PQA: Exploring the Potential of Product Quantization in DNN Hardware Acceleration

AHMED F. ABOUELHAMAYED* and ANGELA CUI*, Cornell University, USA

JAVIER FERNANDEZ-MARQUES, Flower Labs, UK

NICHOLAS D. LANE, University of Cambridge, UK

MOHAMED S. ABDELFATTAH, Cornell University, USA

Conventional multiply-accumulate (MAC) operations have long dominated computation time for deep neural networks (DNNs), espcially convolutional neural networks (CNNs). Recently, product quantization (PQ) has been applied to these workloads, replacing MACs with memory lookups to pre-computed dot products. To better understand the efficiency tradeoffs of product-quantized DNNs (PQ-DNNs), we create a custom hardware accelerator to parallelize and accelerate nearest-neighbor search and dot-product lookups. Additionally, we perform an empirical study to investigate the efficiency–accuracy tradeoffs of different PQ parameterizations and training methods. We identify PQ configurations that improve performance-per-area for ResNet20 by up to 3.1×, even when compared to a highly optimized conventional DNN accelerator, with similar improvements on two additional compact DNNs. When comparing to recent PQ solutions, we outperform prior work by 4× in terms of performance-per-area with a 0.6% accuracy degradation. Finally, we reduce the bitwidth of PQ operations to investigate the impact on both hardware efficiency and accuracy. With only 2–6-bit precision on three compact DNNs, we were able to maintain DNN accuracy eliminating the need for DSPs.

CCS Concepts: • **Computing methodologies** → **Artificial intelligence**; • **Hardware**;

Additional Key Words and Phrases: deep neural network (DNN), product quantization, FPGA acceleration, low arithmetic precision

1 INTRODUCTION

Deep Neural Networks (DNNs) have become an essential computing technique and is finding its way to many computing form factors. Edge computing is especially challenging because of the constrained computing environment and limited power budget. To address this, DNN model optimization has taken many forms, including pruning [7], quantization [23, 41, 47], lightweight architectural designs [37, 42, 44], and faster algorithms [10, 33, 45]. Each of these techniques offers different trade-offs in terms of inference acceleration, model compression, hardware acceleration suitability, and accuracy degradation. Some methods focus solely on decreasing the number of computations while not affecting model size (e.g. Winograd [33]). Other methods such as linear quantization and pruning tend to generally decrease both computation and memory footprint. Alternatively, methods such as some forms of non-linear quantization [43] only decrease model size, and leave the number of computations unaffected. On this spectrum of compute and memory tradeoffs, a new compression methodology, product quantization (PQ), sits on one extreme. Specifically, PQ can eliminate *all* multiplications from the matrix multiplication operation [8], making it an interesting new compression method for approximating DNNs worth further investigation. Our work thoroughly investigates this emerging methodology and determines its practicality and hardware acceleration potential.

PQ emerges from the research area of information retrieval. Specifically, approximate nearest neighbours for information retrieval involves extracting a compact representation of high-dimensional features, for sample images [25,

---

50] in order to fetch similar ones from a data source or database. These input features can be encoded using PQ [29]. More recently, PQ has been repurposed for DNN inference acceleration—specifically, by accelerating matrix multiplication [8]— by encoding layer inputs into a set of *learnable prototypes* that replace inputs during inference according to the closest prototype. The set of prototypes is knowns as the prototype table. This allows a pre-computed dot product between inputs and parameters to be fetched from a lookup table instead of performing the conventional multiply-accumulate operations. However, existing works offer a limited evaluation of this new compression paradigm. For example, by exclusively applying PQ to the final layer of the DNN [8], through extremely simple networks [35], or by ignoring the efficiency implications associated with PQ. This is the case of PECAN [39], which uses short codes to lessen the accuracy degradation that PQ introduces at the cost of a 20× increase in memory footprint, resulting in an overall slowdown in DNN execution despite the elimination of multiply-accumulate operations.

Product quantization accelerates DNN inference by replacing convolutions (and in general any type of layer doing matrix-matrix multiplication) by a series of memory look-ups of pre-computed partial dot-products. Compute speedup is therefore achieved by reducing the computational footprint of compute intensive layers (for example, convolutions), sometimes in exchange of a higher memory footprint (depending on PQ parameters as we show in this work). As shown in previous works [8, 39], PQ has a big potential but it remains unclear how to effectively use this technique to accelerate DNNs without incurring severe accuracy degradation or a large memory footprint. Previous work (PECAN [39]) focused on demonstrating the potential of PQ but disregarded its efficiency trade-offs. For example, prior work did not evaluate whether PQ can actually result in compute speedups, and the resulting model sizes were many times larger than the original (non-PQ) model [39]. As we demonstrate in our work, accounting for FLOPs only is not a guarantee for speedup. This is because PQ replaces compute with memory accesses, and these are not captured when reporting FLOPs. Our work is the first to provide a more holistic efficiency analysis, which is fundamental for future work proposing alternative PQ implementations, for example, with new encoding functions, lightweight distance metrics, or new training methodologies.

Our work also proposes the first custom hardware architecture for DNN acceleration with product quantization—the Product Quantization Accelerator (PQA). We quickly realized that running PQ-DNNs on commodity CPUs or GPUs does not adequately reflect its hardware speedup potential, simply because these hardware options operate in very different ways to PQ-DNNs. Instead, we create a custom PQA on Intel Agilex FPGAs, taking advantage of a custom on-chip memory hierarchy and leveraging parallelism in PQ processing during nearest-neighbour computation, partial product look-up, and accumulation. Furthermore, we explore the use of low numerical bitwidth in these PQ operations. This fundamentally differs from traditional DNN quantization work that uses low bitwidths to approximate matrix multiplication where quantization here is used in components used for distance calculation and in stored results in LUT_{PQ} unlocking some new options explored in Section 2.5.

Motivated by the potential of PQ for inference acceleration, our work performs a holistic study of PQ in DNNs by exploring both its algorithmic efficiency, training dynamics, and hardware implementation. More concretely, our contributions are enumerated below:

(1) Present the first Product Quantization Accelerator (PQA) for lightweight CNNs, demonstrating that unlike GPUs and CPUs, custom hardware can indeed accelerate PQ by up to 3.1× compared to conventional DNNs.
(2) Evaluate opportunities for low numerical bitwidth, with hardware configurations that maintain accuracy with only 2-bit distance calculation operations on a Keyword Spotting task.
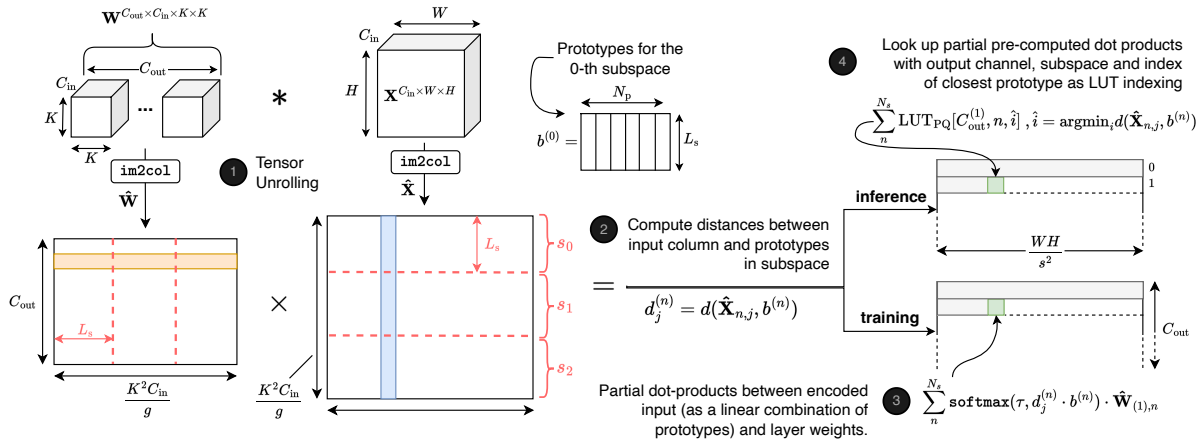
Fig. 1. Transforming a convolutional layer into its PQ equivalent. First, both input and weights tensors need to be unrolled, resulting in $\hat{X}$ and $\hat{W}$. The input matrix $\hat{X}$ is subdivided into N_s subspaces (three in this diagram), and the sub-columns in each one will be encoded using their respective bank of of prototypes $\mathbf{B}_l = [\mathbf{b}^{(0)}, \mathbf{b}^{(1)}, \mathbf{b}^{(2)}]$. Given a distance metric $d(\cdot)$, the input is encoded in a soft manner during training but replaced with a hard one-hot encoding during inference. For deployment, all the pre-computed dot products are stored in LUT_{PQ}.

(3) Evaluate PQ comprehensively for three CNNs (ResNet-20$_{CIFAR10}$, DW$_{EMNIST}$, MicroNet$_{KWS}$) to identify configurations that achieve 4–6× higher throughput compared to the most recent PQ literature, with minor accuracy loss (~0.6–0.9%).

(4) Provide a systematic study on the parameterization and training of PQ and its impact on compute and memory footprint and encoding degradation, including the proposal of a "corrector" DNN to improve PQ-DNN accuracy.

2 PRODUCT QUANTIZATION FOR DNNS

This section explains the process of training and inference for Product-Quantized DNNs to eliminate multiplications from the dot product operations prevalent in DNNs. As illustrated in Figure 1, we explain how convolutional or fully-connected layers can be approximated with PQ during training, and how to deploy a PQ-DNN after it's trained.

2.1 Product Quantization Fundamentals

Almost all layers found in modern ML architectures, including those in CNNs [20, 42, 44] or transformers [36, 46], can be expressed as a matrix-matrix multiplication. This is evident for standard fully connected layers where an input $\mathbf{X} \in \mathbb{R}^{A \times C_{\text{in}}}$ is transformed by weights $\mathbf{W} \in \mathbb{R}^{C_{\text{out}} \times A}$ to obtain output $\mathbf{Y} \in \mathbb{R}^{A \times C_{\text{out}}}$, where C_{in} and C_{out} are the number of input and output features or, more generically, channels. In the case of convolutions, both input $\mathbf{X} \in \mathbb{R}^{C_{\text{in}} \times W \times H}$ and weights $\mathbf{W} \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times K_{\text{w}} \times K_{\text{h}}}$ tensors first need to be unrolled. This unrolling can be done following the `im2col` algorithm [18, 27], resulting in unrolled matrices of size $(K_w K_h C_{\text{in}}/g) \times (WH/s^2)$ and $C_{\text{out}} \times (K_w K_h C_{\text{in}}/g)$ for input and weights tensors respectively with s and g representing stride and groups respectively. This is illustrated in Figure 1 (left).

We now introduce PQ-specific terms and assume a matrix of weights $\mathbf{W} \in \mathbb{R}^{C_{\text{out}} \times A}$ and of inputs $\mathbf{X} \in \mathbb{R}^{A \times C_{\text{in}}}$ exist, whether they are from a linear layer or are the result of a tensor unrolling. With PQ, we aim to aggressively quantize the input matrix on-the-fly so that the multiplication with $\mathbf{W}$ can be reduced to a series of lookups to a table of pre-computed

dot-products of size $\text{LUT}_{\text{PQ}} \in \mathbb{R}^{C_{\text{out}} \times N_{\text{s}} \times N_{\text{P}}}$. The input $\mathbf{X}$ can be split into N_{s} disjoint groups along the rows dimension. We call each of these groups *subspaces* as shown in Figure 1. Each subspace gets assigned a bank of *prototypes* $\mathbf{b}^{(n)}$ of size $L_{\text{s}} \times N_{\text{p}}$, where N_{p} and L_{s} represent the number and the length of the prototypes respectively. This list of prototypes banks $\mathbf{B}_l = [\mathbf{b}^{(0)}, \mathbf{b}^{(1)}, ..., \mathbf{b}^{(N_{\text{s}}-1)}]$ are learnable parameters of PQ layers, with subscript l denoting the layer index. PQ involves getting the index of the closest prototype via distance $d(\cdot)$ to each sub-column of the input matrix (*i.e.* vectors of length L_{s}). In addition, all dot products between the weights and the prototypes are precomputed and stored in a look-up table LUT_{PQ}. Therefore, during inference, a dot product is looked up from LUT_{PQ} depending on the closest prototype matched by an input vector, eliminating multiplication operations altogether.

Training PQ layers involves learning two sets of parameters: the standard layer weights $\mathbf{W}_l$ and prototypes $\mathbf{B}_l$. Then, prior to deployment, the table of pre-computed dot products LUT_{PQ} is obtained by performing a Cartesian product between $\mathbf{W}_l$ and $\mathbf{B}_l$. Only LUT_{PQ} and $\mathbf{B}_l$ are deployed to perform perform inference, with no need for the actual model parameters $\mathbf{W}_l$. This is because LUT_{PQ} explicitly contains all model parameters and their dot products with the input prototypes $\mathbf{B}_l$. Figure 1 (right) illustrates training and inference for PQ-DNNs, explained further below.

