electronics

Article

MobileNets Can Be Lossily Compressed: Neural Network Compression for Embedded Accelerators

Se-Min Lim and Sang-Woo Jun *

Computer Science Department, Donald Bren School of Information and Computer Sciences, University of California, Irvine, CA 92697, USA; seminl1@ics.uci.edu
* Correspondence: swjun@ics.uci.edu

Abstract: Although neural network quantization is an imperative technology for the computation and memory efficiency of embedded neural network accelerators, simple post-training quantization incurs unacceptable levels of accuracy degradation on some important models targeting embedded systems, such as MobileNets. While explicit quantization-aware training or re-training after quantization can often reclaim lost accuracy, this is not always possible or convenient. We present an alternative approach to compressing such difficult neural networks, using a novel variant of the ZFP lossy floating-point compression algorithm to compress both model weights and inter-layer activations and demonstrate that it can be efficiently implemented on an embedded FPGA platform. Our ZFP variant, which we call ZFPe, is designed for efficient implementation on embedded accelerators, such as FPGAs, requiring a fraction of chip resources per bandwidth compared to state-of-the-art lossy compression accelerators. ZFPe-compressing the MobileNet V2 model with an 8-bit budget per weight and activation results in significantly higher accuracy compared to 8-bit integer post-training quantization and shows no loss of accuracy, compared to an uncompressed model when given a 12-bit budget per floating-point value. To demonstrate the benefits of our approach, we implement an embedded neural network accelerator on a realistic embedded acceleration platform equipped with the low-power Lattice ECP5-85F FPGA and a 32 MB SDRAM chip. Each ZFPe module consumes less than 6% of LUTs while compressing or decompressing one value per cycle, requiring a fraction of the resources compared to state-of-the-art compression accelerators while completely removing the memory bottleneck of our accelerator.

Keywords: embedded FPGA accelerators; compression; neural networks

Citation: Lim, S.-M.; Jun, S.-W. MobileNets Can Be Lossily Compressed: Neural Network Compression for Embedded Accelerators. *Electronics* **2022**, *11*, 858. https://doi.org/10.3390/electronics11060858

Academic Editor: Maciej Ławryńczuk

Received: 1 February 2022
Accepted: 1 March 2022
Published: 9 March 2022

1. Introduction

A great amount of research effort has focused on bringing the benefits of deep neural networks to resource-constrained embedded devices in order to support increasingly popular applications, such as the internet of things (IoT) or cyber–physical systems (CPS). Efforts include architectural solutions, including the use of power-efficient, high-performance accelerators, such as mobile GPUs [1] or FPGAs [2,3], as well as algorithmic solutions, such as designing compact neural networks with fewer resource requirements [4–6].

An imperative technology for embedded neural networks is deep compression [7], where a complex neural network can be compressed by *pruning* less significant edges, as well as *quantizing* weights and activations from floating-point values to smaller fixed-point values. Quantizing is an especially important technique. Smaller fixed-point values can not only reduce the memory size and bandwidth requirements, but also replace costly floating-point arithmetic with simpler fixed-point ones. Many important neural network models demonstrated maintaining high accuracy with aggressive quantization to 8-bit integers and beyond [8], and many hardware and software environments are optimized for such quantized models [9–12].

However, not all models can be quantized effectively, and aggressive quantization sometimes results in an unacceptably sharp accuracy decline. A prominent example is

the MobileNet series of neural networks, which use depth-wise separable convolutions to construct compact but powerful networks [4]. The post-training quantization of MobileNets and other similarly structured neural networks have often resulted in a sharp drop in accuracy from 90% to almost 1%, an unacceptable decline [4,13]. The accuracy of these models can be reclaimed to uncompressed levels, using quantization-aware training or by re-training the quantized model [14]. However, it is not always possible or convenient for the user to fine-tune or re-train the model after obtaining it. For example, re-training may be difficult if the training data are unavailable due to legal or privacy issues.

To address these problems, we demonstrate a method of compressing neural network models as well as inter-layer activations, using a hardware-optimized error-bounded lossy floating-point compression algorithm. Compressing neural network weights with lossy algorithms, such as SZ, has shown to retain good accuracy, even with aggressive compression [15], but applying them to embedded accelerators is difficult due to the high resource requirements of such complex algorithms, even when implemented on hardware accelerators [16,17]. We solve this issue by modifying the ZFP error-bounded lossy floating-point compression algorithm [18] to support efficient hardware implementation without a significant increase in error. We call our algorithm ZFPe, and its FPGA implementation is capable of running at 100 MHz, delivering 400 MB/s of throughput while consuming only 2800 (6%) slices on the Lattice ECP5-98F FPGA, a low-power FPGA chip.

We demonstrate our approach using a custom-built, best-effort convolutional neural network (CNN) accelerator designed to use ZFPe. Our accelerator uses an output stationary caching method [19] to efficiently handle large layers, as well as implement simplified, platform-optimized floating-point units to improve resource efficiency. We implement this accelerator on a realistic embedded processing platform, using the open-source ULX3S FPGA development board equipped with a low-power Lattice ECP5-85F FPGA and a 32 MB SDRAM chip. No other lossy floating-point compression accelerator is able to support this level of performance within the chip budget limitations. On this platform, we demonstrate that the compression efficiency of ZFPe directly translates to improved performance by mitigating the memory bottleneck. In fact, on this platform, we are able to completely remove the memory bottleneck with a bit budget of 5 or less per weight and activation.

More importantly, we demonstrate superior accuracy retention on our target neural network MobileNet V2 on the Imagenet dataset [20], compared to a post-training quantized model. Our approach with a bit budget of 12 bits per value is able to maintain full top-5 accuracy (90%) compared to the uncompressed model. It also achieves a competitive 65% accuracy with a bit budget of 8 bits, which is still significantly higher than the results with naive post-training quantization [21].

Our contributions are as follows:

- We present and evaluate ZFPe, a modified ZFP compression algorithm and hardware implementation optimized for efficient embedded hardware implementation.
- We present an alternative neural network compression approach using ZFPe, and demonstrate its relative effectiveness, even on models traditionally difficult to quantize with post-training quantization.
- We evaluate the performance impact of ZFPe in the context of embedded neural network acceleration.

The rest of this paper is organized as follows: Background and related works are explored in Section 2. We describe the ZFPe algorithm and its hardware-optimized nature in Section 3. We present the architecture of our ZFPe-enabled embedded neural network accelerator in Section 4. We provide an in-depth evaluation of our approach and implementation in Section 5, and conclude with future work in Section 6.

2. Background and Related Works

2.1. Embedded Neural Network Acceleration

Due to the computation-intensive nature of neural networks, the use of power-efficient hardware accelerators is one of the most prominent methods of bringing the benefits

of deep neural networks to resource-constrained embedded systems. Embedded neural networks can reduce decision-making latency by removing the requirement for querying a remote server [22,23], as well as reducing power-hungry wireless network transmission requirements [24]. Some acceleration approaches include using mobile GPUs [1], custom-designed application-specific integrated circuits (ASICs) [25–28], as well as FPGAs [2]. The FPGA acceleration of embedded neural network acceleration is of special interest since it can bring together the power performance benefits of dedicated circuits and the capability to deploy microarchitectures optimized for the target neural network model [29–34]. FPGA neural network accelerators demonstrated orders of magnitude higher power efficiency compared to general purpose processing units when applied to complex networks, such as image or video recognition [35,36]. Deploying simple neural networks on general-purpose microcontrollers is also a widely researched topic, but their goals differ from accelerators, due to the more stringent resources and performance limitations of embedded microcontrollers [6,37].

2.2. Neural Network Compression

The deep compression of neural networks is another key component of embedded neural network acceleration [7]. Typically, deep compression involves *pruning*, which removes less significant edges from the network, and *quantization*, which maps the weights and inter-layer activations to a smaller number of typically fixed-point bits. Deep compression can reduce the size of the model, reducing the memory capacity and bandwidth requirements, as well as reducing the computation requirements by replacing costly floating-point operations with simpler fixed-point ones [38–41]. There are many different approaches to efficiently compress neural network models without too much accuracy degradation [14,15,41,42]. Typically, pruned network accuracy suffers a sharp accuracy degradation unless the pruned network structure is *re-trained* with the training data [14,41,43]. Quantization suffers less from this problem, and there are many different approaches, including simply quantizing weights after training (*post-training* quantization), re-training after quantization, and more [44–46]. Quantization can either be applied to weight values, inter-layer activation values, or both. Values can be quantized from the typically 4-byte floating-point values to 8-bit integers [47], or even more aggressively to ternary [48,49] or binary [50,51] values with varying accuracy loss trade-offs. Quantization is typically more readily available on embedded neural networks compared to pruning since pruning can introduce sparsity in the weights, resulting in complex random access [43,52].

