below:

$$result = (pop\_count(input_n) << 1) - n$$

where n is the bit number of $input$, and pop_count counts the number of bits that are 1.

Product. It has two operands and the default implementation is to use multipliers. If only one operand is constrained to one or two bits, the *Product* result is obtained conditionally. For instance, Ternary-Net [1] uses floating point type for activation data and 2-bits for weights. By this way, weights are constrained to {1, -1, 0}. Accordingly, the result of multiplying activation with weights could be the original value, the opposite of activation, or zero, respectively. Furthermore, when both operands are represented by one or two bits, *Product* would also be implemented by bitwise operations. In XNOR-Net, *Product* between 1-bit data is equivalent to XNOR operation. Combining *Sum* and *Product*, convolution defined by *core_expr* in Listing 1 would be customized as below for XNOR-Net:

$$out\_buf[m, h, w] = -N \times K \times K + (pop\_count(in\_buf_{N\times K\times K}[h, w] \odot weight_{N\times K\times K}[m]) << 1)$$

where in_buf and $weight$ are $N \times K \times K$ bits binary strings, $output$ is an integer, and $\odot$ is the XNOR operator.

Activation. It represents all element-wise operations in DNNs. In addition to *ReLU*, *Sigmoid*, and other functions employed in activation layers, batch normalization, data encoding and decoding are all element-wise *Activation* operations. By default, these operations are implementations by arithmetical operation or vendor-provided math library. For short bitwidth data types, Zac can also perform customization. For instance, the default implementation of batch normalization is a multiply-add operation as the following equation:

$$y = \frac{x - u}{\sqrt{\sigma^2 + \varepsilon}}\gamma + \beta$$

where x and y are input and output, respectively, γ and β are the parameters to be learned, and others are given parameters. However, if the output is constrained to n bits, Zac builds a lookup table with 2^n entries. When n is set to 1, batch normalization is simplified to a comparison with a threshold as follow:

$$y = \begin{cases} 1 & x > TH \\ 0 & x \le TH \end{cases} \qquad TH = u - \frac{\beta\sqrt{\sigma^2 + \varepsilon}}{\gamma}$$

where x could be of arbitrary type, and TH is the threshold.

[1]We evaluate resource usage range per operation on Xilinx platforms. A range is given when the implementation is applied to multiple precisions.

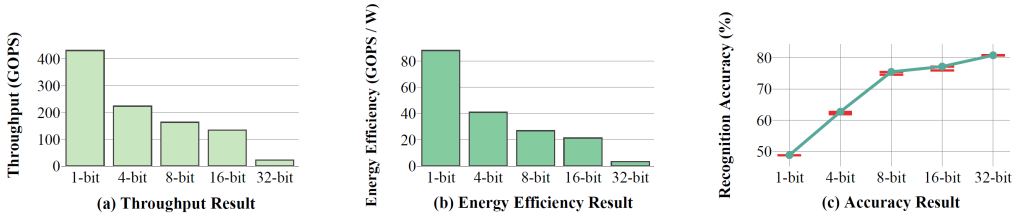

Figure 3: Throughput, energy efficiency, and accuracy of CRNN [16] models on ZC706.

5 EVALUATION

To validate Zac, we use two off-the-shelf embedded devices: Xilinx zynq ZC706 and NVIDIA Jetson TX1. ZC706 board consists of one XC7Z045 FPGA chip, dual ARM Cortex-A9 CPUs, and 1 GB DDR3 memory. TX1 board is composed of a Maxwell GPU, a quad-core ARM A57 CPU, and 4 GB DDR4 memory shared by CPU and GPU. For code generation and deployment, we employ LLVM 3.4 compiler infrastructure, NVCC(v9.0.176), and Xilinx Vivado SDx(v2017.1). For the adaptive simulated annealing in bitwidth optimization, we use the off-the-shelf ASA package v28.12. All quantized models for deployment are obtained using PyTorch framework [10]. In our experiments, we first generate counterparts of state-of-the-art FPGA accelerators designed for various quantized models, including both manually optimized and automatically generated designs. Then we take the text recognition application as a case study for both FPGA and GPU platforms.

5.1 FPGA Results

Comparison with manually optimized designs. We first employ Zac to generate counterparts of three state-of-the-art accelerators for quantized DNNs, as shown in Table 3. We compare resource efficiency to demonstrate performance. For designs using long bitwidth data types, we calculate DSP efficiency (throughput per DSP slice); for designs using short bitwidth data types, we compare logic efficiency (throughput per kilo LUTs) instead.