2.2   Choosing a distance metric

For both training and inference, a distance $d(\cdot)$ needs to be defined to encode the input using the layer prototypes $\mathbf{B}_l$. A reasonable choice would be the Euclidean distance. For PQ, the relative ranking of distances between input column $\hat{\mathbf{X}}_{n,j}$ and the prototypes of that subspace $\mathbf{b}^{(n)}$ is far more important than the actual distance value. This therefore leaves room for more lightweight distance metrics such as the L_1 Manhattan distance as well. Recent work [8] has also proposed using locality-sensitive hashing to match inputs with prototypes thereby performing comparisons between inputs and prototypes more quickly but placing further constraints on the nature of the learned prototypes $\mathbf{B}_l$. In our case, we explore Euclidean distance (L2 norm). While this introduces multiplication to compute the squared distance, performance is still dominated by memory access. We also explore Manhattan distance (L1 norm) but that often led to higher accuracy degradation.

2.3   Input encoding

During training the encoding of the input is done explicitly, *i.e.*, the unrolled input $\hat{\mathbf{X}}$ is actually transformed into a new matrix of the same dimensions that is later multiplied with the unrolled layer weights $\hat{\mathbf{W}}$. Given the j-th column of $\hat{\mathbf{X}}$ along the n-th subspace, $\hat{\mathbf{X}}_{n,j}$, the distances $\mathbf{d}_j^{(n)} = d(\hat{\mathbf{X}}_{n,j}, \mathbf{b}^{(n)})$ to each of the prototypes are computed. The encoded portion of the input $\hat{\mathbf{X}}_{n,j}^{\text{enc}}$ is obtained by a weighed linear combination of $\mathbf{b}^{(n)}$ with normalized distances $\phi(\tau, \mathbf{d}_j^{(n)})$

$$\hat{\mathbf{X}}_{n,j}^{\text{enc}} = \phi\left(\tau, \mathbf{d}_j^{(n)}\right) \mathbf{b}^{(n)} = \sum_p^{N_{\text{p}}} \frac{\mathbf{b}_p^{(n)} \exp(d_j^{(n,p)}/\tau)}{\sum_k^{N_{\text{p}}} \exp(d_j^{(n,k)}/\tau)} \tag{1}$$

where $\mathbf{b}_p^{(n)}$ stands for the p-th prototype in the n-th subspace, $d_j^{(n,p)}$ is the distance to the p-th prototype, and τ is a temperature factor. As $\tau \to 0$, the temperatured-softmax $\phi(\tau, \cdot)$ outputs a sharper distribution over $\mathbf{d}_j^{(n)}$, transitioning in this way from *soft* linear combination of prototypes where all prototypes are considered with different weights into a *hard*, one-hot encoding where only the prototype closest to the input is considered. During inference PQ is implemented with one-hot assignments to match a single prototype to each input vector.

2.4 Constructing LUT$_{\text{PQ}}$

At inference, $\hat{X}_{n,j}^{\text{enc}}$ is not materialised since only the index of the closest prototype is needed to retrieve the partial dot product from LUT$_{\text{PQ}}$. After training, the unrolled weights $\hat{W}$ are partitioned into subspaces along the column dimension (as shown in Figure 1). Then, the dot product between a $1 \times L_s$ sub-row in $\hat{W}$ and a prototype in such subspace, corresponds to one entry in LUT$_{\text{PQ}}$. Repeating this for all N_s, C_{out} and N_p completes the table. This allows us to fetch a precomputed dot product between any prototype B_l and its corresponding layer weights.

2.5 Exploring Low Numerical Bitwidth for PQ-DNNs

Although PQ is fundamentally a vector quantization method, it offers scope for applying *additional* quantization to its constituent operations. Specifically, lower bitwidth can be used within the prototype table and the LUT_{PQ}, thereby offering significant efficiency and area advantages. We apply post-training asymmetric linear quantization for both of these operations. For the prototype table, we abstain from dequantizing the stored quantized values. Instead, the incoming input is quantized using identical parameters, and the distance calculation is performed within the quantized domain. As we see in the results, this approach yields satisfactory results. In contrast, for the LUT_{PQ}, we employ dequantization of stored quantized values to perform wide accumulation in 16-bit precision. Determining the scale and offset for quantization entails various possibilities, each bringing its unique cost and benefit trade-off. The most cost-effective option is to have a single scale and offset value for the entire model. Per-layer and per-channel quantization parameters are also prevalent in conventional quantization to improve accuracy. Furthermore, we introduce a novel *per-subspace* quantization parameters—uniquely-suited to PQ-based matrix multiplication, and as we show in the results, achieve the highest accuracy.

2.6 The Potential for Compute Speedups on CPU, GPU, and PQA

Commodity hardware such as CPUs and GPUs are inherently unsuitable for PQ-DNNs but are increasingly better suited to conventional DNNs, making a comparison on those platforms somewhat skewed. A key reason for creating PQA is to be able to fairly assess the efficiency of PQ-DNNs compared to conventional DNNs. PQA has the potential to attain high efficiency because of the high levels of parallelism and memory banking that is possible as described in Section 3. To quantify this, we measure the speedup of executing PQ layers of ResNet20 [39] on a CPU, GPU, and PQA. Our baseline custom hardware is DLA [1] for native convolutions. Figure 2 shows the speedup percentage for the unique layers of the network, obtained by running PQ and conventional convolution on different hardware and comparing the latencies. We sweep different values of L_s while N_p is kept at 16. It is clear that there is a consistent slowdown when running PQ on a CPU or a GPU as there is no speedup for different values of L_s and the same trend appears when trying different values of N_p. Measuring the effect of PQ on commodity hardware like CPUs or GPUs is therefore a poor way of assessing its acceleration potential. However, using a custom hardware that is specifically designed for PQ achieves improvements in latency for *certain* configurations over DLA [1, 3]. While this is not the case in all layers, the speedup reaches up to 150% in the last layer of the network at a large value of L_s.

3 PRODUCT QUANTIZATION ACCELERATOR (PQA) ARCHITECTURE

In this section we detail the design of a custom PQ inference Accelerator (PQA) on an FPGA. We also benchmark PQ-DNNs on CPUs and GPUs, and we argue with empirical results that these devices are not suitable for assessing the
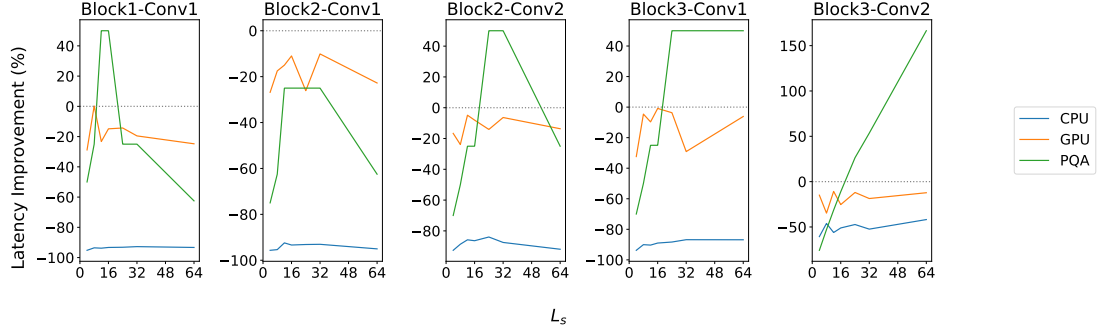
Fig. 2. Speedup of PQ vs conventional convolutions on different hardware when run on all unique layers of ResNet20. CPU is 13th Gen Intel(R) Core(TM) i9-13900K and GPU is NVIDIA GeForce RTX 4090.

efficiency potential of PQ. This motivates the design of PQA which we show to indeed result in faster DNN execution for some PQ parameters.

3.1 PQA Overview

Rather than relying on multiply-accumulate operations to calculate the result of a matrix-matrix multiplication, PQ performs memory lookups of precomputed results. This requires designing hardware that can perform these lookups efficiently. Luckily, PQ has many hardware-friendly properties that can make a custom hardware design efficient. (1) Subspaces are completely independent, allowing us to partition the large dot product table (LUT_{PQ}) memory across multiple small on-chip memories to allow parallel memory accesses. (2) The compute-heavy distance computation—to find the closest prototype—needs to be done only once regardless of the number of output channels. (3) With the exception of distance calculation, all other operations do not require heavy computations as they are either memory lookups or accumulations.

Figure 3 shows the architecture of PQA: our novel compute engine that can be used to perform PQ inference. The engine consists of 3 main modules: A **distance calculation** module is responsible for determining the indices of the prototypes that are closest to the corresponding inputs. Next, the **product lookup** module consists of a partitioned memory array that is responsible for looking up the product of the closest prototype and the current weight for each subspace. Finally, the **accumulator** is responsible for adding results from different subspaces and producing the final outputs. Note that we implement PQA on an Intel Agilex DE10 board with DDR4 external memory with 36 GB/s transfer speeds, but we extrapolate our results to high-bandwidth memory (HBM) that can reach up to 460 GB/s using our hardware-verified cycle-accurate simulator that we develop based on performance model explained in Section 3.4. We provide more analysis of HBM in Section 6.

3.2 PQA Hardware Components

PQA is organized into processing lanes, replicated N_s^{vec} times to process subspaces in parallel. It is also able to produce N_{out}^{vec} outputs at a time by further breaking down the large LUT_{PQ} into smaller memories as shown in Figure 3. This allows producing multiple outputs at the same time. As for inputs, the distance calculation module compares the input with N_{p}^{vec} prototypes at a time where it compares L_s^{vec} elements of the prototype each cycle. The Compute Engine receives L_s^{vec} portions of the input every cycle to be compared to all prototypes in each subspace, and to find the closest
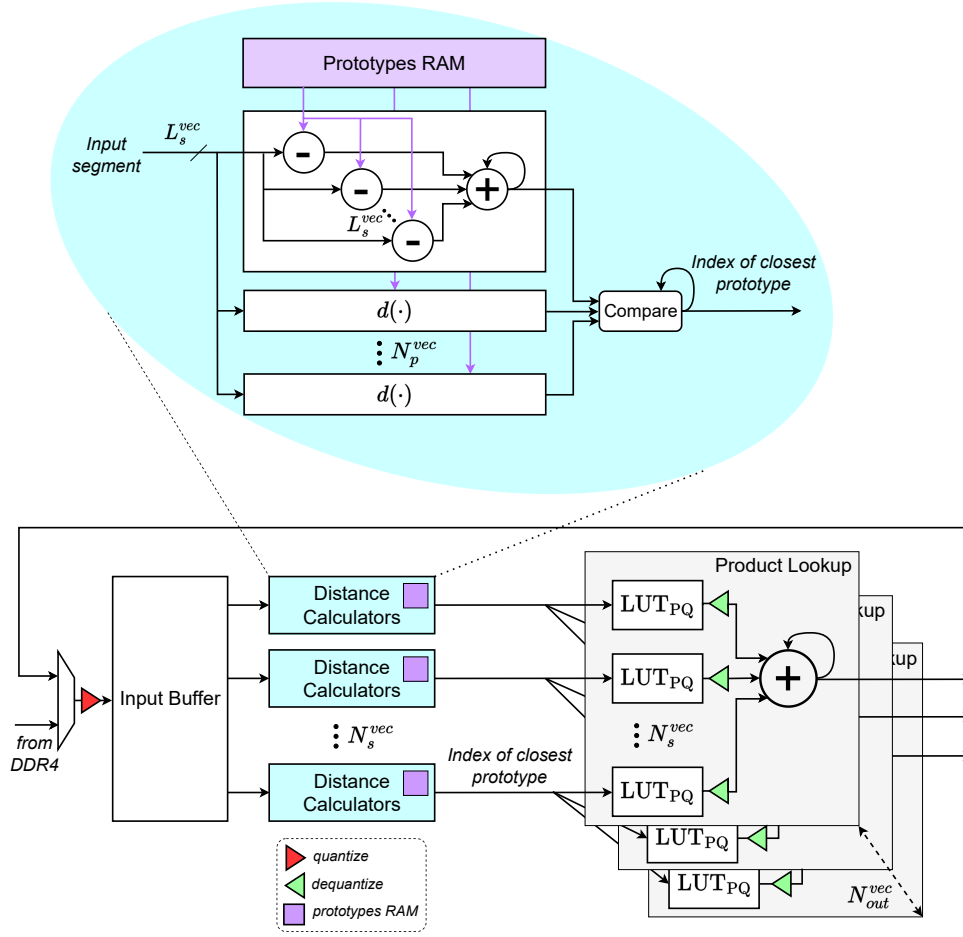
Fig. 3. Block diagram of PQA, the proposed custom hardware implementation for PQ.

one depending on the distance function $d(\cdot)$. We implement both the L1 and L2 distance functions in PQA. The index of the closest prototype is then sent to the product lookup portion that iterates over all weights, looks up the precomputed partial dot products, and accumulates these to compute each output. In addition to the *vectorization* parameters L_s^{vec}, N_p^{vec}, N_s^{vec}, N_{out}^{vec}, PQA contains *maximum* parameters that are primarily used for sizing the input and LUT_{PQ} buffers that hold layer values on-chip to enable single-cycle memory fetches during that layer's execution: L_s^{max}, N_p^{max}, N_s^{max}, N_{out}^{max}, and N_{in}^{max}.