2.3. Quantization Effectiveness on MobileNets

One interesting characteristic of deep compression is that post-training quantization works poorly on our target class of models designed for embedded applications, including the popular MobileNet class of models [4,13]. These networks use depth-wise separable convolutions to construct compact models with accuracy comparable to much larger networks. However, post-training quantization on these models results in an unacceptably sharp decline in accuracy [21], dropping from 90% or better to 1% or worse on the ImageNet dataset. Accuracy can be reclaimed using various methods, including re-training and quantization-aware training [53,54], but this is not always possible or convenient if the newly required computation requirement or expertise is high, or if the training data are unavailable due to legal or privacy issues. We posit that error-bound lossy compression algorithms may be an alternative, accuracy-preserving method of compressing depth-wise separable models.

2.4. Floating-Point Compression

In this work, we use error-bounded lossy compression algorithms for floating-point data instead of traditional quantization approaches, and present a compression algorithm optimized for embedded FPGAs. Lossy algorithms, such as SZ [55–57] and ZFP [18,58,59], operate with the knowledge of floating-point data encoding and can achieve orders of magnitude compression of floating-point data, given an error margin acceptable by the

target application. These algorithms are widely used to reduce the storage capacity requirements of scientific data archiving [60], as well as to reduce intermediate data movement for iterative scientific computing algorithms. In more detail, SZ uses preceding neighbors in the multidimensional space to predict the value of each data point and compresses the data size by an error-controlled quantization and customized Huffman coding. In the case of ZFP, it divides the whole data into 4^d value blocks (d means dimension) and performs the compression in each block separately. We select ZFP for the compression method because, to date, there are more studies that exploited ZFP or variants [17,61] derived from ZFP, compared to those that used SZ. In addition, to the best of our best knowledge, a lossy compression of neural networks using ZFP has not been studied yet, compared to DeepSZ [15], which uses SZ for the compression scheme.

While it has been shown that error-bounded lossy compression applied to neural network models can achieve high compression relative to accuracy degradation [15], the complexity of these algorithms prevents them from being used in the high-performance datapath for real-time weight and activation compression and decompression, even with hardware acceleration. This is especially true for resource-constrained embedded accelerators, due to their high resource requirements [16,17]. Furthermore, their application to activation compression has also yet not been explored.

For example, the best existing effort on lossy compression of neural networks, DeepSZ [15], employs an FPGA implementation of the SZ compression algorithm, which consumes almost 20,000 Arria ALMs for a single pipeline implementation [16]. As we emphasize in Section 5, this is a prohibitively large number for the embedded FPGAs we target.

3. ZFPe: A Novel Lossy Floating-Point Compression for Embedded Accelerators

In this work, we compress neural network model weights as well as dynamically generated inter-layer activation values, using a novel variant of the ZFP error-bounded lossy floating-point compression algorithm. The novel algorithm, which we call ZFPe, modifies the *embedded coding* step of the ZFP algorithm to enable high performance and low on-chip resource utilization for a small loss in the compression ratio. We first introduce the original ZFP algorithm and its performance bottleneck in a hardware implementation and then describe the ZFPe algorithm in detail.

3.1. Original ZFP Algorithm Analysis

The ZFP algorithm performs the compression and decompression of single- or double-precision floating-point numbers in units of 4^d value *blocks*, where d is the operating dimension ranging from 1 to 4. For example, $1D$ ZFP compresses four values at a time, and $4D$ ZFP compresses 256 values at a time. Intuitively, ZFP operates by extracting common factors, such as exponents, from a block of values. As a result, larger block configurations typically result in more effective compression. The algorithm also includes a block transformation step similar to JPEG, which makes such extraction more effective.

The ZFP algorithm can operate in one of two modes, either to limit the amount of tolerable error during compression or to limit the total number of bits used per floating-point value.

Block compression consists of four stages:

1. Fixed-point conversion: All values in the block are aligned to the maximum exponent in the block and converted to the fixed point.
2. Block transformation: A series of simple convolution operations are applied to the block in order to spatially de-correlate values [18]. The resulting block is statistically transformed such that most integers turn into small signed values clustered around zero. This is similar to the discrete cosine transform used by JPEG.
3. Deterministic reordering: Values in a block are shuffled according to a predetermined sequence, which results in values in a roughly monotonically decreasing order. This stage is not necessary for $1D$ compression.

4. Embedded coding: The *group testing* algorithm is used to encode each block in a smaller number of bits. It encodes one *bit plane* at a time in the order of significance until either the error tolerance bound is hit or all the provided bit budget is consumed.

Where stages one to three can be parallelized in a straightforward way in hardware, the final embedded coding step of the original ZFP algorithm is the prominent performance bottleneck for hardware implementation due to a tight feedback loop in the algorithm.

The embedded coding stage in the original ZFP algorithm encodes blocks one *bit plane* at a time, starting from the most significant bit plane. A bit plane is constructed by collecting bits from the same position of all values in a block, where the N-th bit plane consists of N-th bits of each element in a block. For a d-dimensional block with 32-bit values, there are 32 bit planes, each with 4^d bits.

The compression efficiency of ZFP comes from this embedded coding scheme. First of all, bit planes are encoded in the order of significance until the specified error bound or bit budget limitations are met. As more significant bit planes have more impact on accuracy, this is an efficient way to encode values. Furthermore, because values in a block are de-correlated and reordered to roughly decreasing order, many of the most significant bits in each bit plane are expected to be zeros, especially for earlier, more significant bit planes. To exploit these characteristics, the embedded coding algorithm of ZFP aims to compactly encode the small number of nonzero bits in the least significant portion of each bit plane.

Algorithm 1 presents the original ZFP's group testing scheme in the embedded coding stage. For ease of understanding, we note that it omits one algorithmic feature, where it exploits the ordered nature of the block by emitting some least-significant bits of each bit plane verbatim without flag bits, depending on the number of non-zero bits in the previous bit plane.

Algorithm 1: ZFP's embedded coding stage emits bits one by one from each bit plane.

```
Data: 4^d-bit bit plane
while bitplane! = 0 do
    emit 1 ;                                        ▷ Emit ''not done'' bit
    while True do
        lsb ← bitplane[0];
        emit LSB ;                                        ▷ Emit data bits
        RightShift(bitplane, 1);
        if lsb == 1 then
            break ;                                  ▷ Repeat group testing
        end
    end
end
emit 0 ;                                                ▷ Emit ''done'' bit
```

As shown in Algorithm 1, the *group testing* method in ZFP encodes one least significant bit at a time while checking if the remaining *bit plane* is zero after emitting each bit. If the remaining bit plane is zero, it emits a single bit *"done"* flag, **0** is emitted, and the algorithm moves on to the next bit plane. If the remaining bit plane is still not zero, it emits a *"not done"* flag, **1**, and emits *data bits* one by one in the order of least significance until a data bit of **1** is encountered and emitted. Once a nonzero data bit is emitted, the group testing process is repeated with the remaining bits. For example, a 4-bit bit plane with a value of binary "0011" is encoded as $\underline{1}1\,\underline{1}1\,\underline{0}$, where the underlined bits are flag bits.

The performance bottleneck of hardware implementations of the ZFP algorithm is in this embedded coding stage. ZFP encodes data one *bit plane* at a time, and the number of bits required to encode each bit plane can only be determined after encoding is done.

This means the encoding and decoding of multiple bit planes cannot be parallelized, as the algorithm cannot know where the next encoded bit plane starts before the current one is processed. This makes efficient hardware implementation different because even in the best case, only d bits can be encoded or decoded per cycle, where d is the dimensions of the block.

This is especially limiting for FPGAs, which typically have lower clock speeds, compared to ASICs or CPUs. For example, a $1D$ ZFP decompressor running at 100 MHz has a best-case throughput of 50 MB/s, which is not sufficient to support dynamic compression and decompression at rates required by neural network accelerators. This problem is exacerbated by the fact that the original ZFP algorithm encodes each bit plane one bit at a time, opting to improve compression ratios at the cost of performance.

3.2. Novel ZFPe Optimizations

Our proposed compression algorithm, ZFPe, solves the performance issue of ZFP on hardware accelerators by introducing a new encoding scheme. ZFPe encodes each *value* in a block independently instead of re-organizing values in a block into a series of bit planes.