As Table 3 shows, for VGG-16 [17], the design generated by Zac achieves 0.211 GOPS/DSP compared with 0.213 GOPS/DSP of [6]. This small performance gap is due to the last three fully connected layers which are bounded by bandwidth. For DoReFa-Net [24] and XNOR-Net [13], they are both quantized variants of AlexNet. For their first convolution layers, image input is represented by 20-bit fixed point or floating point data. For the rest layers, DoReFa-Net uses one bit for activation data and two bits for weights, while all data in XNOR-Net are 1-bit. Zac customizes these layers using XNOR and popularity count operations as discussed in section 4.3. These bitwise operations only need a small number of logic resources. The saved DSPs lead to huge energy consumption saving. For DoReFa-Net, the generated design achieves 405.82 GOPS throughput and 72.47 GOPS/W energy efficiency. However, [4] achieves higher performance for its efforts in data arrangement and architecture design. For XNOR-Net, the logic efficiency of the generated design is 8.615 GOPS/kLUT compared with 4.431 GOPS/kLUT of [23]. In another word, the generated design offers 1.39X throughput speedup and 2.05 energy efficiency improvement than [23] with similar resources. In summary, for quantized DNN models, Zac is able to generate designs that achieve comparable performance with manually optimized accelerators.

Comparison with automatically generated designs. In addition to dedicated accelerator designs, we also conduct a comparison between Zac and one of these automated tools TVM [2]. TVM employs an FPGA-based tensor processor and generates instructions for ResNet [3]. Since the tensor processor only supports 8-bit fixed point data, when deploying a 4-bit model, it would convert data to 8 bits and achieve similar the performance result to the 8-bit model deployment result. Instead, Zac is able to deploy both 4-bit and 8-bit models while scaling the performance. Besides, the designs generated byZac achieve higher throughput due to architecture difference. Since tensor processor acts as a general solution for various models, it is usually less energy-efficient since extra energy is consumed by fetching and decoding instructions. Instead, the designs generated by Zac are totally composed of datapaths though they are dedicated for ResNet only.

Case study of text recognition. We then use deploying a DNN model CRNN used in text recognition application [16] on Xilinx ZC706 platform as a case study. CRNN is composed of 7 convolutional layers, 4 pooling layers, 2 batch normalization layers, and 2 LSTM gates. We use Street View Text dataset for training and testing. We employ Zac to deploy four quantized models using 1, 4, 8, and 16 bits along with the original full precision model.

As Figure 3(a) and (b) illustrates, by utilizing quantized models, both throughput and energy efficiency are significantly improved. Especially, for the 1-bit model, the design achieves 431.5 GOPS throughput and 88.06 GOPS/W energy efficiency, which are 19.18X and 25.44X higher than the ones of full precision design, respectively. The performance gain is owing to higher achieved parallelism. When the model is quantized using 1-bit data, Zac implements computation like convolutions with bitwise operations such as XNOR and popularity count operations. In other cases, MAC operations are used. Lower precision operation requires much fewer resources, leading to higher operation parallelism. From Figure 3(c), we conclude that designs using lower precision data tend to have lower accuracies. However, the accuracy loss mainly comes from the inherent error when quantizing models. As the error bars in the figure show, Zac keeps errors from being exacerbated. In all cases, the extra errors introduced by intermediate data are limited within 2%.

5.2 GPU Results

We also deploy three CRNN models using 1, 16, and 32 bits onto the TX1 platform since TX1 supports integer, FP16, and FP32. The implementation comparisons are presented in Table 4. Similar to FPGA implementations, bitwise operations are used for the 1-bit model, while MAC operations are employed for the other two models. Though all these operations are processed by the same arithmetic unit, the parallelism of processing bitwise operation, FP16 MAC, and FP32 MAC are 32, 2, and 1, respectively. As Table 4 shows, with

Table 3: Performance comparison with previous DNN accelerators.

Model	VGG-16		DoReFa-Net		XNOR-Net		ResNet-18		
	[6]	Zac	[4]	Zac	[23]	Zac	[2]	Zac	
Platform	Nallatech 385A	ZC706	ZC702	ZC706	ZC702	ZC706	PYNQ board	ZC706	
Frequency (MHz)	150	166	200	200	143	200	200	200	
Precision	Activation:16-bit fixed point Weight:8-bit fixed point		Activation:2-bit fixed point Weight:1-bit fixed point		All 1-bit fixed point		All 8-bit fixed point	All 8-bit fixed point	All 4-bit fixed point
DSP Usage	1518[1]	793	89	84	3	31	220[2]	818	53
On-chip RAM Usage	1900	652	105.5	384	94	112	280[2]	708	416
Logic Usage (kLUT)	161	122.5	44	55.5	46.9	51.3	53.2[2]	100.2	163.5
Throughput (GOPS)	645.25	167.58	410.2	405.82	207.8	441.95	7.2[2]	124.9	276.4
DSP EFF (GOPS / DSP)	0.213	0.211	-	-	-	-	-	0.153	-
Logic EFF (GOPS / kLUT)	-	-	9.323	7.312	4.431	8.615	-	-	1.691
Power (W)	21.2	6.08	2.26	5.6	4.7	4.88	-	7.31	5.22
Energy EFF (GOPS/W)	30.44	26.68	181.5	72.47	44.2	90.56	-	17.09	52.96

Table 4: Experimental results of CRNN [16] models on TX1.