3.2.1 Distance Calculators. This module consists of many processing lanes, one per N_s^{vec}. The lane contains N_p^{vec} difference calculators. Each difference calculator corresponds to a certain prototype. The lane receives an input which is L_s^{vec} elements wide, passes it to all difference calculators, each producing the difference between this input and its corresponding prototype. These differences go to a comparator which is responsible for outputting the index of the prototype with the minimum distance from the input vector. The comparator needs to cache the minimum distance so

far to compare it with upcoming comparisons in the next clock cycle with the next set of prototypes since the hardware is vectorized over N_p^{vec} to only perform some of the comparisons in each cycle.

Each difference calculator contains one group of prototypes N_p and is responsible for calculating the distance between the input and these specific prototypes. It does that by passing each of the input elements along with its corresponding element in the prototype into the difference module then the output of all difference modules is added to an accumulator to produce the total distance after processing all L_s^{vec} portions of the input.

3.2.2 Product Lookup and Accumulators. The product table memory is partitioned into N_s^{vec} different banks to allow parallel lookups for each subspace. Furthermore, each subspace's LUT is partitioned into N_{out}^{vec} different memories—each handles a different set of output channels—to further parallelize product lookup and to produce multiple outputs simultaneously. PQA is therefore able to perform $N_{out}^{vec} \times N_s^{vec}$ LUT$_{PQ}$ lookups in parallel. This custom on-chip memory hierarchy is key to PQA's acceleration, and is not achievable with commodity hardware. Finally, results are accumulated across subspaces using a set of parallel accumulators repeated N_{out}^{vec} times. The accumulator is needed (and not a simpler adder tree) because of the vectorization along N_s^{vec} which means that the difference calculated at each cycle might only be a partial dot product until all subspaces are processed.

3.3 Numerical Bitwidths for PQA Components

To further improve efficiency, lower numerical bitwidth can be used for PQ operations as explained in Section 2.5. We parameterize our hardware so that it can operate with any bitwidth in both the distance calculation and product lookup portions—each of those parts can use a different bitwidth since they are only connected with the index of the closest prototype, used to form the address for the LUT$_{PQ}$ memories. The output of LUT$_{PQ}$ is dequantized to 16 bits before accumulation, then subsequently, the output of the accumulators is quantized again before storing in the input buffer. This ensures that that all buffers benefit from the smaller bitwidths, while using a wider bitwidth for accumulation.

3.4 Performance Modeling

Because of the vectorization along the L_s^{vec}, N_p^{vec}, N_s^{vec}, and N_{out}^{vec} dimensions, a simple estimate of the total number of compute cycles per layer can be computed using Equation 2.

$$Cycles_{compute} = \max\left(\left\lceil \frac{N_p}{N_p^{vec}} \right\rceil \times \left\lceil \frac{L_s}{L_s^{vec}} \right\rceil, \left\lceil \frac{C_{out}}{N_{out}^{vec}} \right\rceil\right) \times \left\lceil \frac{N_s}{N_s^{vec}} \right\rceil \times \frac{WH}{s^2} \tag{2}$$

There are 2 compute stages: Distance calculation, taking $\left\lceil \frac{N_p}{N_p^{vec}} \right\rceil \times \left\lceil \frac{L_s}{L_s^{vec}} \right\rceil$ cycles, and product lookup, taking $\left\lceil \frac{C_{out}}{N_{out}^{vec}} \right\rceil$ cycles, the maximum of those 2 determines the total compute taken to process the subspace. Since N_s^{vec} subspaces are processed in parallel, we multiply that maximum by $\left\lceil \frac{N_s}{N_s^{vec}} \right\rceil$. Finally, this process is repeated for each of the $\frac{WH}{s^2}$ columns of the input. In parallel to the compute cycles, memory loading of prototypes and LUT_{PQ} occurs in parallel. The number of cycles needed for that can be computed using Equation 3.

$$Cycles_{load} = \max\left(\left\lceil \frac{C_{out} \times N_p \times N_s}{N_{out}^{vec} \times N_s^{vec}} \right\rceil, \left\lceil \frac{|\mathbf{B}_l| + |\text{LUT}_{PQ}|}{\text{Mem}_{BW}} \right\rceil\right) \tag{3}$$

where $|\mathbf{B}_l|$ and $|\text{LUT}_{PQ}|$ are the size in bits of all prototypes and LUT$_{PQ}$ for each layer. Mem$_{BW}$ is the external memory bandwidth in bits/s. Equation 3 takes a maximum between the memory loading cycles from the external memory bus, and the internal memory bandwidth of LUT$_{PQ}$. In most cases the internal memory bandwidth is much higher than

external, but for some configurations of PQA with HBM memory where the value of Mem_{BW} is much higher, the situation may be flipped, where the internal memory bandwidth becomes the bottleneck, thus necessitating the max operator in our analytical model.

4 EXPERIMENTAL SETUP

Here we provide a detailed description of the hardware environments, models, dataset, hyperparameters and training schemes used in our per-layer study and full-model results. We also explain our enhancements to training PQ-DNNs to minimize accuracy degradation.

4.1 HW-Setup

We designed PQA using Intel's OpenCL SDK 21.2, targeting the DE10-Agilex FPGA board with the AGFB014R24B2E2V FPGA. Throughout our results, we compare PQA to a canonical systolic array-based deep learning accelerator (DLA) for native convolutions [1, 3]. One of the important considerations when designing a custom hardware is the area it requires. Area and Fmax shown in the study are based on a real hardware accelerator running on the FPGA using the Intel OpenCL runtime. We express the area in terms of the number of equivalent Adaptive Logic Modules (eALMs) by following prior work that quantified the area of one DSP to be equivalent to 30 ALMs, and one BRAM is equivalent to 40 ALMs [40][1]. For number of cycles, we verified that the numbers resulting from our analytical model described in Section 3.4 with the measured hardware performance measured by running on the DE-10 Agilex board. We used the analytical performance model in some of our results, especially to simulate HBM memory instead of the DDR4 that is available on the DE-10 board. We compare PQA to a canonical systolic array-based deep learning accelerator (DLA) for native convolutions [3] running on an Arria 10 FPGA (20 nm technology node). For a fair comparison with our 8-nm Agilex PQA, we scale the reported DLA frequency by 1.6× to account for the newer process technology. We get this scaling factor by comparing the frequency of a finite impulse response (FIR) filter design (1.60× frequency scale factor), and a Fast Fourier Transform (FFT) design (1.47× frequency scale factor) when implemented on the Arria 10 versus the Agilex FPGA, and we opted to use the higher scaling factor to favor the baseline DLA.

4.2 Models & Training

4.2.1 Datasets. We make use of two image classification datasets: CIFAR-10 [32] and EMNIST [12]. The former is comprised of 50K 32×32 RGB images for training and 10K for testing, with both sets evenly split along ten image classes. The EMNIST dataset on the other hand is much larger totaling 112,800 and 18,800 images for training and testing respectively. We use the *balanced* partitioning of EMNIST which contains 28×28 greyscale images of digits and letters resulting in 47 classes with, as the name suggests, the same number of examples. The current best performing architecture on this dataset reaches 91.06% [26] in this partition. For both training sets we randomly leave 10% out for validation. For keyword spotting we rely on the SpeechCommands [48] data which is comprised of 105,829, 16-KHz 1-second long audio clips of a spoken word (e.g. "yes", "up", "stop") and the task is to classify these correctly into 12 possible classes. Similar to previous works [4, 6, 53] we pre-process each audio clip and extract 10 MFCC [13] features using a 40ms window with a 20ms stride resulting in 10×49 input matrices.

---

[1]While prior work uses a different family of FPGAs, it is the best available public estimate to the best of our knowledge and we believe it is sufficient for our evaluation.

Table 1. Hyperparameters used in our experiments. Due to the large space of PQ configurations and τ values considered, there is not a single *golden config* for each model. Here we report the ranges of hyperparemters that we found to deliver good quality models across PQ settings. Column *Early Stop τ* indicate the epoch from which temperature τ is kept fixed for the remaining of the training.

Model	Epochs	Batch	LR (params)	LR (proto)	Start τ	End τ	Early Stop τ	Scheduler
ResNet20	120	64	[0.005,...,0.05]	[0.025,...,0.075]	1.0	0.0005	10	StepLR [40,60,90]
MicroNet-PQ	30	96	[0.001,...,0.01]	[0.001,..., 0.025]	1.0	[0.0005, 0.001]	10	ExponentialLR [0.25,...,2.0]
DW$_{\text{EMNIST}}$	90	96	[0.000,...,0.001]	[0.005,...,0.05]	1.0	0.0005	10	StepLR [30,50,70]

4.2.2   Training Infrastructure. We implement our training infrastructure with PyTorch [38]. We analyze 3 models: ResNet-20[21] on CIFAR10, MicroNet[4] on KWS and a custom CNN with 10 depth-wise separable layers called DW on EMNIST. Full details of the model architectures is in Appendix A. Training PQ models is difficult and is currently only proven to work on smaller models like the ones we consider. This is similar to other work in the literature[39]. We use two optimizers: one for the bank of prototypes in each layer $\mathbf{B}_l$; and another for the rest of the parameters in the model (e.g. the layer weights, and non-PQ layers.). Both instantiate an Adam optimizer [30] albeit with different learning rate. There are also two learning rate schedulers, each with its own decaying coefficient and scheduling. The hyperparameters for each model are presented in Table 1. We found learning rates and scheduling parameters to have large impact, not only in final model quality, but also in terms of training stability.

4.2.3   PQ Training Enhancements. A number of training techniques and tricks were considered and introduced to our training pipeline. We found gradient clipping to be crucial when training deeper models (*i.e.* ResNet20 and DW$_{\text{EMNIST}}$). Value-based gradient clipping [51] with small thresholds (*e.g.* 0.25, 0.5) alleviated exploding gradients in some cases. During the first epochs of training, the encoded input is obtained with τ values that are gradually decreased. As it becomes smaller, the construction of the *prototyped* input gets close to one-hot, impacting the flow of gradients. We implemented two mechanisms to counteract this while still ensuring the performance of one-hot encoding is not affected: formulating the prototype selection via a Gumbel-Softmax [24]. This, however didn't seem to have a clear impact on the quality of our training. What did have a small positive impact was to introduce a stochastic masking to the final encoded input $\hat{X}_{n,j}^{\text{enc}}$ prior to multiplying it with the layer weights. This masking, applied at the subspace level and individually to input column, allowed a fraction ρ of vectors to remain unencoded. We found that small $\rho = 0.1$ lead to higher final one-hot PQ accuracy. Higher values would prevent layer weights and prototypes to adequately operate in one-hot scenarios, as it is the case during inference. Furthermore, we add a new term to our Cross Entropy training loss that encourages prototypes within a subspace to be orthogonal to each other [9]. We only found this regularization term to be useful to lessen the impact of sub-optimal training hyperparameters.

5   PRODUCT-QUANTIZABILITY: STUDY & TRADE-OFFS

This section investigates PQ parameterizations and training enhancements, and the corresponding tradeoffs in PQ-DNN accuracy, compute, and memory footprint.

5.1   PQ Implementation Trade-Offs

Different parameterizations of PQ (*i.e.* choice of $\{N_p, L_s\}$), will lead to dramatically different levels of model acceleration, memory footprint, and accuracy degradation. Understanding these trade-offs is fundamental to the design of PQ-DNNs.

Table 2. The best performing models in PECAN-D require a high number of prototypes (N_p) of short length (L_s). As a result, look-up tables are many times larger than the original non-PQ model.