Given a bit budget p per single-precision floating point value, ZFPe simply encodes the p most significant bits for each block-transformed integer value. Because each encoded value represents a single value in the block, encoding and decoding can always process one floating-point value per cycle, instead of waiting until all bit planes of the block are processed. Since specifying a bit budget is part of the ZFPe algorithm, it only supports the bit budget operation mode and does not support specifying error bounds.

While this approach is good for performance, it leaves some compression efficiency on the table because it does not take advantage of the typically many leading zeros in the most significant bits of the block-transformed values. To remedy this, ZFPe adds a one-bit flag per value to specify whether the n most significant bits of the values contain a nonzero bit. If a nonzero bit exists, ZFPe encodes p bits starting from the MSB. If a nonzero bit does not exist, ZFPe encodes p bits starting from the n-th bit from the MSB, which has the effect of encoding n more LSB bits within the same bit budget. Figure 1 shows this scheme being applied to two numbers, using $n = 4$ and $p = 5$. As described earlier, the value of p is chosen according to the bit budget requirements. The value of n is chosen by observing the nonzero bit distribution of the target dataset. We chose a value of $n = 4$ after observing the data distribution of many neural networks and scientific data distributions and discovered a sharp drop-off in the number of values with more than 4 most significant zero bits.

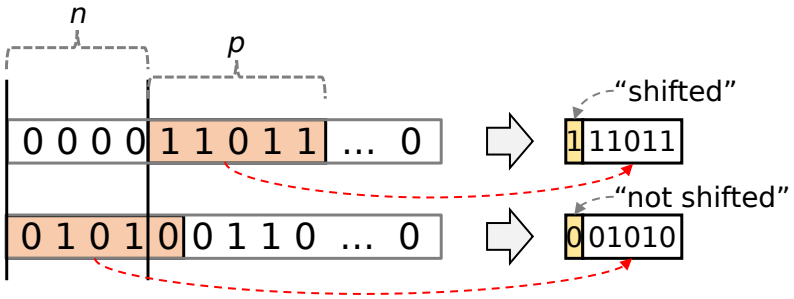

Figure 1. ZFPe encoding using $n = 4$ and $p = 5$.

Figure 2 compares the ZFPe scheme against the original ZFP approach, emphasizing the difference between ZFP's bit plane order and ZFPe's value order encoding. Both algorithms encode four 3-bit values in a $1D$ block, `110`, `100`, `011`, and `010`, shown in the left side of the figure. The bits highlighted in yellow are header bits, and it is readily visible that ZFPe uses much fewer headers.

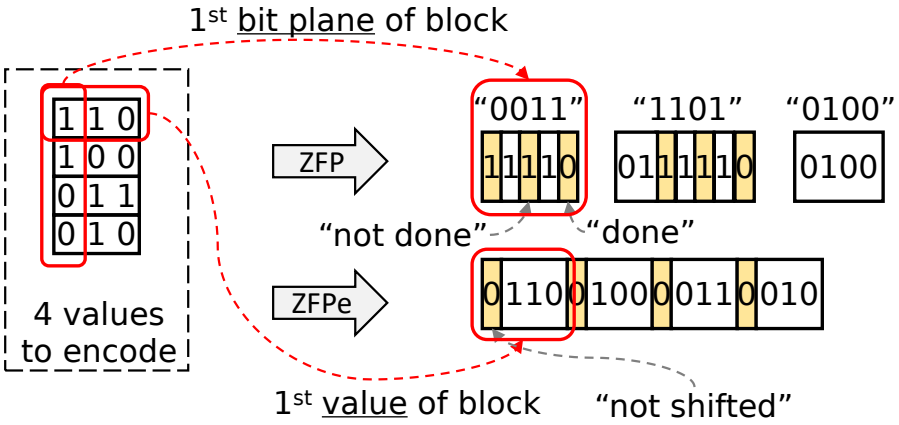

Figure 2. ZFP and ZFPe encoding three-bit planes of four values.

Considering that each header represents a dependency for encoding and decoding, we can expect the performance advantage of ZFPe. We note that the bit from the first value becomes the LSB of each bit plane, and that is why the first bit plane is listed as `0011` instead of `1100`.

For completeness, Figure 2 also shows the full effects of ZFP, including the verbatim bit emission mentioned in Section 3.1. For each bit plane, ZFP may emit some least significant bits verbatim before applying group testing. The number of bits that are emitted verbatim in the current bit plane is the number of non-zero bits in the previous bit plane. This is why ZFP encodes the third bit plane, as well as the first two bits of the second bit plane, without headers.

Figure 2 also shows the headers in the encoded bit stream, but all headers are zeros because $p < n$ in this simple example.

One characteristic of both ZFP and ZFPe that we have so far omitted is that for each block, the common exponent bit for all values must also be encoded. This is used to convert floating-point representation to fixed-point representation and vice versa. For ZFP and ZFPe, 9 bits are used to encode the exponent, as described in the original ZFP paper [18]. Since each encoded block invariably requires 9 bits to store the exponent before any data are encoded, the usable bit budget per float value (p) is actually smaller than the parameter value given by the user. For example, if the goal is to use 8 bits per float while using a $1D$ algorithm, the actual p value is $((8 \times 4) - 9)/4 = 5.75$. Since we cannot store bit fractions, our accelerator actually assigns 6 bits for the first three values and 5 bits for the last value to satisfy the bit budget.

4. Neural Network Accelerator Architecture

Figure 3 shows the overall architecture of the embedded neural network inference accelerator we constructed to evaluate our compression approach. As our target neural network models are too large to fit in the on-chip BRAM of low-power FPGAs, such as the Lattice ECP5, the neural network model, output activations from the previous layer, and the output of the current layer are all stored in off-chip memory, such as DRAM. All three data types are stored in a compressed form using ZFPe. Data input to the accelerator is decompressed on the fly using a pair of decompressors, and the output is compressed on the fly using a compressor. In order to make the best use of memory bandwidth, we also implement a burst memory arbiter between the compression cores and memory.

4.1. ZFPe Accelerator Architecture

To minimize on-chip resource utilization, our ZFPe design is a variant of the $1D$ ZFP, operating in units of four single-precision floating-point values. Higher-dimensional algorithms require more complex block transformation algorithms, which are not well suited for embedded accelerators. Figure 4 shows the internal pipelined architecture of our decompressor configured with a bit budget of 8 bits per float. The compression core is designed in the exact same fashion but in reverse order.

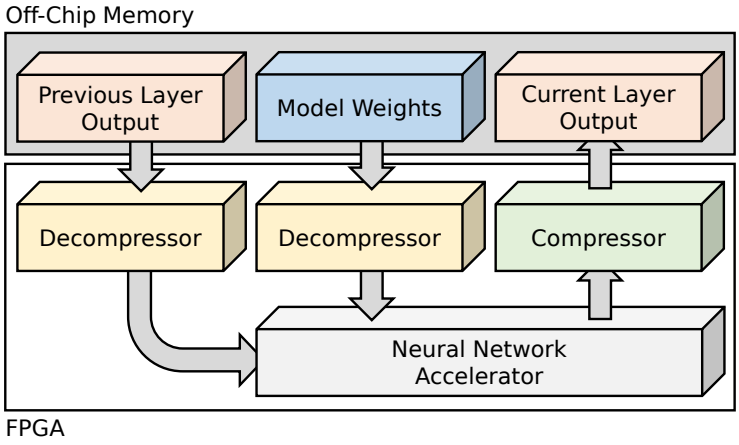

Figure 3. A diagram of overall architecture.

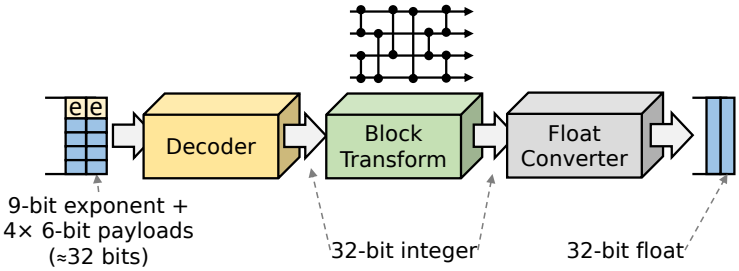

Figure 4. A ZFPe decompressor can always emit one float value every cycle.

The decoder takes up to one fixed-size compressed block as input per cycle, and if the compressed data rate is sufficient, it invariably emits one decompressed floating-point value per cycle. Since it takes four cycles to emit four decompressed float values, on average, the decompressor can ingest one compressed block per four cycles. As described in Section 3.2, in order to achieve an effective 8 bits per float while also encoding 9 bits of exponent per block, the first three values in the block are given a 6-bit budget, and the last one is given 5 bits, adding up to 32 bits per four floats.