Precision	Tool	Throughput (GOPS)	Power (W)	Energy Efficiency (GOPS/W)	Accuracy
1-bit	Zac	1581.4	10.7	147.79	48.8%
16-bit	Zac	135.2	11.1	12.18	78.1%
	TensorRT	156.4	11.0	14.22	
32-bit	Zac	87.4	10.8	8.09	80.8%
	TensorRT	103.1	11.1	9.29	

similar power, the 1-bit and 16-bit models achieve 18.09X and 1.55X throughput improvements, respectively. Compared with TensorRT designs, the performance results are slightly lower since TensorRT performs graph-level fusion optimization. However, TensorRT is unable to generate a design for the 1-bit model while Zac does.

6 RELATED WORKS

Deploying DNNs on embedded devices. Quantization is an essential optimization knob for deploying DNNs on embedded devices. Efforts have been made to manually deploy these quantized models on hardware platforms. [18, 23] design architectures to accelerate 1-bit XNOR-Net [13]. [4] focuses on DoReFa-Net [24] where 1-bit and 2-bit data are used to represent weights and activation, respectively. There are also works [5, 11, 20] targeting other quantized models. However, they target only one precision or model.

Automated tools for deploying DNNs. To serve fast-changing DNN models and various platforms, automated tools are proposed. [2, 19] generate code for both GPU and CPU platforms, while TensorRT targets GPUs. However, none of them provide comprehensive optimization for quantized DNNs as Zac does. For FPGA platforms, some works [2] generate instructions for specialized hardware, while others [21, 22] instantiate templates. However, all these previous works are hard to change employed data types or customize basic operations accordingly.

7 CONCLUSION

In this paper, to address the massive computations and data, we propose Zac to automatically optimize and deploy DNNs on embedded devices. In Zac, we develop quantization-specific passes which utilize low precision data to improve throughput and energy efficiency. First, we propose tensor expressions as a unified cross-platform programming interface. Then based on these expressions, Zac performs type conversion, data type optimization, and expression customization to boost performance while minimizing extra errors. Experiments demonstrate the generated designs are comparable to state-of-the-art manually optimized designs. The case study using text recognition application shows that low precision designs generated by Zac achieve up to 19.18X throughput and 25.44X energy efficiency compared with full precision designs.

[1] One DSP slice of Nallatech platform is equivalent to two DSP slices of ZC706 platform when performing 16×8 fixed-point multiplications.

[2] The resource utilization and throughput value are not reported in [2]. We list the total resources of the used FPGAs and estimate the value from its latency breakdown figure.

8 ACKNOWLEDGEMENT

This work is supported by Beijing Natural Science Foundation (No. L172004), Municipal Science and Technology Program under Grant Z181100008918015.

REFERENCES

[1] Hande Alemdar and et al. 2017. Ternary neural networks for resource-efficient AI applications. In *IJCNN*.
[2] Tianqi Chen and et al. 2018. TVM: End-to-End Optimization Stack for Deep Learning. In *SysML*.
[3] Kaiming He and et al. 2016. Deep residual learning for image recognition. In *CVPR*.
[4] Li Jiao and et al. 2017. Accelerating low bit-width convolutional neural networks with embedded FPGA. In *FPL*.
[5] Liqiang Lu, Yun Liang, Qingcheng Xiao, and Shengen Yan. 2017. Evaluating fast algorithms for convolutional neural networks on fpgas. In *FCCM*.
[6] Yufei Ma and et al. 2017. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In *FPGA*.
[7] Ramon E Moore, R Baker Kearfott, and Michael J Cloud. 2009. *Introduction to interval analysis*.
[8] Anshuman Nayak and et al. 2001. Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs. In *DATE*.
[9] Young H. Oh and et al. 2018. A Portable, Automatic Data Qantizer for Deep Neural Networks. In *PACT*.
[10] Adam Paszke and et al. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.
[11] Adrien Prost-Boucle and et al. 2017. Scalable high-performance architecture for convolutional ternary neural networks on FPGA. In *FPL*.
[12] Shankar Ganesh Ramasubramanian and et al. 2014. SPINDLE: SPINtronic deep learning engine for large-scale neuromorphic computing. In *ISLPED*.
[13] Mohammad Rastegari and et al. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*.
[14] Raphael Ernani Rodrigues, Victor Hugo Sperle Campos, and Fernando Magno Quintao Pereira. 2013. A fast and low-overhead technique to secure programs against integer overflows. In *CGO*.
[15] H. Sharma and et al. 2018. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network. In *ISCA*.
[16] Baoguang Shi, Xiang Bai, and Cong Yao. 2017. An End-to-End Trainable Neural Network for Image-Based Sequence Recognition and Its Application to Scene Text Recognition. *TPAMI* (2017).
[17] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv* (2014).
[18] Yaman Umuroglu and et al. 2017. Finn: A framework for fast, scalable binarized neural network inference. In *FPGA*.
[19] Nicolas Vasilache and et al. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv* (2018).
[20] Xuechao Wei and et al. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *DAC)*.
[21] Qingcheng Xiao and et al. 2019. Fune: An FPGA Tuning Framework for CNN Acceleration. *IEEE Design & Test* (2019).
[22] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. 2017. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs. In *DAC*.
[23] Ritchie Zhao and et al. 2017. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *FPGA*.
[24] Shuchang Zhou and et al. 2016. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv* (2016).