Model	Params	N_p	L_s	$|\text{LUT}_{PQ}|$	Mem. Footprint
LetNet	61K	64	9 & 8	489K	8.0×
ResNet20	269K	128 & 64	3 & 4	5.7M	21.3×

5.1.1 Compute footprint. The number of FLOPs to perform an `im2col`-equivalent convolution (as in Figure 1) is given by $\text{FLOPs}_{\text{im2col}} = 2K^2 C_{\text{in}} WHC_{\text{out}}$, assuming groups and stride equal to one and squared kernels. The same layer but implemented with PQ would result in $\text{FLOPs}_{PQ} = \text{FLOPs}^{\text{enc}} + \text{FLOPs}^{\text{add}}$. The first term is given by $\text{FLOPs}^{\text{enc}} = N_s d(\cdot)_{\text{FLOPs}} WH$, with $d(\cdot)_{\text{FLOPs}}$ representing the FLOPs to compute the distances $\mathbf{d}_j^{(n)}$. Assuming Euclidean distance, $d(\cdot)_{\text{FLOPs}} = 3N_p L_s$. Term $\text{FLOPs}^{\text{add}} = (N_s - 1)WHC_{\text{out}}$ accounts for the cost of performing the addition of partial dot-products needed to complete a full row-column dot-product $\hat{\mathbf{X}} \cdot \hat{\mathbf{W}}$. The FLOPs savings ratio is given by

$$\frac{\text{FLOPs}_{\text{im2col}}}{\text{FLOPs}_{PQ}} = \frac{2K^2 C_{\text{in}} WHC_{\text{out}}}{N_s WH(d(\cdot)_{\text{FLOPs}} + C_{\text{out}})} = \frac{2C_{\text{out}} L_s}{3N_p L_s + C_{\text{out}}} \tag{4}$$

with $N_s = K^2 C_{\text{in}}/L_s$ and simplifying $(N_s - 1)$ as just N_s. This expression suggests that reducing the number of prototypes has a larger impact than increasing their length. This can also be seen in Figure 5 where the grey-shaded squares indicating the FLOPs increase heavily as you increase N_p but they do not increase as much when you use larger L_s.

5.1.2 Memory footprint. The number of parameters in a convolutional layer is given by $C_{\text{out}} C_{\text{in}} K^2$, assuming squared kernels and groups=1. When transformed into PQ, this number becomes $\text{params}_{PQ} = |\mathbf{B}_l| + |\text{LUT}_{PQ}| = N_s N_p (L_s + C_{\text{out}})$. In this way, and assuming that $C_{\text{out}} \gg L_s$, savings in parameter count is possible when $C_{\text{in}} K^2/N_s N_p = L_s/N_p > 1$, i.e., as it was the case when assessing the compute footprint of PQ, longer prototypes and few of them are also preferred.

5.1.3 Accuracy degradation. Previous attempts of applying PQ to the entire network required using a larger number of short prototypes to maximise model accuracy. In PECAN-D [39], this translated into a very large increase in memory footprint, see Table 2. While intuitively lower L_s should always be preferred when prioritising accuracy degradation, it is unclear what the underlying trade-offs between N_p and L_s are at different layers of a network.

5.2 PQ Layer-wise Parameter Sweeps

In this section we assess the impact of different $\{N_p, L_s\}$ in terms of memory and compute footprint as well as accuracy degradation. We design a per-layer analysis where the task for each PQ layer is to generate an output $\mathbf{Y}_{PQ}$ that is as close as possible to $\mathbf{Y}$, the output of an equivalent layer from a standard, non-PQ, pretrained model. Framing this empirical study in such way enables us to isolate individual layers from the impact of other elements in the model, training hyperparameters, and dynamics. This study uses DW_{EMNIST} a CNN containing 10 depth-wise separable convolutional layers designed for image classification on the EMNIST [12] dataset as discussed in Section 4.2.

Given a PQ layer with randomly initialised $\mathbf{B}_l$ and weights from its non-PQ counterpart, a two-phases study is conducted. First, the prototypes are trained to minimise $\text{MSE}^{\text{enc}}(\hat{\mathbf{X}}^{\text{enc}}, \mathbf{X})$; then, the divergence of $\mathbf{Y}_{PQ}$ w.r.t. $\mathbf{Y}$ is measured in four different scenarios, namely when every other element in the layer is kept frozen (blue dots) in top plot of Figure 4, when prototypes are further finetuned to minimise $\text{MSE}^{\text{out}} = \text{MSE}(\mathbf{Y}_{PQ}, \mathbf{Y})$ (green dots), when only
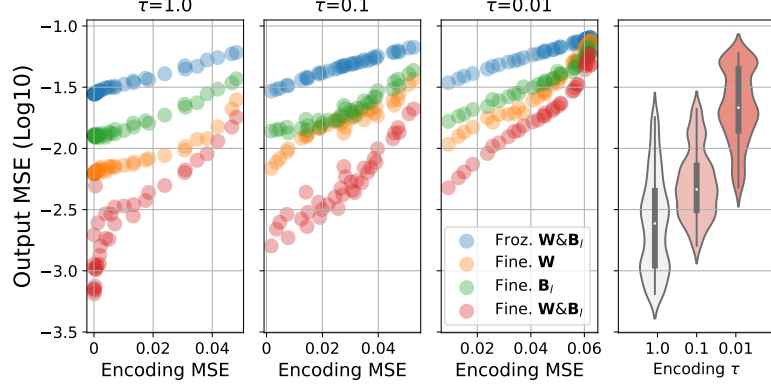
Fig. 4. As the temperature τ decreases, the error in the output of a PQ layer increases rapidly. Even when prototypes and layer parameters are finetuned to minimise $\text{MSE}(\mathbf{Y}_{\text{PQ}}, \mathbf{Y})$, most of the $\{N_{\text{p}}, L_{\text{s}}\}$ configurations (each configuration is a dot) leads to much larger error (see rightmost subplot).
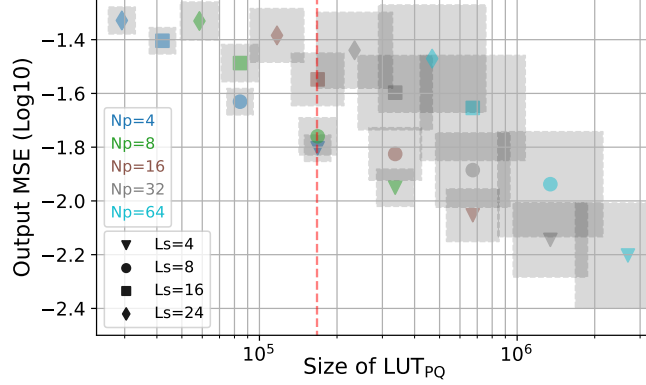

Fig. 5. Analysis of a layer with a $366 \times 4 \times 4$ input. When spatial dimensions of the input are small relative to their channel dimension, fewer prototypes lead to similar error to other configurations that result in larger LUT_{PQ} and a higher $\text{FLOPs}^{\text{enc}}$ (area of grey-shaded squares) as in Eq. 4. The red line indicates the size in the equivalent non-PQ layer.

the layer weights are finetuned (orange dots), and when both prototypes and layer weights are jointly finetuned (red dots). All settings involving finetuning start from the same state of trained prototypes (blue dots). These two phases are repeated for a broad $\{N_{\text{p}}, L_{\text{s}}, \tau\}$ range and across all layers. We make the following observations.

1. Learning one-hot encodings is hard. As temperature parameter τ decreases, there is a shift in the distribution over $\{N_{\text{p}}, L_{\text{s}}\}$ settings that lead similar MSE^{enc} and MSE^{out} trends (blue, green orange dots in Figure 4). More evidently (red dots and violin plot) is the impact on MSE^{out} when both layer weights and prototypes are finetuned. These results suggest that at least certain $\{N_{\text{p}}, L_{\text{s}}\}$ can perform well at lower τ values but most cannot.

2. Reasons for larger N_{p} decrease with depth. Accuracy decreases faster by lowering L_{s} than increasing N_{p} as can be observed in Figure 5. However, this dimension brings a large increase in memory footprint since the size of LUT_{PQ} is inversely proportional to L_{s} (*i.e.* shorter prototypes lead to more subspaces, which in turn leads to more pre-computed dot products). In comparison, larger N_{p} has a lesser impact on MSE^{out} overall but affects both LUT_{PQ}
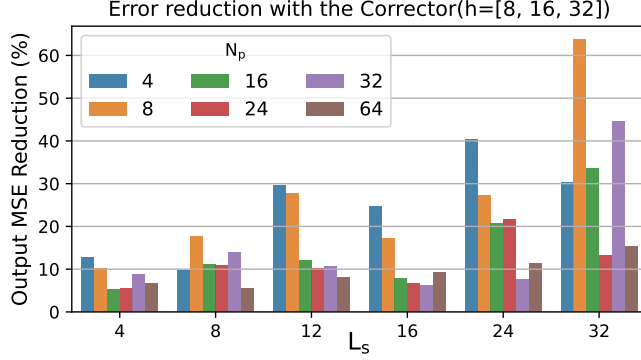
Fig. 6. Maximum observed error reduction with corrector: a 2-layer MLP with hidden dimension h. Longer and fewer prototypes benefit more on average.

size and FLOPs^{enc} linearly. Because of this reason, trading N_p for L_s is desirable even at the cost of some accuracy degradation (*e.g.* {$N_p = 4, L_s = 4$} vs {$N_p = 32, L_s = 8$}, the latter resulting in just a 1.13× lower error.).

3. Amortizing and reducing encoding costs. Computing the distances $\mathbf{d}^{(n)}$ between input columns and prototypes in the n-th subspace might come at a too high cost given that PQ only makes use of the index of the *closest* prototype to each column and not its actual value. A less computationally demanding distance could be used. Alternatively, we could further leverage $\mathbf{d}^{(n)}$ and better amortise FLOPs^{enc}. To this end, we design a lightweight MLP *corrector* that, given these distances, it learns a transformation that helps further reducing MSE^{out}. Intuitively, the corrector has a higher potential for correction with longer and fewer prototypes. This is the pattern in Figure 6 where we show the maximum observed reduction in MSE^{out} across all layers in the study. Even though a well tuned corrector can still benefit PQ layers with short prototypes, its computational footprint for a given hidden dimension h, dominated by $N_s N_p \times h$, does not justify its use. Therefore, we do not use the concept of a correct MLP in this work and we defer to future work to build on this idea.

5.3 Numerical Bitwidth Analysis

After training PQ-DNNs, we perform post-training quantization as described in Section 2.5. Figure 7 illustrates the impact of decreasing the bitwidth from 16 bits down to 2 bits for the distance calculator, product lookup, and when reducing the bitwidth for both. A validation set was used to identify the minimum and maximum values to dynamically compute the scale and offset values for quantization. We achieved better accuracy when using the $30th$ and $70th$ percentile values instead of the full range for computing the scale and offset values for our integer quantization—this enhanced version has been applied to MicroNet$_{KWS}$ in Figure 7. As Figure 7 shows, PQ operations can tolerate very low bitwidths (2–5 bits, depending on the DNN), especially when per-subspace quantization is used. Prototypes are easier to quantize than LUT$_{PQ}$. In the following evaluations, we tailor these bitwidths to each PQ-DNN, in addition to finding the ideal hardware vectorization parameters to maximize efficiency.

6 EXPERIMENTAL EVALUATION

This section investigates the performance and efficiency of PQA through both layerwise and end-to-end DNN execution. To guage the efficiency of PQA, we compare to an optimized deep learning accelerator (DLA) from prior work [1, 3]. In
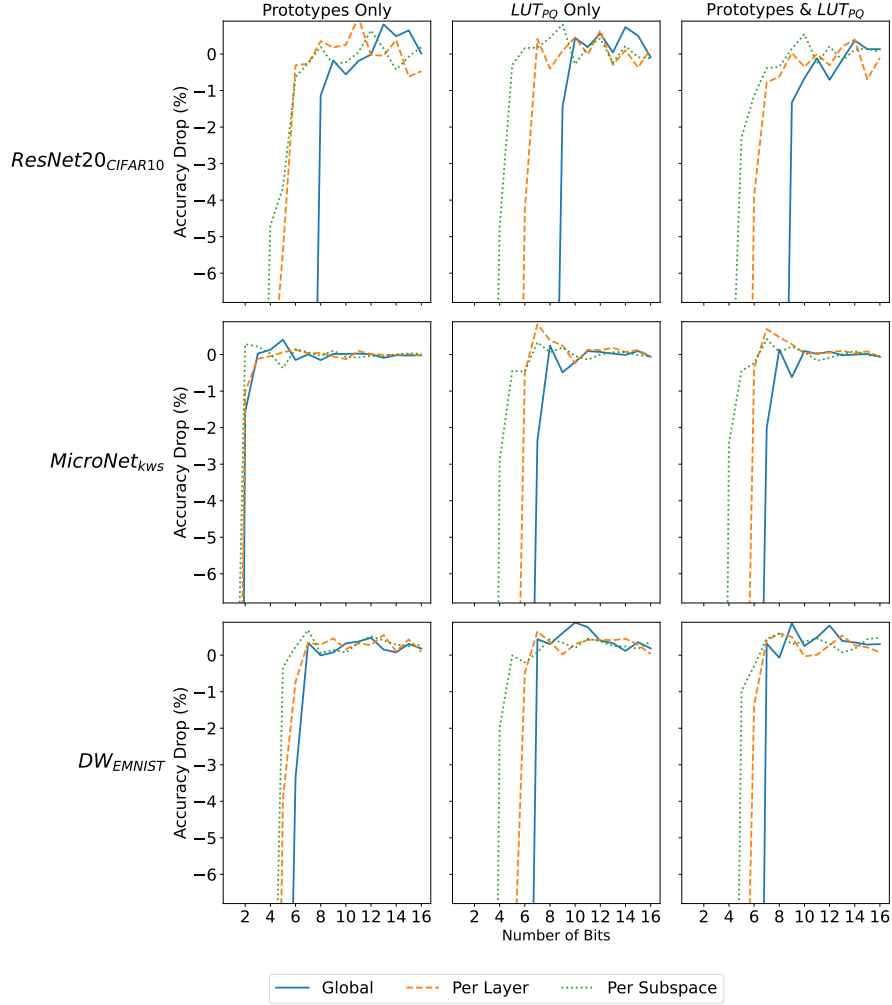
Fig. 7. Drop in accuracy due to quantization using different techniques for multiple models. **Global** means that a single scale and offset are used for whole model. **Per Layer** and **Per Subspace** mean that the scale and offset are different for each layer and each subspace respectively.

addition, we compare to the latest PQ work, PECAN [39]. Our results demonstrate both performance and efficiency improvements compared to both PECAN and DLA, paving the way for further work to establish the utility of PQ for DNN efficiency.