Because ZFPe encoding has no dependencies between the four values in a block, the decoder can immediately serialize the encoded values to decode and emit one 32-bit value per cycle. This approach requires much fewer on-chip resources, compared to working on each bit-plane of all four values at once. The block transform layer must again collect all four values in a parallel group because the block transform stage involves a pipeline of convolution operations between pairs of values in the block, as illustrated in Figure 4, as a network resembling sorting networks. After the block transform stage, results can again be serialized for independent floating-point conversion.

4.2. Burst Memory Arbiter

Despite its name, the dynamic random access memory (DRAM) performance can degrade significantly with fine-grained random access because of the multi-cycle overhead of accessing a new *row* [62,63]. DRAM rows are multiple kilobytes in size, and the best performance can only be achieved if most of the bits for every newly opened rows are read and used. As a result, sequential access typically demonstrates higher performance compared to random. This is an especially important feature for embedded systems, where the memory may be under-provisioned and the efficient use of memory bandwidth is crucial for performance.

To achieve high memory performance, we implement a burst memory arbiter between the memory and the rest of the accelerator in order to take advantage of the sequential access patterns of each input and output stream. Each memory read or write request happens in 256-byte bursts, and burst reads are only issued whenever the on-chip read

buffer has enough space to accommodate all 256 bytes. While the design of our arbiter is not new, it is necessary to achieve optimal memory performance.

4.3. Neural Network Inference Accelerator

Our neural network inference accelerator is implemented over a configurable number of processing elements (PEs), each with floating-point multiply–accumulator (MAC) units. Each PE caches multiple working partial sums until each is completely calculated. Figure 5 shows the internal architecture of the accelerator, as well as the data movement to and from data in off-chip memory. The accelerator effectively implements a matrix–matrix multiply unit with intelligent caching, along with a selection of activation functions, such as ReLU.

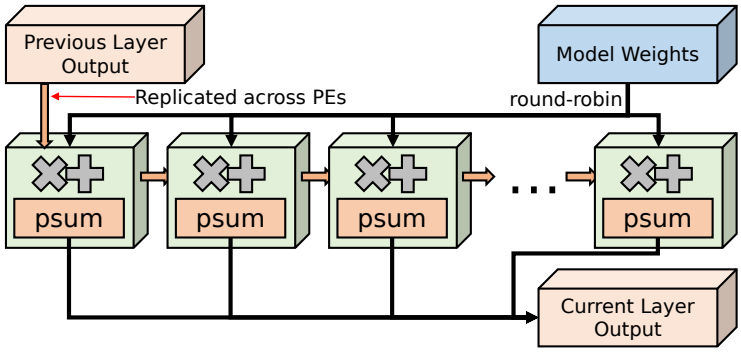

Figure 5. Output stationary accelerator architecture for large models.

Each PE uses a local buffer of multiple ongoing partial sums (*psum*) implemented using on-chip block RAM (BRAM), and each output is emitted, collected, and written to memory once all of the multiply–accumulate operations are completed for each output. Activations from the previous layer output are broadcast to all PEs over a chain of pipeline registers, and model weights are distributed to each PE in a round-robin fashion. In order to hide the high latency of floating-point MAC operations, each PE tries to have many operations in flight by working on multiple partial sums at once. We achieve this by working on batches of input, as well as multiple weight columns. To simplify memory access, input batches store their values in an interleaved fashion such that the simple sequential read from memory results in correct read ordering. The weight matrix is stored in a column-major order for similar simplicity of memory access, with the bias value stored at the end of each column. Figure 6 conceptually shows how our accelerator issues multiple MAC operations in parallel, with example input and weight column batch sizes of two.

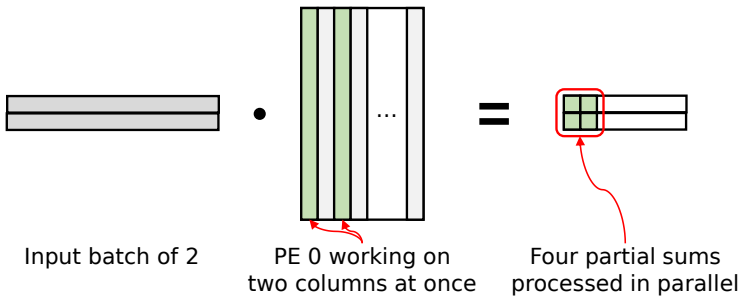

Figure 6. Output stationary accelerator architecture for large models.

4.4. Simplified Floating-Point Cores

Another part of our best effort to implement a performance-effective neural network accelerator platform is the use of resource-efficient, simplified floating-point MAC units. In the interest of saving resources, our MAC unit omits some of the special corner cases supported by the IEEE-754 standard, such as NaN, Infinity, or subnormal number support, instead of handling all of them as zeros. We also optimize the MAC unit to our target platform of Lattice ECP5 FPGAs, which includes an 18 $\times$ 18 multiplier DSP block, by

only supporting 18-bit fractions. During multiplication or addition, each 18-bit fraction is expanded to 36 bits, computed, then truncated back to 18 bits. Our benchmarks show that due to the loss in fractional precision already introduced by compression, these simplifications do not have much impact on accuracy.

5. Evaluation

In this section, we present the evaluation of our ZFPe approach to neural network acceleration first by comparing the raw compression efficiency against unmodified ZFP and then evaluating the MobileNet inference accuracy compared to ZFP and post-training quantization. We also present an accuracy and performance efficiency evaluation compared to the state-of-the-art neural network compression accelerator DeepSZ [15] in Section 5.6.

5.1. Implementation Detail

We implement our ZFPe-augmented neural network accelerator on the ULX3S off-the-shelf embedded acceleration prototyping board [64]. The ULX3S board is a realistic prototyping environment for embedded acceleration, equipped with a low-power Lattice ECP5-85F FPGA [37], as well as a 32 MB on-board SDRAM chip. We use the open-source toolchain Yosys [65] to program the FPGA chip and collect resource utilization statistics. Our neural network accelerator is able to fit eight PEs on the ECP5-85F chip along with two decompressors, one compressor, and the *Shell*, which provides platform logic, including the memory arbiter and host-side UART. All designs, including the DRAM controller, are clocked at 100 MHz. Table 1 shows the detailed breakdown of on-chip resource utilization collected by Yosys.

Table 1. On-chip resource utilization breakdown.

Module	Slice (%)	DP16KD BRAM (%)	MULT18X18D Multiplier (%)
1× PE	2.7 K (6%)	11 (5%)	1 (0.5%)
1× Decompressor	2.8 K (6%)	0 (0%)	0 (0%)
1× Compressor	3 K (7%)	0 (0%)	0 (0%)
Shell	2 K (5%)	3 (1%)	0 (0%)
Total	32 K (76%)	91 (43%)	8 (5%)

Our accelerator implementation has comparable levels of efficiency compared to state-of-the-art FPGA implementations. The MAC pipeline consumes around 700 slices of each PE, which is very similar to the well-optimized published implementations [66,67].

The single SDRAM chip is capable of reading 16 bits every five cycles in the best-case scenario, which translates to 40 MB/s of best-case bandwidth. The performance can be halved if the access pattern does not make efficient use of the row buffers.

5.2. ZFPe Compression Efficiency Evaluation

We first evaluate the error characteristics of ZFPe, compared to the original ZFP algorithm [18,59]. At this point, we are not yet applying ZFPe to the neural network inference, simply evaluating its efficiency on floating-point data. For this purpose, we use five floating-point datasets, including three neural network weight matrices and two scientific datasets from SDRBench (Scientific Data Reduction Benchmarks) [60,68]. For the neural network weights, we extract weight matrices from Keras [69] pre-trained model for ImageNet [20]. Details of each dataset are listed below:

- Fully connected layer no. 2 of VGG19 [70];
- Fully connected layer of Inception V3 [71];
- Fully connected layer of MobileNet V2 [13];
- Climate simulation dataset (CESM-ATM);
- Molecular dynamics simulation dataset (EXAALT).

We obtain the average error values by three steps: compressing each dataset at first, decompressing, and taking the mean of the error magnitude for all data points. Figure 7 shows the average error values, in terms of absolute value, of ZFPe and the original ZFP on each bit budget usage per single-precision floating point value. Although the original ZFP has a slightly lower error rate than ZFPe for lower bit budgets, the saturation points are almost the same. In the VGG19 average error result, there is nearly no difference between ZFPe and ZFP, and ZFPe even shows less loss at one bit budget per one float value in the case of the EXAALR dataset.