6.1 PQA Layerwise Performance Analysis

Figure 8 shows the speedup of running a convolution layer using PQ on a custom PQA compared to running its non-PQ equivalent on a custom DLA. In this study, all PQA vectorization parameters are set to 16. The convolution has a kernel of size 3×3 and the number of input channels is assumed to be the same as the number of output channels.
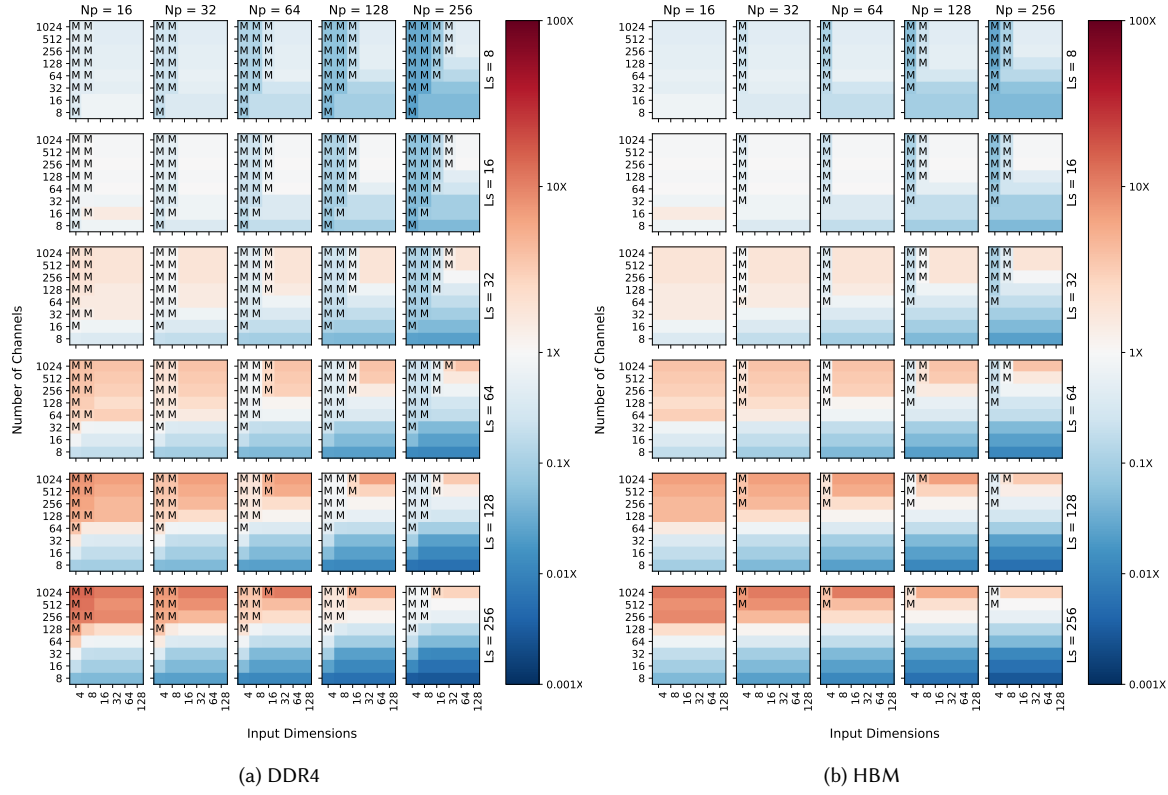
Fig. 8. Speedup of different $\{N_p, L_s\}$, input size and channels given a 3×3 kernel on PQA relative to DLA at 16 bits, when using (a) DDR4 and (b) HBM. M = Memory Bound.

As expected, smaller N_p leads to faster PQ execution as the cycles needed to compare prototypes are fewer. Larger N_p increases the LUT_{PQ} size and the time needed to load it from external memory, causing more points to become memory-bound. When using a High Bandwidth Memory (HBM) in Figure 8b, many memory bound configurations are no longer memory bound, for example $N_p = 16$ has no memory bound cells in HBM while more than half of the cells with input dimensions 4 and 8 are memory bound in DDR4. We can also see some of the cells that didn't result in a speedup in DDR4, resulting in a speedup in HBM because they are no longer memory bound. For example, in $L_s = 32$, the speedup area in all values of $N_p \geq 32$ in HBM includes smaller dimensions that didn't show a speedup in case of DDR4. When increasing L_s, the number of subspaces (N_s) decreases and speeds up overall execution—we observe speedups starting from $L_s = 32$. We can see a single row in $L_s = 16$ showing speedup which is the row that has the number of channels and N_p set to 16 as well. This is because our vectorization parameters are all set to 16 making these dimensions utilize the hardware very efficiently unlike the surrounding cells that do not match the vectorization parameters perfectly. Finally, we can see that larger layers benefit more from PQ with speedups up to 100× at small N_p. In general, more aggressive PQ quantization leads to less computation and memory and therefore a better speedup. It is clear that running PQ on custom hardware is not always advantageous, especially when compared to an optimized DLA. However, we have shown that unlike CPUs and GPUs, we are able to find layer sizes and PQ parameterizations
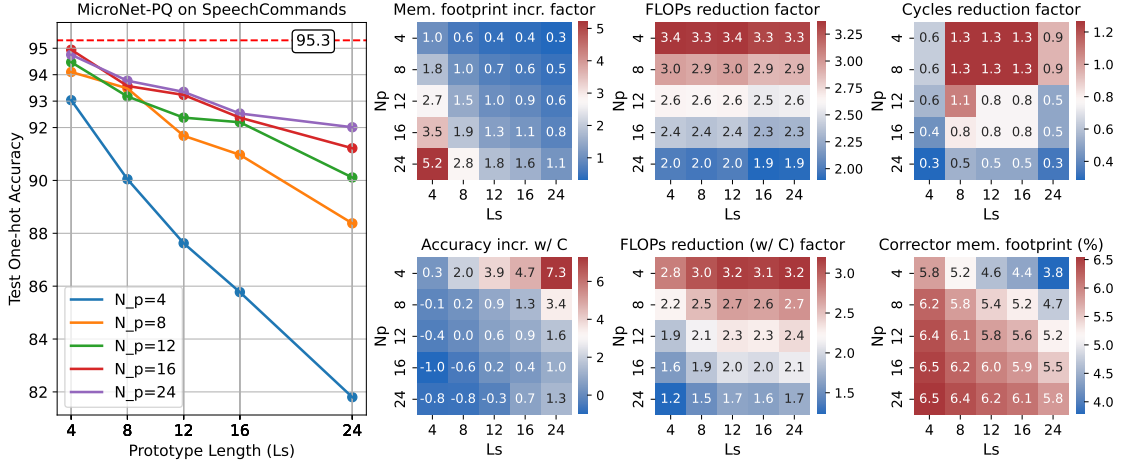
Fig. 9. MicroNet-PQ reaches 95.0% accuracy with $\{N_p = 16, L_s = 4\}$ leading to a 2.4× reduction in the number of inference FLOPs. However, fewer FLOPs does not always translate to faster inference (cycles). When increasing memory footprint is not possible ($\{N_p = 8, L_s = 8\}$), Micronet-PQ still reaches over 93.5% while resulting in nearly 3× lower computational footprint. With the corrector (displayed as 'w/C'), models with longer prototypes can reach that threshold accuracy too, at a small increase in memory footprint and FLOPs.

for which PQ can indeed outperform conventional convolutions. We have also shown that HBM alleviates memory bottlenecks and often favors PQA over DLA because of the reliance of PQ on relatively larger memory bandwidth.

6.2 MicroNet Case Study

Figure 9 sheds light on the accuracy, compute, memory, and latency implications of PQ with different parameterizations of L_s and N_p. The line plot in Figure 9 (left) shows the performance of MicroNet-PQ under varying configurations with just a 0.3% gap compared to the non-PQ baseline when $\{N_p = 16, L_s = 4\}$. The accompanying heatmaps highlight different trade-offs between MicroNet-PQ and their non-PQ counterpart. When the accuracy requirement is lower, longer prototypes (higher L_s) can be selected to achieve a better FLOPs speedup ratio. With the corrector introduced in Section 5.2, some of the accuracy drop introduced by longer prototypes can be recovered by up to 7.3% at a small memory footprint increase. Figure 9 highlights the theme of our analysis in Section 3: FLOPs and memory footprint are not a good proxy for hardware latency for PQ as shown in the top 3 heatmaps of the figure. Looking at the top row of heatmaps, we can see that a decrease in both FLOPs and memory footprint does not always result in a proportional decrease in hardware execution cycles. However, many PQ configurations, with N_{out}^{vec} set to 64, eventually lead to a speedup for PQA, as shown in the top right heatmap in the figure where the cycle reduction factor reaches 1.3.

6.3 Area, Latency and Frequency Trends of PQA

We study the effects of varying the inputs/prototype and LUT_{PQ} bitwidths on the two main PQ operations, distance calculation and product lookup, by sweeping bitwidths on a version of PQA for MicroNet. We used the PQ parameters $\{L_s = 4, N_P = 16\}$ that optimized accuracy and performance on MicroNet, and we used the same values for the hardware vectorization parameters $\{L_s^{vec} = 4, N_p^{vec} = 16\}$. We choose the remaining vectorized parameters for performance considerations and on-chip buffer sizes according to the MicroNet$_{KWS}$ layer parameters [5]: $N_s^{vec} = 16$, $N_{out}^{vec} = 16$, $\{L_s^{max} = 4,$
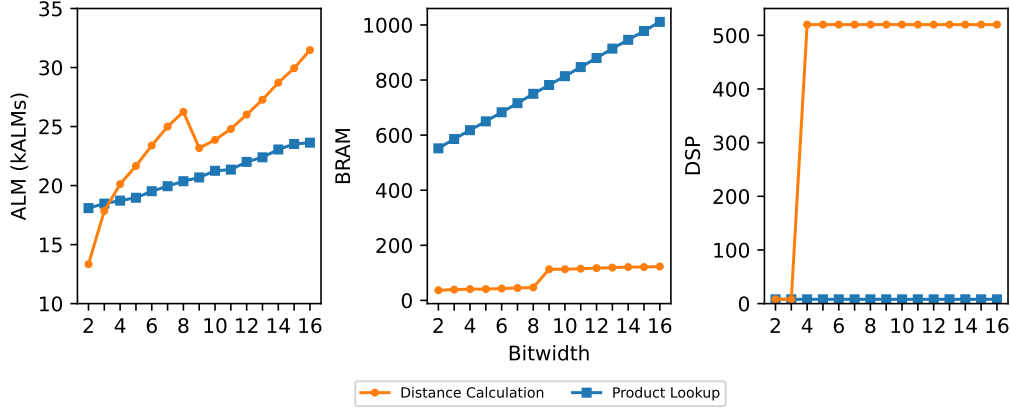
Fig. 10. Area trends of the distance calculation operation with varying prototype bitwidths and product lookup operation with varying LUT_{PQ} bitwidths. All PQA vectorization parameters are set to 16.

N_s^{max}=32, N_p^{max}=32, N_{out}^{max}=256, N_{in}^{max}=128}. In Figure 10, we use 16 bits as a baseline, and we vary the bitwidths of the distance calculators and product lookup portions of PQA to quantify the impact on FPGA resources utilization. When quantizing the distance calculators, both the inputs and the prototypes share the same bitwidth.