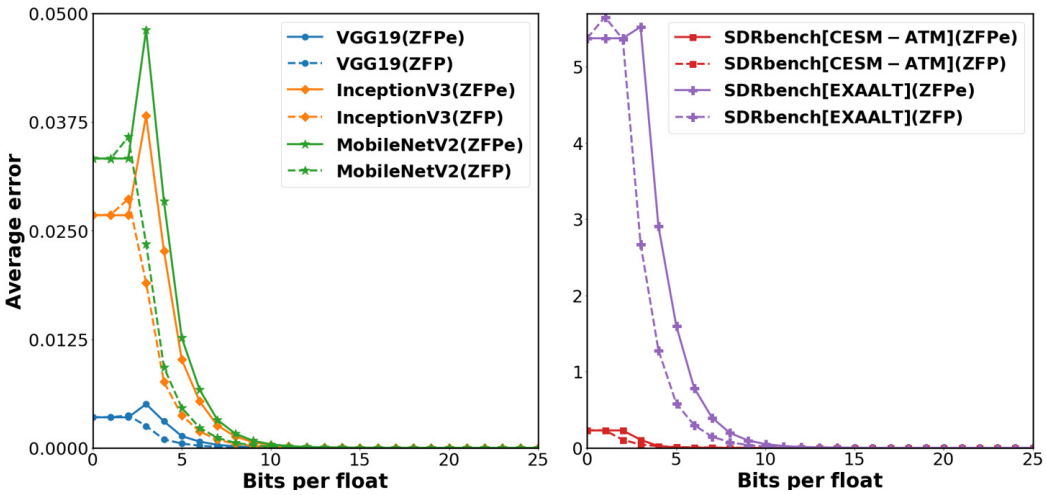

Figure 7. ZFPe shows comparable error rates compared to original ZFP across scientific and neural network datasets.

5.3. Neural Network Accuracy—Single Dense Layer

We first evaluate the accuracy impact of our ZFPe approach by evaluating its accuracy impact on the dense layers of two realistic neural networks: VGG19 [70] and MobileNetV2 [13]. We use the verification set from the popular ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2017. Due to the capacity limitations of the on-board SDRAM chip, we first only focus on the dense layers of the two models. We extract the dense layers from the two models from the Keras 2.4.3 [69] pre-trained neural network library. We also extract the input activations to these layers by running the two models on the Imagenet dataset and extracting the inter-layer activation matrices. For ZFPe, we evaluate the accuracy impact by loading the accelerator with all of the weight matrices and streaming in the extracted input data in phases. For ZFP, we augment Keras to compress and decompress the neural network models and the inter-layer activations, using a Python binding for the $1D$ ZFP.

Figure 8 shows the top-5 and top-1 accuracy of ZFP and ZFPe-augmented neural networks, with varying bit budgets per float. We see that ZFPe demonstrates similar accuracy compared to the original ZFP. For top-5 accuracy, both algorithms demonstrate less than 10% point loss of accuracy at 5 bits per float and no accuracy loss at 8 bits per float and beyond. Top-1 accuracy suffers a slightly steeper degradation but still shows a similar trend.

We note that the state-of-the-art lossy neural network compression accelerator DeepSZ [15] achieves almost lossless accuracy at a much smaller bit budget but at a much higher chip resource utilization. We present a more detailed comparison in Section 5.6.

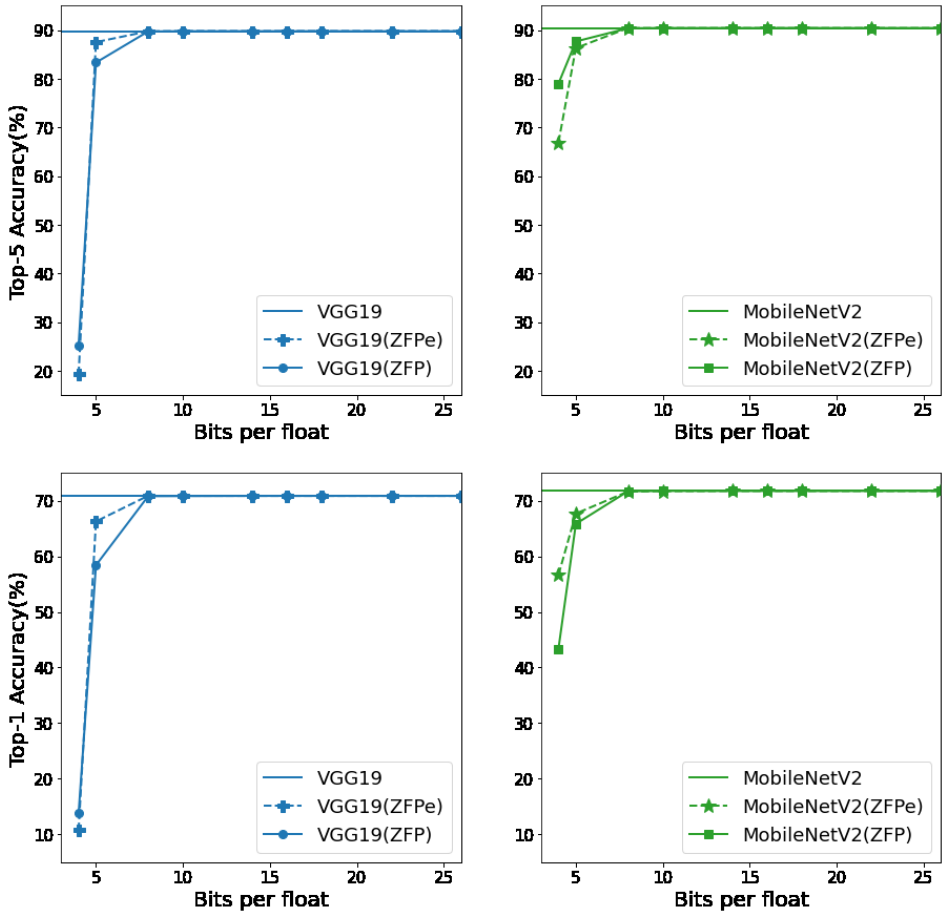

Figure 8. Top-5 and top-1 accuracy of the original and compressed VGG19 and MobileNetV2.

5.4. Neural Network Accuracy—Complete MobileNet V2

In order to determine whether our ZFPe approach can solve the accuracy problem of post-training quantization of MobileNets, we evaluate our approach on the end-to-end MobileNet V2 neural network. Weights for all layers are compressed using ZFPe, and output activations from each depthwise separable convolution layer are compressed and then decompressed using ZFPe before being read by the next layer. Due to the memory capacity limitations of our hardware platform, we perform the end-to-end evaluations on a software simulator and measure the impact on accuracy.

Figure 9 shows the accuracy impact of the conventional 8-bit post-training quantization (`INT8`) and our ZFPe approach with 8-, 10-, and 12-bit budgets per float, compared to the uncompressed model (`Float`). We present both the top-1 and top-5 accuracy results, as they are the most commonly used accuracy metrics. We can see that ZFPe-8 achieves significantly better accuracy compared to `INT8`, despite the same bit–width constraint for both weights and activations. In fact, our approach shows no loss in accuracy at a bit budget of 12 bits per float in both cases, demonstrating that our approach is a viable alternative for the post-training quantization of MobileNets. We note that retraining can reclaim almost full accuracy, even for MobileNets [46]. Our approach is targeted for the scenario where retraining is not feasible due to security or other issues.

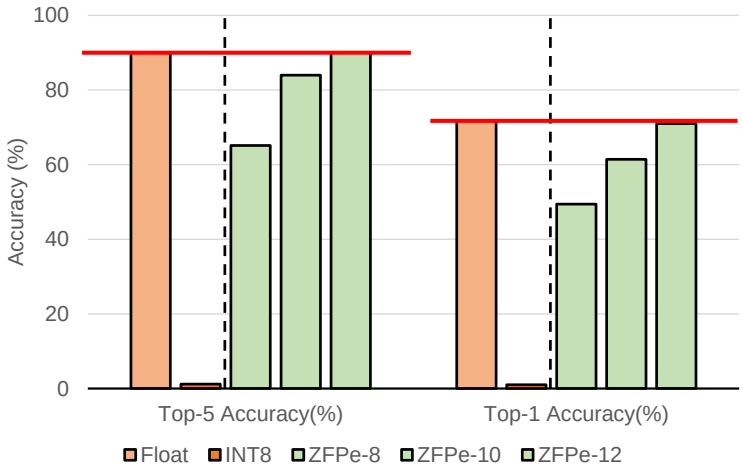

Figure 9. Top-5 and top-1 accuracy on MobileNet V2.