Frequency remained relatively stable for each network when varying either the prototype or LUT bitwidth, according to Table 3, and was often limited by the OpenCL *shell* or board support package (BSP) that contains the infrastructure logic to connect PQA to PCIe and DDR4. Frequency of DW$_{EMNIST}$ is notably slower due to the its larger area consumption.

Lower bitwidths for both distance calculation and product lookup significantly decrease area usage compared to the baseline as shown in Figure 10. The distance calculation has a steadily increasing linear trend for ALMs when varying the prototype bitwidth. The offset in the trend at 9 bits is attributed to the synthesis of the input buffer as BRAM instead of MLABs as the input size increases. For 2–3 bit wide inputs and prototypes, 0 DSPs are synthesized due to the low arithmetic complexity.

When analyzing area trends in the product lookup operation, we account for both the lookup and accumulate kernels in order to capture any savings in the adder tree area when quantizing LUT_{PQ} entries. The number of ALMs increases linearly due to both the increased accumulation size and the larger pipelined "never-stall" load store unit (LSU). The increase in BRAM is mainly attributed to a larger LUT_{PQ} at higher bitwidths. Additionally, some BRAM is also utilized for storing the accumulator state as well but that does not increase with bitwidth as the accumulators are always fixed at 16 bits.

As Figure 7 shows, lower bitwidths can be used without substantially impacting accuracy, especially with per-subspace quantization parameters. Per-subspace scale and offset results in the best accuracy at lower bitwidths while, at higher bitwidths, both per layer and even global scale and offset values yield comparable accuracy. By carefully adjusting the scale factors and clipping thresholds during per-subspace quantization, MicroNet$_{KWS}$ can use as little as 2 bits without any discernible effect on accuracy. We use the combined results of Figures 7 and 10 to customize three PQA variants for our three PQ-DNNs: MicroNet$_{KWS}$, DW$_{EMNIST}$, and ResNet20$_{CIFAR10}$.

Table 3. Results of our PQ on multiple networks. Hardware latency is shown for DLA (baseline)[1] and PQA for two differerent external memory chips: DDR4 and HBM with 36 GB/s and 460 GB/s bandwidth respectively. PECAN-D[39] is the current state of the art in PQ. PQA-Q is the quantized version of our accelerator with its parameters as bitwidths of prototypes and product tables respectively.

Model	Setting	Ls	Np	Param.	Acc (%)	Fmax (MHz)	Area (keALMs)	Latency (us)		Performance/Area (input/s/keALM)	
								DDR4	HBM	DDR4	HBM
ResNet20$_{CIFAR10}$	Baseline	–	–	269k	92.6	485	290	37	36	94	95
	PECAN-D	3	64	5.7M	87.9	451	217	448	276	10	17
		9	64	1.93M	85.0	482	227	143	86	31	51
	PQA	9	16	476k	84.4	490	225	35	24	127	185
		9	8	239k	84.1	471	174	28	25	207	230
	PQA-Q (6,5)	9	16	476k	84.1	487	186	24	24	222	222
		9	8	239k	83.8	475	138	25	25	**291**	291
MicroNet$_{KWS}$	Baseline	–	–	61k	95.3	485	290	4	4	885	885
	PQA	4	16	212k	95.0	465	183	11	11	496	509
		8	8	63k	93.6	428	174	6	6	983	983
	PQA-Q (2,6)	4	16	212k	94.7	481	119	10	10	808	808
		8	8	63k	93.3	473	118	5	5	**1599**	1599
DW$_{EMNIST}$	Baseline	–	–	1.1M	90.4	485	290	47	13	74	259
	PQA	4	12	3.1M	87.6	288	746	231	51	6	26
		8	8	1.1M	85.8	287	399	82	27	31	93
	PQA-Q (5,5)	4	12	3.1M	86.8	274	449	91	53	25	42
		8	8	1.1M	85.0	306	341	34	25	**87**	116

6.4 PQA Acceleration Potential

In Table 3, we bring together our improved PQ training, efficient PQ-DNNs, and hardware PQA accelerator to assess the current state of product quantization on our three DNNs: ResNet20$_{CIFAR10}$, MicroNet$_{KWS}$, and DW$_{EMNIST}$. We use DLA [1, 3] with conventional DNN execution as a baseline to which we can compare both the accuracy and performance of PQ-DNNs on PQA. Furthermore, we are able to compare our ResNet20 PQ-DNN directly to prior work from PECAN [39]—the closest work on PQ in the literature. To optimize PQA performance, we modified its vectorization parameters by setting N_{out}^{vec} to 32 and by keeping other parameters at the minimum of 16 and the closest power of two of L_s and N_p as having a $L_s^{vec} > L_s$ or $N_p^{vec} > N_p$ is a waste of resources. We additionally present results with the best quantization parameters discussed from Section 5.3, denoted with PQA-Q in Table 3. Both designs have a higher Fmax and lower area with almost no accuracy drop, indicating the importance of reduced bitwidths to improve PQA performance and area. It's worth noting that the vectorization parameters can be further increased to reach even better performance in the cases where PQA-Q is compute-bound. For example, changing N_s^{vec} from 16 to 32 in PQA-Q for *MicroNet$_{KWS}$* in case of $L_s = 4$ and $N_p = 16$ improves the Performance/Area by around 1.5X showing that further vectorization parameters tuning can lead to even better gains in PQA.

Significant speedup is possible with PQ at lower accuracy. Focusing on ResNet20$_{CIFAR10}$, our largest DNN, we find that we are able to outperform both the baseline DLA, and PECAN considerably. Specifically, our best PQA-Q

architectures achieved a 3.1× and 9.4× improvement in overall performance/area compared to the baseline DLA and PECAN alternatives. While there is still a considerable gap to conventional DNNs in terms of accuracy, our improvements over PECAN-D comes at only 0.6% accuracy degradation (at 4× performance/area boost) or up to 1.2% degradation (with 9× performance/area boost). This is achieved due to the smaller N_p and larger L_s used and the efficiency of the optimized PQA-Q in performing the needed computations when compared to the conventional DLA that requires lots of multiplications. Our proposed efficient PQ-DNNs and PQA design therefore present a compelling accuracy-efficiency tradeoff for implementation on constrained edge devices. Improvements can be seen in other DNNs as well where our best PQA-Q architectures achieve 81% and 18% improvement in overall performance/area for MicroNet$_{KWS}$ and DW$_{EMNIST}$ respectively. Notably, the accuracy drop is lowest with MicroNet$_{KWS}$ with as little as 0.3% degradation for some of our parameterizations.

Lower bitwidth alleviates memory bandwidth bottlenecks. As we have seen in our per-layer analysis, HBM favours PQA more than DLA because of the higher external memory bandwidth demands of PQ-DNNs. However, when lower bitwidths were used with PQA, this is not the case anymore and the memory bandwidth bottleneck was alleviated as shown by the identical performance for both DDR4 and HBM for PQA-Q design points.

All considered, we were able to, for the first time, demonstrate a hardware speedup for PQ-DNNs, even when comparing against an optimized systolic array-based DLA. Our results have shown that, while there is still an accuracy gap to conventional DNNs, our choice of PQ parameters, training improvements, and the *accuracy corrector* can help decrease the accuracy degradation. Our novel PQA architecture has helped to highlight that significant hardware speedup may be possible even if CPU and GPU architectures are not a good fit, especially when customizing the bitwidths for this new DNN computing paradigm as we have shown in our work.

7 RELATED WORK

Product Quantization. Standard quantization approaches perform a scalar-to-scalar mapping while product quantization (PQ) operates with higher-dimensionality, mapping vectors to vectors [17, 29]. This has made PQ a good fit for applications such as image retrieval [25, 31, 50] and compression [14, 43]. Different from those works is [8], where PQ is used to accelerate matrix-matrix multiplications in a two-step process: columns of the input are mapped to *prototypes*, a set of learnable vectors; then these vectors are used to construct a look-up table of pre-computed dot products between prototypes and the layer weights. At inference time, the pre-computed values can be retrieved by mapping each input column to its closest prototype, trading numerical degradation for larger speedups in some cases. Follow up work extends PQ to an entire fully connected network and provides a preliminary analysis on the overheads of accelerating PQ [35]. A more generalised implementation of PQ is PECAN [39], which not only applies PQ to CNNs but also proposes a distance metric for input-to-prototype encoding that does not require multiplications. However, this method resulted in severe memory overheads since minimising accuracy degradation was prioritized. Unlike prior work, we perform a broader and systematic study of the impact of PQ settings in terms of memory, compute, and accuracy which we then use to inform our hardware accelerator design. Furthermore, we present the first hardware architecture to accelerate PQ-DNNs, and we demonstrate significant performance gains compared to prior work and to conventional DNNs.

Hardware Acceleration of DNNs. Many custom hardware accelerators have been recently developed, especially for DNNs. These accelerators outperform conventional CPUs and GPUs by leveraging DNN-specific properties in their hardware architectures [11, 15, 16]. For example, Google's TPU [28] uses a 2D systolic array of multiply-accumulate units that can more directly and efficiently transfer data between compute units instead of expensive synchronization over GPU register files. Another example is Groq's TSP [2], which utilizes spatial compute units to enable the construction

of custom compute engines for each DNN layer. Many prior works have used field-programmable gate-arrays (FPGAs) to build accelerators for deep learning, leveraging low precision, sparsity, a custom memory hierarchy, or novel dataflows [1, 3, 19].

In the literature, other accelerators based on similar quantization techniques like vector quantization have been proposed [34]. However, this work is fundamentally different than our work. Their vector quantization algorithm is different from our product quantization algorithm. Specifically, we quantize activations on the fly and leave weights unquantized, while prior work[34] quantizes weights and does not quantize activations. This significantly changes the hardware design and has implications on accuracy which are not quantified in the prior work unlike our work which focuses on finding product quantization parameters and optimizing the training process to achieve good accuracy. Prior work is also based on STD cell ASIC implementation, and area estimates are only based on post-synthesis results and without SRAM area included, as shown in the footnote in Table 1. However, the PQ accelerator (both in our work and in [34]) is heavily-based on SRAM for product lookup. This makes the results in prior work much more difficult to compare with non-PQ accelerators. In contrast, our work is based on FPGAs and we are able to directly compare to a widely-used conventional deep learning accelerator to understand the true efficiency gap between the two approaches (PQA vs DLA).

To the best of our knowledge, none of the existing work has attempted to accelerate PQ using custom hardware[2], nor were there any studies of PQ efficiency using a custom accelerator. Our work aims to fill this gap by presenting the first product quantization accelerator (PQA).

8 CONCLUSION

In this work we have identified several practical limitations that prevent PQ from being treated like other, more mature, optimizations techniques for DNN acceleration. As evidenced throughout our study, PQ requires a careful tuning to avoid incurring large overheads, especially in terms of memory and compute footprint. Unlike CPUs and GPUs, our custom hardware PQA has demonstrated that PQ can indeed accelerate entire DNNs but not as much as higher-level proxy metrics such as FLOPs might suggest. PQA is 3.1× more efficient than a conventional DLA in terms of performance/area on ResNet20. To our knowledge, this is the first time a hardware speedup for PQ was demonstrated on any hardware platform. Furthermore, by codesigning PQ parameters and hardware, our PQ ResNet20 is 4× better in performance per area than the most recent PQ work with just 0.6% lower accuracy. We also demonstrated the use of lower bitwidths through linear quantization on top of PQ to further decrease hardware area with low drop in accuracy. In the future, we plan to investigate more robust and faster methods for PQ training, and explore more *complementary* compression methods such as channel pruning.

9 LIMITATIONS AND FUTURE WORK

In this work we made use of PQ to accelerate relatively small ML models for image classification and keyword spotting. Their size and complexity allowed us to consider a broad range of experiments, including an extensive per-layer analysis. An obvious extension of our work would be to apply PQ to much larger models. Even though our evaluation focuses on relatively small networks and datasets, these networks are not over-parameterised for their respective tasks, meaning that achieving further compression is challenging without incurring accuracy degradation. This being said, compression is not the only objective of PQ. Our objective is to speedup inference through PQ without incurring a large increase in memory footprint due to the large lookup tables. Currently, training PQ models requires much longer training times

---

[2]With the exception of a 1-page abstract at FCCM 2018 [52]. However, without a description of the architecture.

and memory utilization during the input-to-protoype encoding stage that require computing the distances of each input column of each subspace to each prototype. For example, training the MicroNet with PQ for SpeechCommands takes approximately 6 hours for one run, already making it challenging to produce most of the results in our paper. Not to mention, that PQ is highly sensitive to hyperparameter settings, which required us to run hundreds of hyperparameter search steps for each architecture-dataset pair.

The use of PQ is complementary to other forms of accelerating inference. For example, using quantization (as discussed in Section 5.3) could be used to reduce the computational footprint of the encoding stage as well as the overheads of storing and reading from LUT_{PQ} and the external memory bandwidth limitations. This opens the door for exploring designs with higher vectorization parameters which may lead to higher speedups. In addition to that, Structured pruning, in particular channel pruning [7, 22], can be applied first to a model and then obtain its PQ representation. With channel pruning, the number of output channels is reduced and so the acceleration potential of PQ increases. This was shown analytically when analysing the trade-offs of PQ and empirically in our hardware layer-wise analysis.

ACKNOWLEDGMENTS

REFERENCES

[1] Mohamed S. Abdelfattah, David Han, Andrew Bitar, Roberto Dicecco, Shane O'Connell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, Andrew C. Ling, and Gordon R. Chiu. 2018. DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration. *28th International Conference on Field Programmable Logic and Applications (FPL)* (2018), 411–4117.

[2] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E. R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. 2020. Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 145–158. https://doi.org/10.1109/ISCA45697.2020.00023

[3] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C Ling, and Gordon R Chiu. 2017. An opencl™ deep learning accelerator on arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 55–64.

[4] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. 2021. MicroNets: Neural Network Architectures for Deploying TinyML Applications on Commodity Microcontrollers. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 517–532.

[5] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. 2021. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. (2021), 517–532.

[6] Axel Berg, Mark O'Connor, and Miguel Tairum Cruz. 2021. Keyword Transformer: A Self-Attention Model for Keyword Spotting. In *Proc. Interspeech 2021*. 4249–4253. https://doi.org/10.21437/Interspeech.2021-1286

[7] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. 2020. What is the State of Neural Network Pruning?. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 129–146. https://proceedings.mlsys.org/paper/2020/file/d2ddea18f00665ce8623e36bd4e3c7c5-Paper.pdf

[8] Davis Blalock and John Guttag. 2021. Multiplying Matrices Without Multiplying. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 992–1004.

[9] Andrew Brock, Theodore Lim, J. M. Ritchie, and Nick Weston. 2016. Neural Photo Editing with Introspective Adversarial Networks. https://doi.org/10.48550/ARXIV.1609.07093

[10] Hanting Chen, Yunhe Wang, Chunjing Xu, Boxin Shi, Chao Xu, Qi Tian, and Chang Xu. 2020. AdderNet: Do We Really Need Multiplications in Deep Learning?. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[11] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308. https://doi.org/10.1109/JETCAS.2019.2910232

[12] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. 2017. EMNIST: Extending MNIST to handwritten letters. *2017 International Joint Conference on Neural Networks (IJCNN)* (2017). https://doi.org/10.1109/ijcnn.2017.7966217

[13] S. Davis and P. Mermelstein. 1980. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 28, 4 (1980), 357–366. https://doi.org/10.1109/TASSP.1980.1163420

[14] Alaaeldin El-Nouby, Matthew J. Muckley, Karen Ullrich, Ivan Laptev, Jakob Verbeek, and Hervé Jégou. 2022. Image Compression with Product Quantized Masked Image Modeling. https://doi.org/10.48550/ARXIV.2212.07372

[15] H. Fan, T. Chau, S. I. Venieris, R. Lee, A. Kouris, W. Luk, N. D. Lane, and M. S. Abdelfattah. 2022. Adaptable Butterfly Accelerator for Attention-based NNs via Hardware and Algorithm Co-design. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA, USA, 599–615. https://doi.org/10.1109/MICRO56248.2022.00050

[16] H. Fan, T. Chau, S. I. Venieris, R. Lee, A. Kouris, W. Luk, N. D. Lane, and M. S. Abdelfattah. 2022. Adaptable Butterfly Accelerator for Attention-based NNs via Hardware and Algorithm Co-design. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA, USA, 599–615. https://doi.org/10.1109/MICRO56248.2022.00050

[17] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized Product Quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36, 4 (2014), 744–755. https://doi.org/10.1109/TPAMI.2013.240

[18] Junli Gu, Yibing Liu, Yuan Gao, and Maohua Zhu. 2016. OpenCL Caffe: Accelerating and Enabling a Cross Platform Machine Learning Framework. In *Proceedings of the 4th International Workshop on OpenCL* (Vienna, Austria) *(IWOCL '16)*. ACM, New York, NY, USA, Article 8, 5 pages. https://doi.org/10.1145/2909437.2909443

[19] Mathew Hall and Vaughn Betz. 2020. HPIPE: Heterogeneous Layer-Pipelined and Sparse-Aware CNN Inference for FPGAs. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 320. https://doi.org/10.1145/3373087.3375380

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Jun 2016). https://doi.org/10.1109/cvpr.2016.90

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[22] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel Pruning for Accelerating Very Deep Neural Networks. arXiv:1707.06168 [cs.CV]

[23] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (Jun 2018). https://doi.org/10.1109/cvpr.2018.00286

[24] Eric Jang, Shixiang Gu, and Ben Poole. 2016. Categorical Reparameterization with Gumbel-Softmax. https://doi.org/10.48550/ARXIV.1611.01144

[25] Young Kyun Jang and Nam Ik Cho. 2020. Generalized Product Quantization Network for Semi-Supervised Image Retrieval. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[26] Pranav Jeevan, Kavitha Viswanathan, Anandu A S, and Amit Sethi. 2022. WaveMix: A Resource-efficient Neural Network for Image Analysis. https://doi.org/10.48550/ARXIV.2205.14375

[27] Yangqing Jia. 2014. *Learning Semantic Image Representations at a Large Scale*. Ph. D. Dissertation. EECS Department, University of California, Berkeley.

[28] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (jun 2017), 1–12. https://doi.org/10.1145/3140659.3080246

[29] Herve Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33, 1 (2011), 117–128. https://doi.org/10.1109/TPAMI.2010.57

[30] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. https://doi.org/10.48550/ARXIV.1412.6980

[31] Benjamin Klein and Lior Wolf. 2019. End-To-End Supervised Product Quantization for Image Search and Retrieval. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[32] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[33] Andrew Lavin and Scott Gray. 2016. Fast Algorithms for Convolutional Neural Networks. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Jun 2016). https://doi.org/10.1109/cvpr.2016.435

[34] Heng Lee, Yi-Heng Wu, Yu-Sheng Lin, and Shao-Yi Chien. 2019. Convolutional Neural Network Accelerator with Vector Quantization. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–5. https://doi.org/10.1109/ISCAS.2019.8702105

[35] Calvin McCarter and Nicholas Dronen. 2022. Look-ups are not (yet) all you need for deep learning inference. https://doi.org/10.48550/ARXIV.2207.05808

[36] Sachin Mehta and Mohammad Rastegari. 2022. MobileViT: Light-weight, General-purpose, and Mobile-friendly Vision Transformer. In *International Conference on Learning Representations*.

[37] Junting Pan, Adrian Bulat, Fuwen Tan, Xiatian Zhu, Lukasz Dudziak, Hongsheng Li, Georgios Tzimiropoulos, and Brais Martinez. 2022. EdgeViTs: Competing Light-weight CNNs on Mobile Devices with Vision Transformers. In *European Conference on Computer Vision*.

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS)*. 8026–8037.

[39] Jie Ran, Rui Lin, Jason Chun Lok Li, Jiajun Zhou, and Ngai Wong. 2022. PECAN: A Product-Quantized Content Addressable Memory Network. https://doi.org/10.48550/ARXIV.2208.13571

[40] Rafat Rashid, J Gregory Steffan, and Vaughn Betz. 2014. Comparing performance, productivity and scalability of the TILT overlay processor to OpenCL HLS. In *2014 International Conference on Field-Programmable Technology (FPT)*. IEEE, 20–27.

[41] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *CoRR* abs/1603.05279 (2016). arXiv:1603.05279

[42] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (Jun 2018). https://doi.org/10.1109/cvpr.2018.00474

[43] Pierre Stock, Armand Joulin, Rémi Gribonval, Benjamin Graham, and Hervé Jégou. 2020. And the Bit Goes Down: Revisiting the Quantization of Neural Networks. In *International Conference on Learning Representations (ICLR)*.

[44] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. arXiv:1905.11946

[45] Michael Tschannen, Aran Khanna, and Animashree Anandkumar. 2018. StrassenNets: Deep Learning with a Multiplication Budget. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, Stockholmsmässan, Stockholm Sweden, 4985–4994.

[46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.

[47] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. HAQ: Hardware-Aware Automated Quantization With Mixed Precision. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[48] Pete Warden. 2018. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209* (2018).

[49] Di Wu, Yu Zhang, Xijie Jia, Lu Tian, Tianping Li, Lingzhi Sui, Dongliang Xie, and Yi Shan. 2019. A High-Performance CNN Processor Based on FPGA for MobileNets. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 136–143. https://doi.org/10.1109/FPL.2019.00030

[50] Tan Yu, Junsong Yuan, Chen Fang, and Hailin Jin. 2018. Product Quantization Network for Fast Image Retrieval. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

[51] Jingzhao Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie. 2020. Why Gradient Clipping Accelerates Training: A Theoretical Justification for Adaptivity. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJgnXpVYwS

[52] Jialiang Zhang and Jing Li. 2018. PQ-CNN: Accelerating Product Quantized Convolutional Neural Network on FPGA. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 207–207. https://doi.org/10.1109/FCCM.2018.00041

[53] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. 2018. Hello Edge: Keyword Spotting on Microcontrollers. arXiv:1711.07128 [cs.SD]

Table 4. The DW$_{\text{EMNIST}}$ architecture used in the per-layer study designed for EMNIST image classification. Pointwise convolutions dominate the compute footprint of this model. Because of this, only point-wise convolutions are considered to be replaced with PQ – for which we show the shapes of unrolled inputs and weights. This model has a total of 1.05M parameters and requires 50.1M FLOPs per $1 \times 28 \times 28$ input.

Layer	Input Shape	Unrolled Inputs	Unrolled Weights	Parameters	FLOPs	FLOPs (%)
Conv	[1, 1, 28, 28]	–	–	640	1,003,520	2.00
DepthW-1	[1, 64, 28, 28]	–	–	576	225,792	0.45
PointW-1	[1, 64, 14, 14]	[1, 64, 196]	[96, 64]	6,144	2,408,448	4.81
DepthW-2	[1, 96, 14, 14]	–	–	864	338,688	0.68
PointW-2	[1, 96, 14, 14]	[1, 96, 196]	[120, 96]	11,520	4,515,840	9.01
DepthW-3	[1, 120, 14, 14]	–	–	1,080	423,360	0.85
PointW-3	[1, 120, 14, 14]	[1, 120, 196]	[150, 120]	18,000	7,056,000	14.08
DepthW-4	[1, 150, 14, 14]	–	–	1,350	132,300	0.26
PointW-4	[1, 150, 7, 7]	[1, 150, 49]	[187, 150]	28,050	2,748,900	5.49
DepthW-5	[1, 187, 7, 7]	–	–	1,683	164,934	0.33
PointW-5	[1, 187, 7, 7]	[1, 187, 49]	[234, 187]	43,758	4,288,284	8.56
DepthW-6	[1, 234, 7, 7]	–	–	2,106	206,388	0.41
PointW-6	[1, 234, 7, 7]	[1, 234, 49]	[292, 234]	68,328	6,696,144	13.37
DepthW-7	[1, 292, 7, 7]	–	–	2,628	84,096	0.17
PointW-7	[1, 292, 4, 4]	[1, 292, 16]	[366, 292]	106,872	3,419,904	6.83
DepthW-8	[1, 366, 4, 4]	–	–	3,294	105,408	0.21
PointW-8	[1, 366, 4, 4]	[1, 366, 16]	[457, 366]	167,262	5,352,384	10.68
DepthW-9	[1, 457, 4, 4]	–	–	4,113	131,616	0.26
PointW-9	[1, 457, 4, 4]	[1, 457, 16]	[572, 457]	261,404	8,364,928	16.70
DepthW-10	[1, 572, 4, 4]	–	–	5,148	41,184	0.08
PointW-10	[1, 572, 2, 2]	[1, 572, 4]	[512, 572]	292,864	2,342,912	4.68
Linear	[1, 512]	–	–	24,111	48,222	0.10

A MODEL ARCHITECTURES

The per-layer study presented in the main paper made use of a CNN with 10 depth-wise separable layers. We designed this network, that we name DW$_{\text{EMNIST}}$, to be lightweight but deep enough so we could conduct the study considering a very large set of PQ-related parameters, $\{N_p, L_s, \tau\}$, and training related hyperparamters (batch sizes, learning rates, etc). A detailed description of the network is presented in Table 4. This network has a total of 1.05M paramters and requires 50.1M FLOPs when implemented as im2col. In Section 6 we analyse how a NAS-optimized MicroNet [4][3] for keyword spotting performs under different PQ settings. We refer to this network as MicroNet-PQ and a per-layer breakdown of parameters and shapes is provided in Table 5. This MicroNet baseline is an INT8 model designed to be run on microcontrollers. The last model considered in this work is a ResNet20 [21] for CIFAR-10. This is the same model evaluated in PECAN [39], and Table 6 provides a detailed view of this architecture.