The state-of-the-art DeepSZ [15] accelerator does not list a MobileNet evaluation, but we present a projected performance efficiency comparison in Section 5.6.

5.5. Accelerator Performance Evaluation

Even if our approach can achieve high accuracy, it would not be very useful if the ZFPe hardware implementation is not fast enough to keep the on-chip processing elements busy. This is why even though the original ZFP algorithm and DeepSZ [15] demonstrates better error bound characteristics, compared to ZFPe, the low performance per chip resource characteristics prevent it from being useful in our target scenario.

Table 2 compares the performance of ZFPe against the two most prominent lossy floating-point compression accelerators, ZFP [17] and GhostSZ [16]. Thanks to the low resource utilization of ZFPe, it can achieve much better performance efficiency compared to existing works. As Altera ALM and Lattice Slice are both structured around two 4-LUTs, we believe the comparison of the direct numbers can provide a tolerable estimate.

This is a very important goal of ZFPe design, as resource-intensive algorithms simply cannot be deployed on resource-constrained embedded FPGAs like the ones we target. The difference in design is actually more pronounced than presented; ZFPe runs on the slower 100 MHz clock provided by our target embedded FPGA. The performance efficiency will improve further on server-grade FPGAs used by the comparison systems.

When provided with a steady stream of compressed data, our ZFPe decompressor is able to invariably emit one single-precision floating-point value per cycle. Similarly, the compressor is able to invariably ingest one floating-point value per cycle. Since our accelerator is clocked at 100 MHz, this translates to 400 MB/s of uncompressed throughput, which is impressive, considering the mere ~3000 slice utilization per module.

Table 2. Performance efficiency comparison of compression accelerators.

Design	Clock Speed (MHz)	Throughput (MB/s)	Slice/ALM	Efficiency (MB/s/KSlice)
ZFP [17]	200	4000	150 K	26.6
GhostSZ [16]	200	800	19 K	42.1
ZFPe **(Ours)**	100	400	3 K	**133.3**

We measure the performance impact of ZFPe during a single fully connected layer evaluation using the second fully connected layer from VGG19, which has a 4096×4096 weight matrix and 4096 bias values. The accelerator is configured with a batch size of 64, and each PE is configured to process 16 weight columns in parallel, meaning each PE concurrently works on the partial sums of 1024 output values. The batched operation

results in high enough data reuse such that, coupled with compression, the bandwidth requirement is kept at manageable levels.

Figure 10 compares the performance of a baseline accelerator and one augmented with ZFPe configured with varying bit budgets per float. Performance is presented in terms of PE utilization, measuring the percentage of cycles in which new MAC operations are inserted into the MAC pipeline. Utilization of 100% means we are achieving the peak performance attainable by the hardware. We see that performance increases proportionally with smaller bit budgets, as we mitigate the memory bandwidth bottleneck using ZFPe. In fact, we are able to completely saturate all of the MAC modules in all PEs with a bit budget of 5 or less.

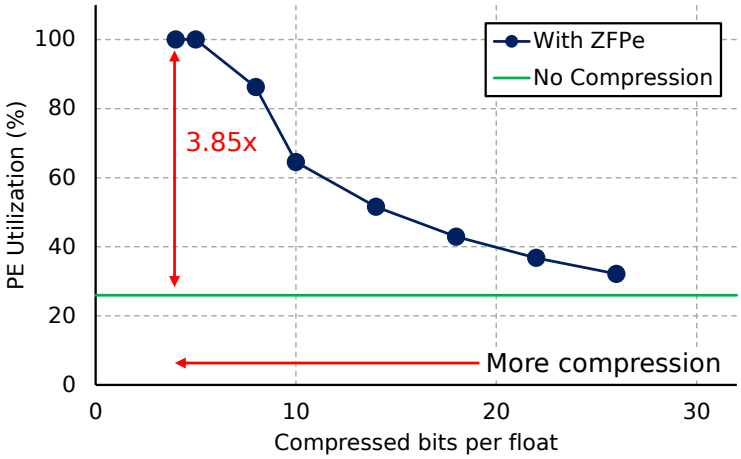

Figure 10. Performance impact of ZFPe acceleration, with smaller bit budgets to the left.

We note that this evaluation uses an input batch size of 64, and while performance will decrease if the input batch size needs to become smaller due to application requirements, the relative performance benefits of ZFPe will stay the same. This is because memory reuse decreases with smaller input batches, moving the performance bottleneck back to the slow memory bandwidth. On the other hand, we can achieve higher performance with smaller batch sizes by using faster off-chip memory, and ZFPe will still be able to support the higher bandwidth. As we noted in Section 5.2, each decompressor is capable of supporting 400 MB/s of decompressed bandwidth, or 100 MB/s of raw memory bandwidth—adding up to 100 MB/s of memory bandwidth between two decompressors—leaving room to take advantage of potentially faster memory bandwidth. Memory writes are not likely to be a performance bottleneck since the output rate of output-stationary memory re-use is very low compared to the reads. If the memory bandwidth becomes too high for a single ZFPe accelerator, we can also instantiate more compressors and decompressors for groups of PEs to parallelize the ZFPe algorithm between multiple datapaths.

5.6. Comparison against State of the Art

We compare our evaluations against DeepSZ [15], the best existing accelerator with lossy floating-point compression to neural network inference. To the best of our knowledge, no other existing neural network inference accelerator exists which uses FPGA implementations of lossy floating-point compression algorithms for model or activation compression. DeepSZ does not present accuracy evaluations on MobileNet, so we compare the two systems assuming similar accuracy compared to the presented models. The DeepSZ paper also does not present an accelerator design or throughput evaluations, so we assume the performance efficiency of GhostSZ [16], the best-published FPGA implementation of SZ.

In terms of compression efficiency, DeepSZ demonstrates superior ratios compared to ZFPe. DeepSZ regularly achieves 50× compression without significant accuracy loss, while ZFPe only achieves lossless accuracy with less than 4× compression.

However, ZFPe is a more promising approach to removing the memory bottleneck from embedded accelerators, due to the significantly fewer resource requirements. For example, the low-power Lattice ECP5 chip has only 85,000 slices, which is already relatively larger compared to even lower power FPGAs, such as the Lattice iCE 40 or the Microsemi IGLOO2. Installing three GhostSZ modules, each consuming almost 20,000 slices, would leave a very small amount of resources for the actual inference accelerator. Furthermore, even though the compression efficiency of ZFPe is lower than DeepSZ, it is still sufficient to remove the bandwidth bottleneck of our edge accelerator hardware platform. While installing a faster memory on the platform may change the comparison, it is not likely to cause a significant difference, as the memory bandwidth on embedded devices is also often limited by power budgets.

5.7. Power-Performance Evaluation

We measure the projected power consumption of our accelerator using the Lattice Diamond power estimator [72], which gives an estimated 330 mW of power consumption. With 8 PEs running at 100 MHz, our accelerator achieves a peak performance of 800 MFLOPS, resulting in a power performance of 2.4 GFLOPS/Watt. This power efficiency is comparable to prior work on optimizing neural network accelerators for floating-point operations on FPGAs [73], which gives us confidence that we have evaluated our approaches on reasonable hardware and accelerator implementations.

6. Conclusions

In this work, we present an alternative approach to post-training quantization of neural networks, using a variant of the ZFP error-bounded lossy floating-point algorithm optimized for embedded accelerators. We demonstrate that our compression algorithm, ZFPe, is able to achieve superior performance per chip resource usage compared to existing work without causing significant accuracy degradation of inference, making it possible to deploy accurate inference acceleration on resource-constrained embedded FPGAs. Not only can our ZFPe-augmented FPGA accelerator achieve competitive power performance compared to existing work, but it can also achieve almost no accuracy loss on the MobileNet V2 image recognition network, while removing the bandwidth bottleneck of off-chip memory. This is a significant achievement considering that MobileNets famously suffer unacceptably sharp accuracy degradation with post-training quantization, requiring retraining or other methods to reclaim accuracy.

While our approach needs to bear the relatively high resource overhead of floating-point operations compared to fixed-point ones, we think many systems may still benefit from the memory performance benefits and high accuracy.

We plan to explore the efficacy of this approach to other neural network classes that have reported accuracy degradation with post-training quantization, including recurrent neural networks such as LSTMs. We are also exploring methods of computing MAC operations on compressed data directly, which can potentially remove the floating-point operation requirements.

Author Contributions: Conceptualization, S.-W.J.; methodology, S.-W.J.; software, S.-M.L.; validation, S.-M.L.; formal analysis, S.-M.L.; investigation, S.-M.L.; resources, S.-W.J.; data curation, S.-M.L.; writing—original draft prepration, S.-M.L.; writing—review and editing, S.-W.J. and S.-M.L.; visualization, S.-W.J. and S.-M.L.; project administration, S.-W.J.; funding acquisition, S.-W.J. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Mittal, S. A Survey on optimized implementation of deep learning models on the NVIDIA Jetson platform. *J. Syst. Archit.* **2019**, *97*, 428–442. [CrossRef]
2. Guo, K.; Zeng, S.; Yu, J.; Wang, Y.; Yang, H. A survey of FPGA-based neural network accelerator. *arXiv* **2017**, arXiv:1712.08934.
3. Qiu, J.; Wang, J.; Yao, S.; Guo, K.; Li, B.; Zhou, E.; Yu, J.; Tang, T.; Xu, N.; Song, S.; et al. Going deeper with embedded fpga platform for convolutional neural network. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 21–23 February 2016; pp. 26–35.
4. Howard, A.G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv* **2017**, arXiv:1704.04861.
5. Iandola, F.N.; Han, S.; Moskewicz, M.W.; Ashraf, K.; Dally, W.J.; Keutzer, K. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size. *arXiv* **2016**, arXiv:1602.07360.
6. Sanchez-Iborra, R.; Skarmeta, A.F. Tinyml-enabled frugal smart objects: Challenges and opportunities. *IEEE Circuits Syst. Mag.* **2020**, *20*, 4–18. [CrossRef]
7. Han, S.; Mao, H.; Dally, W.J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv* **2015**, arXiv:1510.00149.
8. Park, E.; Ahn, J.; Yoo, S. Weighted-Entropy-Based Quantization for Deep Neural Networks. In Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 21–26 July 2017; pp. 7197–7205.
9. Dokic, K.; Martinovic, M.; Mandusic, D. Inference speed and quantisation of neural networks with TensorFlow Lite for Microcontrollers framework. In Proceedings of the 2020 5th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM), Corfu, Greece, 25–27 September 2020; pp. 1–6.
10. Migacz, S. NVIDIA 8-bit inference with TensorRT. In Proceedings of the GPU Technology Conference, San Jose, CA, USA, 8–11 May 2017.
11. Kim, J.H.; Lee, J.; Anderson, J.H. FPGA architecture enhancements for efficient BNN implementation. In Proceedings of the 2018 International Conference on Field-Programmable Technology (FPT), Naha, Japan, 10–14 December 2018; pp. 214–221.
12. Liang, S.; Yin, S.; Liu, L.; Luk, W.; Wei, S. FP-BNN: Binarized neural network on FPGA. *Neurocomputing* **2018**, *275*, 1072–1086. [CrossRef]
13. Sandler, M.; Howard, A.; Zhu, M.; Zhmoginov, A.; Chen, L.C. Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–22 June 2018; pp. 4510–4520.
14. Krishnamoorthi, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv* **2018**, arXiv:1806.08342.
15. Jin, S.; Di, S.; Liang, X.; Tian, J.; Tao, D.; Cappello, F. DeepSZ: A novel framework to compress deep neural networks by using error-bounded lossy compression. In Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, Phoenix, AZ, USA, 24–28 June 2019; pp. 159–170.
16. Xiong, Q.; Patel, R.; Yang, C.; Geng, T.; Skjellum, A.; Herbordt, M.C. Ghostsz: A transparent fpga-accelerated lossy compression framework. In Proceedings of the 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), San Diego, CA, USA, 28 April–1 May 2019; pp. 258–266.
17. Sun, G.; Jun, S.W. ZFP-V: Hardware-optimized lossy floating point compression. In Proceedings of the 2019 International Conference on Field-Programmable Technology (ICFPT), Tianjin, China, 9–13 December 2019; pp. 117–125.
18. Lindstrom, P. Fixed-rate compressed floating-point arrays. *IEEE Trans. Vis. Comput. Graph.* **2014**, *20*, 2674–2683. [CrossRef]
19. Sze, V.; Chen, Y.H.; Yang, T.J.; Emer, J.S. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* **2017**, *105*, 2295–2329. [CrossRef]
20. Deng, J.; Dong, W.; Socher, R.; Li, L.J.; Li, K.; Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 20–25 June 2009; pp. 248–255.
21. Yun, S.; Wong, A. Do All MobileNets Quantize Poorly? Gaining Insights into the Effect of Quantization on Depthwise Separable Convolutional Networks through the Eyes of Multi-scale Distributional Dynamics. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, Nashville, TN, USA, 19–25 June 2021; pp. 2447–2456.
22. Laskaridis, S.; Venieris, S.I.; Almeida, M.; Leontiadis, I.; Lane, N.D. SPINN: Synergistic progressive inference of neural networks over device and cloud. In Proceedings of the 26th Annual International Conference on Mobile Computing and Networking, London, UK, 21–25 September 2020; pp. 1–15.
23. Cooke, R.A.; Fahmy, S.A. Quantifying the latency benefits of near-edge and in-network FPGA acceleration. In Proceedings of the Third ACM International Workshop on Edge Systems, Analytics and Networking, Heraklion, Greece, 27 April 2020; pp. 7–12.
24. Chen, J.; Hong, S.; He, W.; Moon, J.; Jun, S.W. Eciton: Very Low-Power LSTM Neural Network Accelerator for Predictive Maintenance at the Edge. In Proceedings of the 2021 31st International Conference on Field Programmable Logic and Applications (FPL), Dresden, Germany, 30 August–3 September 2021.
25. Chen, Y.H.; Krishna, T.; Emer, J.S.; Sze, V. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE J. Solid-State Circuits* **2016**, *52*, 127–138. [CrossRef]
26. Moons, B.; Uytterhoeven, R.; Dehaene, W.; Verhelst, M. 14.5 Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable Convolutional Neural Network processor in 28nm FDSOI. In Proceedings of the 2017 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, USA, 5–9 February 2017; pp. 246–247.

27. Shin, D.; Lee, J.; Lee, J.; Yoo, H.J. 14.2 DNPU: An 8.1 TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks. In Proceedings of the 2017 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, USA, 5–9 February 2017; pp. 240–241.
28. Whatmough, P.N.; Lee, S.K.; Brooks, D.; Wei, G.Y. DNN engine: A 28-nm timing-error tolerant sparse deep neural network processor for IoT applications. *IEEE J. Solid-State Circuits* **2018**, *53*, 2722–2731. [CrossRef]
29. Dundar, A.; Jin, J.; Martini, B.; Culurciello, E. Embedded Streaming Deep Neural Networks Accelerator With Applications. *IEEE Trans. Neural Networks Learn. Syst.* **2017**, *28*, 1572–1583. [CrossRef] [PubMed]
30. Jiao, L.; Luo, C.; Cao, W.; Zhou, X.; Wang, L. Accelerating low bit-width convolutional neural networks with embedded FPGA. In Proceedings of the 2017 27th International Conference on Field Programmable Logic and Applications (FPL), Ghent, Belgium, 4–8 September 2017; pp. 1–4.
31. Feng, G.; Hu, Z.; Chen, S.; Wu, F. Energy-efficient and high-throughput FPGA-based accelerator for Convolutional Neural Networks. In Proceedings of the 2016 13th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT), Nanjing, China, 25–28 October 2016; pp. 624–626.
32. Roukhami, M.; Lazarescu, M.T.; Gregoretti, F.; Lahbib, Y.; Mami, A. Very Low Power Neural Network FPGA Accelerators for Tag-Less Remote Person Identification Using Capacitive Sensors. *IEEE Access* **2019**, *7*, 102217–102231. [CrossRef]
33. Lemieux, G.G.; Edwards, J.; Vandergriendt, J.; Severance, A.; De Iaco, R.; Raouf, A.; Osman, H.; Watzka, T.; Singh, S. TinBiNN: Tiny binarized neural network overlay in about 5000 4-LUTs and 5 mw. *arXiv* **2019**, arXiv:1903.06630.
34. Rongshi, D.; Yongming, T. Accelerator Implementation of Lenet-5 Convolution Neural Network Based on FPGA with HLS. In Proceedings of the 2019 3rd International Conference on Circuits, System and Simulation (ICCSS), Nanjing, China, 13–15 June 2019; pp. 64–67.
35. Fuhl, W.; Santini, T.; Kasneci, G.; Rosenstiel, W.; Kasneci, E. Pupilnet v2. 0: Convolutional neural networks for cpu based real time robust pupil detection. *arXiv* **2017**, arXiv:1711.00112.
36. Li, Z.; Eichel, J.; Mishra, A.; Achkar, A.; Naik, K. A CPU-based algorithm for traffic optimization based on sparse convolutional neural networks. In Proceedings of the 2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE), Windsor, ON, Canada, 30 April–3 May 2017; pp. 1–5.
37. Semiconductor, L. Lattice ECP5. Available online: https://www.latticesemi.com/Products/FPGAandCPLD/ECP5 (accessed on 1 April 2021).
38. Han, S.; Liu, X.; Mao, H.; Pu, J.; Pedram, A.; Horowitz, M.A.; Dally, W.J. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Comput. Archit. News* **2016**, *44*, 243–254. [CrossRef]
39. Lin, D.; Talathi, S.; Annapureddy, S. Fixed point quantization of deep convolutional networks. In Proceedings of the 33rd International Conference on Machine Learning, New York City, NY, USA, 19–24 June 2016; pp. 2849–2858.
40. Shin, S.; Hwang, K.; Sung, W. Fixed-point performance analysis of recurrent neural networks. In Proceedings of the 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Shanghai, China, 20–25 March 2016; pp. 976–980.
41. Jacob, B.; Kligys, S.; Chen, B.; Zhu, M.; Tang, M.; Howard, A.; Adam, H.; Kalenichenko, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–22 June 2018; pp. 2704–2713.
42. Luo, J.H.; Wu, J.; Lin, W. Thinet: A filter level pruning method for deep neural network compression. In Proceedings of the IEEE international conference on computer vision, Venice, Italy, 22–29 October 2017; pp. 5058–5066.
43. Pietron, M.; Wielgosz, M. Retrain or Not Retrain?-Efficient Pruning Methods of Deep CNN Networks. In Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, 3–5 June 2020; pp. 452–463.
44. Fang, J.; Shafiee, A.; Abdel-Aziz, H.; Thorsley, D.; Georgiadis, G.; Hassoun, J.H. Post-training piecewise linear quantization for deep neural networks. In *European Conference on Computer Vision*; Springer: Cham, Switzerland, 2020; pp. 69–86.
45. Lee, D.; Kim, B. Retraining-based iterative weight quantization for deep neural networks. *arXiv* **2018**, arXiv:1805.11233.
46. Jin, Q.; Yang, L.; Liao, Z. Adabits: Neural network quantization with adaptive bit-widths. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, Virtual, 14–19 June 2020; pp. 2146–2156. Available online: https://openaccess.thecvf.com/content_CVPR_2020/papers/Jin_AdaBits_Neural_Network_Quantization_With_Adaptive_Bit-Widths_CVPR_2020_paper.pdf (accessed on 1 February 2022).
47. Gong, J.; Shen, H.; Zhang, G.; Liu, X.; Li, S.; Jin, G.; Maheshwari, N.; Fomenko, E.; Segal, E. Highly efficient 8-bit low precision inference of convolutional neural networks with intelcaffe. In Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-Designing Pareto-Efficient Deep Learning, Williamsburg, VA, USA, 24 April 2018; p. 1.
48. Mellempudi, N.; Kundu, A.; Mudigere, D.; Das, D.; Kaul, B.; Dubey, P. Ternary neural networks with fine-grained quantization. *arXiv* **2017**, arXiv:1705.01462.
49. Zhu, C.; Han, S.; Mao, H.; Dally, W. Trained ternary quantization. *arXiv* **2016**, arXiv:1612.01064.
50. Courbariaux, M.; Hubara, I.; Soudry, D.; El-Yaniv, R.; Bengio, Y. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or −1. *arXiv* **2016**, arXiv:1602.02830.
51. Pouransari, H.; Tu, Z.; Tuzel, O. Least squares binary quantization of neural networks. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops, Seattle, WA, USA, 14–19 June 2020; pp. 698–699.
52. Fujii, T.; Sato, S.; Nakahara, H. A threshold neuron pruning for a binarized deep neural network on an FPGA. *IEICE Trans. Inf. Syst.* **2018**, *101*, 376–386. [CrossRef]

53. Nagel, M.; Baalen, M.v.; Blankevoort, T.; Welling, M. Data-free quantization through weight equalization and bias correction. In Proceedings of the IEEE/CVF International Conference on Computer Vision, Seoul, Korea, 27 October–2 November 2019; pp. 1325–1334.
54. Jain, S.R.; Gural, A.; Wu, M.; Dick, C.H. Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks. *arXiv* **2019**, arXiv:1903.08066.
55. Di, S.; Cappello, F. Fast error-bounded lossy hpc data compression with sz. In Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Chicago, IL, USA, 23–27 May 2016; pp. 730–739.
56. Tao, D.; Di, S.; Chen, Z.; Cappello, F. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Orlando, FL, USA, 29 May–2 June 2017; pp. 1129–1139.
57. Liang, X.; Di, S.; Tao, D.; Li, S.; Li, S.; Guo, H.; Chen, Z.; Cappello, F. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In Proceedings of the 2018 IEEE International Conference on Big Data (Big Data), Seattle, WA, USA, 10–13 December 2018; pp. 438–447.
58. Lindstrom, P.; Isenburg, M. Fast and efficient compression of floating-point data. *IEEE Trans. Vis. Comput. Graph.* **2006**, *12*, 1245–1250. [CrossRef]
59. Diffenderfer, J.; Fox, A.L.; Hittinger, J.A.; Sanders, G.; Lindstrom, P.G. Error analysis of zfp compression for floating-point data. *SIAM J. Sci. Comput.* **2019**, *41*, A1867–A1898. [CrossRef]
60. SDRBench. Scientific Data Reduction Benchmarks. Available online: http://sdrbench.github.io (accessed on 1 July 2021).
61. Sun, G.; Kang, S.; Jun, S.W. BurstZ: A bandwidth-efficient scientific computing accelerator platform for large-scale data. In Proceedings of the 34th ACM International Conference on Supercomputing, Barcelona, Spain, 29 June–2 July 2020; pp. 1–12.
62. Choi, H.; Lee, J.; Sung, W. Memory access pattern-aware DRAM performance model for multi-core systems. In Proceedings of the (IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software, Austin, TX, USA, 10–12 April 2011; pp. 66–75.
63. Cuppu, V.; Jacob, B.; Davis, B.; Mudge, T. A performance comparison of contemporary DRAM architectures. In Proceedings of the 26th annual international symposium on Computer architecture, Atlanta, GA, USA, 1–4 May 1999; pp. 222–233.
64. Radiona. ULX3S. Available online: https://radiona.org/ulx3s/ (accessed on 1 April 2021).
65. Shah, D.; Hung, E.; Wolf, C.; Bazanski, S.; Gisselquist, D.; Milanovic, M. Yosys+ nextpnr: An open source framework from verilog to bitstream for commercial fpgas. In Proceedings of the 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), San Diego, CA, USA, 28 April–1 May 2019; pp. 1–4.
66. Ehliar, A. Area efficient floating-point adder and multiplier with IEEE-754 compatible semantics. In Proceedings of the 2014 International Conference on Field-Programmable Technology (FPT), Shanghai, China, 10–12 December 2014; pp. 131–138.
67. Roesler, E.; Nelson, B. Novel optimizations for hardware floating-point units in a modern FPGA architecture. In *International Conference on Field Programmable Logic and Applications*; Springer: Berlin/Heidelberg, Germany, 2002; pp. 637–646.
68. Zhao, K.; Di, S.; Lian, X.; Li, S.; Tao, D.; Bessac, J.; Chen, Z.; Cappello, F. SDRBench: Scientific Data Reduction Benchmark for Lossy Compressors. In Proceedings of the 2020 IEEE International Conference on Big Data (Big Data), Atlanta, GA, USA, 10–13 December 2020; pp. 2716–2724.
69. Chollet, F. Keras. 2015. Available online: https://keras.io (accessed on 1 April 2021).
70. Simonyan, K.; Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. In Proceedings of the International Conference on Learning Representations (ICLR), San Diego, CA, USA, 7–9 May 2015.
71. Szegedy, C.; Vanhoucke, V.; Ioffe, S.; Shlens, J.; Wojna, Z. Rethinking the inception architecture for computer vision. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 26 June–1 July 2016; pp. 2818–2826.
72. Semiconductor, L. Lattice Diamond Software. Available online: https://www.latticesemi.com/latticediamond (accessed on 1 April 2021).
73. Zhang, C.; Li, P.; Sun, G.; Guan, Y.; Xiao, B.; Cong, J. Optimizing fpga-based accelerator design for deep convolutional neural networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2015; pp. 161–170.