B EXTENDED PQ IMPLEMENTATION TRADE-OFFS

In Fig. 11 we show an extended version of the results. Two main trends become evident in this visualisation. First, at deeper layers, where the spatial dimensions of the input tend to be smaller (*i.e.* leading to fewer columns in the unrolled input) the impact on Output MSE of having more prototypes, higher N_p, is less pronounced. For example, in layer 2 and short prototypes $L_s = 4$ having $N_p = 64$ compared to $N_p = 8$ offers an Output MSE 2.4× smaller. In layer 8, this difference

---

[3]open sourced in: https://github.com/ARM-software/ML-zoo/tree/master/models/keyword_spotting/micronet_small/tflite_int8

Table 5. The MicroNet-PQ architecture designed for the SpeechCommands dataset. Pointwise convolutions dominate the compute footprint of this model, accounting for almost 75% of the FLOPs. Because of this, only point-wise convolutions are considered to be replaced with PQ – for which we show the shapes of unrolled inputs and weights. This model has a total of 60.6K parameters and requires 17.01M FLOPs per $1 \times 10 \times 49$ input. This model corresponds with MicroNet-KWS-S [4]. We modified the number of channels of the second depth-wise block from 112 to 120 so it's divisible with common L_s choices used throughout this work.

Layer	Input Shape	Unrolled Inputs	Unrolled Weights	Parameters	FLOPs	FLOPs (%)
Conv	[1, 1, 10, 49]	–	–	3,444	3,375,120	19.75
DepthW-1	[1, 84, 10, 49]	–	–	756	189,000	1.11
PointW-1	[1, 84, 5, 25]	[1, 84, 125]	[120, 84]	10,080	2,520,000	14.75
DepthW-2	[1, 120, 5, 25]	–	–	1,080	270,000	1.58
PointW-2	[1, 120, 5, 25]	[1, 120, 125]	[84, 120]	10,080	2,520,000	14.75
DepthW-3	[1, 84, 5, 25]	–	–	756	189,000	1.11
PointW-3	[1, 84, 5, 25]	[1, 84, 125]	[84, 84]	7,056	1,764,000	10.32
DepthW-4	[1, 84, 5, 25]	–	–	756	189,000	1.11
PointW-4	[1, 84, 5, 25]	[1, 84, 125]	[84, 84]	7,056	1,764,000	10.32
DepthW-5	[1, 84, 5, 25]	–	–	756	189,000	1.11
PointW-5	[1, 84, 5, 25]	[1, 84, 125]	[196, 84]	16,464	4,116,000	24.08
Linear	[1, 196]	–	–	2,364	4,728	0.03

Table 6. The ResNet20 architecture for CIFAR-10. All layers in the network with the exception of the input convolution and output linear layer are replaced with their PQ counterparts in our evaluation. For these we show the shapes of unrolled inputs and weights. This model has a total of 268K parameters and requires 81.1M FLOPs per $3 \times 32 \times 32$ input.

Layer	Input Shape	Unrolled Inputs	Unrolled Weights	Parameters	FLOPs	FLOPs (%)
Conv	[1, 3, 32, 32]	–	–	432	884,736	1.09
Block1-Conv1	[1, 16, 32, 32]	[1, 144, 1024]	[16, 144]	2,304	4,718,592	5.82
Block1-Conv2	[1, 16, 32, 32]	[1, 144, 1024]	[16, 144]	2,304	4,718,592	5.82
Block1-Conv3	[1, 16, 32, 32]	[1, 144, 1024]	[16, 144]	2,304	4,718,592	5.82
Block1-Conv4	[1, 16, 32, 32]	[1, 144, 1024]	[16, 144]	2,304	4,718,592	5.82
Block1-Conv5	[1, 16, 32, 32]	[1, 144, 1024]	[16, 144]	2,304	4,718,592	5.82
Block1-Conv6	[1, 16, 32, 32]	[1, 144, 1024]	[16, 144]	2,304	4,718,592	5.82
Block2-Conv1	[1, 16, 32, 32]	[1, 144, 256]	[32, 144]	4,608	2,359,296	2.91
Block2-Conv2	[1, 32, 16, 16]	[1, 288, 256]	[32, 288]	9,216	4,718,592	5.82
Block2-Conv3	[1, 32, 16, 16]	[1, 288, 256]	[32, 288]	9,216	4,718,592	5.82
Block2-Conv4	[1, 32, 16, 16]	[1, 288, 256]	[32, 288]	9,216	4,718,592	5.82
Block2-Conv5	[1, 32, 16, 16]	[1, 288, 256]	[32, 288]	9,216	4,718,592	5.82
Block2-Conv6	[1, 32, 16, 16]	[1, 288, 256]	[32, 288]	9,216	4,718,592	5.82
Block3-Conv1	[1, 32, 16, 16]	[1, 288, 64]	[64, 288]	18,432	2,359,296	2.91
Block3-Conv2	[1, 64, 8, 8]	[1, 576, 64]	[64, 576]	36,864	4,718,592	5.82
Block3-Conv3	[1, 64, 8, 8]	[1, 576, 64]	[64, 576]	36,864	4,718,592	5.82
Block3-Conv4	[1, 64, 8, 8]	[1, 576, 64]	[64, 576]	36,864	4,718,592	5.82
Block3-Conv5	[1, 64, 8, 8]	[1, 576, 64]	[64, 576]	36,864	4,718,592	5.82
Block3-Conv6	[1, 64, 8, 8]	[1, 576, 64]	[64, 576]	36,864	4,718,592	5.82
Linear	[1, 64]	–	–	650	1,300	0.00

is reduced to 1.7×. The second trend has to do with the costs of performing the input-to-prototype indexing, or in other words, computing the distance of each input column with the bank of prototypes in a given subspace. Due to the larger spatial dimensions in early layers in the network, the encoding costs $\text{FLOPS}^{\text{enc}}$ are much larger for a constant $\{N_p, L_s\}$ configuration than in deeper layers. For example for $\{N_p = 64, L_s = 8\}$, $\text{FLOPS}^{\text{enc}}$ is 3.2× larger in layer 2 than layer 8.
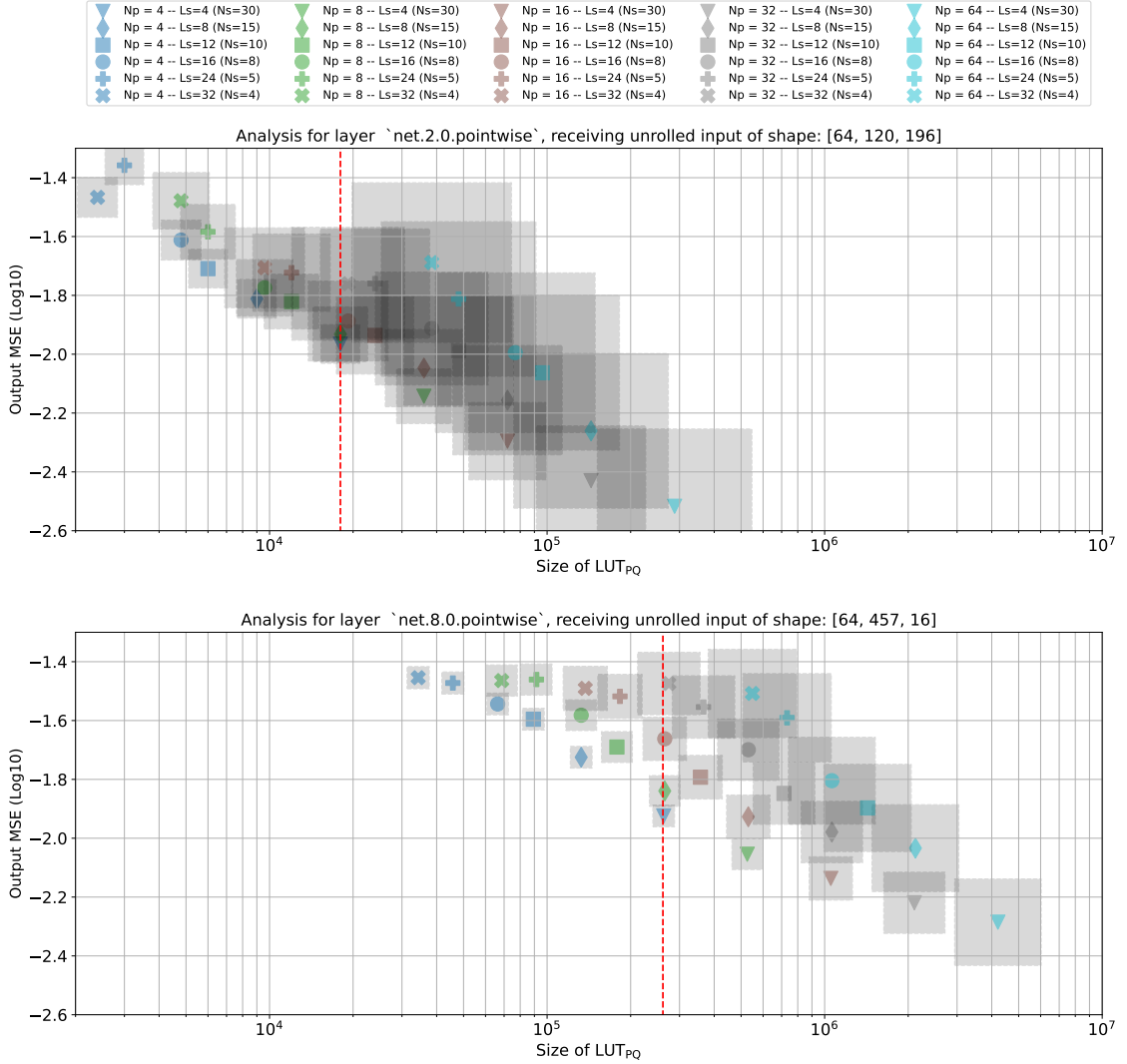
Fig. 11. Analysis of the quality of the output generated by a PQ layer compared to the output of a non-PQ layer of a pre-trained model. The unit in the x-axis is number of parameters and we map to it the size of the resulting look up table, LUT_{PQ}, as well as the number of parameters that the same non-PQ layer has (this is shown as a vertical dashed red line). In addition to different memory footprints, each $\{N_p, L_s\}$ pair also translates into different compute overheads when performing the input encoding. This is represented by the grey-shaded squares whose areas are proportional to $FLOPS^{enc}$.

C MEMORY AND COMPUTE FOOTPRINT OF PQ

We presented our study on the PQ trade-offs, we derived an expression for the speedup in terms of FLOPs achievable by PQ over im2col-equivalent convolution in Section 5.1. In Fig 12 we show how that expression maps to different speedups when varying N_p, L_s, and C_{out}. Having fewer prototypes (*i.e.* a smaller N_p) has a larger impact than designing PQ with longer prototypes (*i.e.* larger L_s), although this is also desirable. Another important observation is that PQ

might not always be faster than im2col. This is true for convolutional layers with few output channels ($C_{out} < 64$) and a moderate number of prototypes per sub-space.
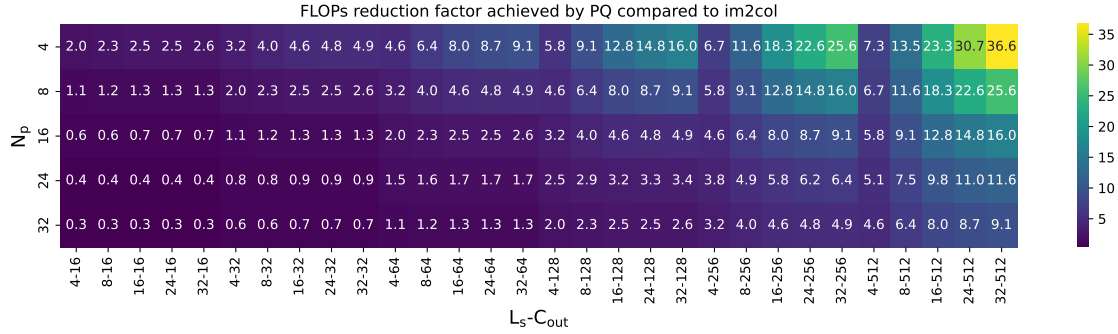

Fig. 12. Reduction factor in the number of FLOPs when a layer with C_{out} is implemented as a PQ layer with $\{N_p, L_s\}$ parameterization. The speedup grows faster with N_p than with L_s. For layers with very few output channels, there is no reduction in the number of FLOPs. Note this analysis assumes that looking-up the pre-computed dot product is cost-free.
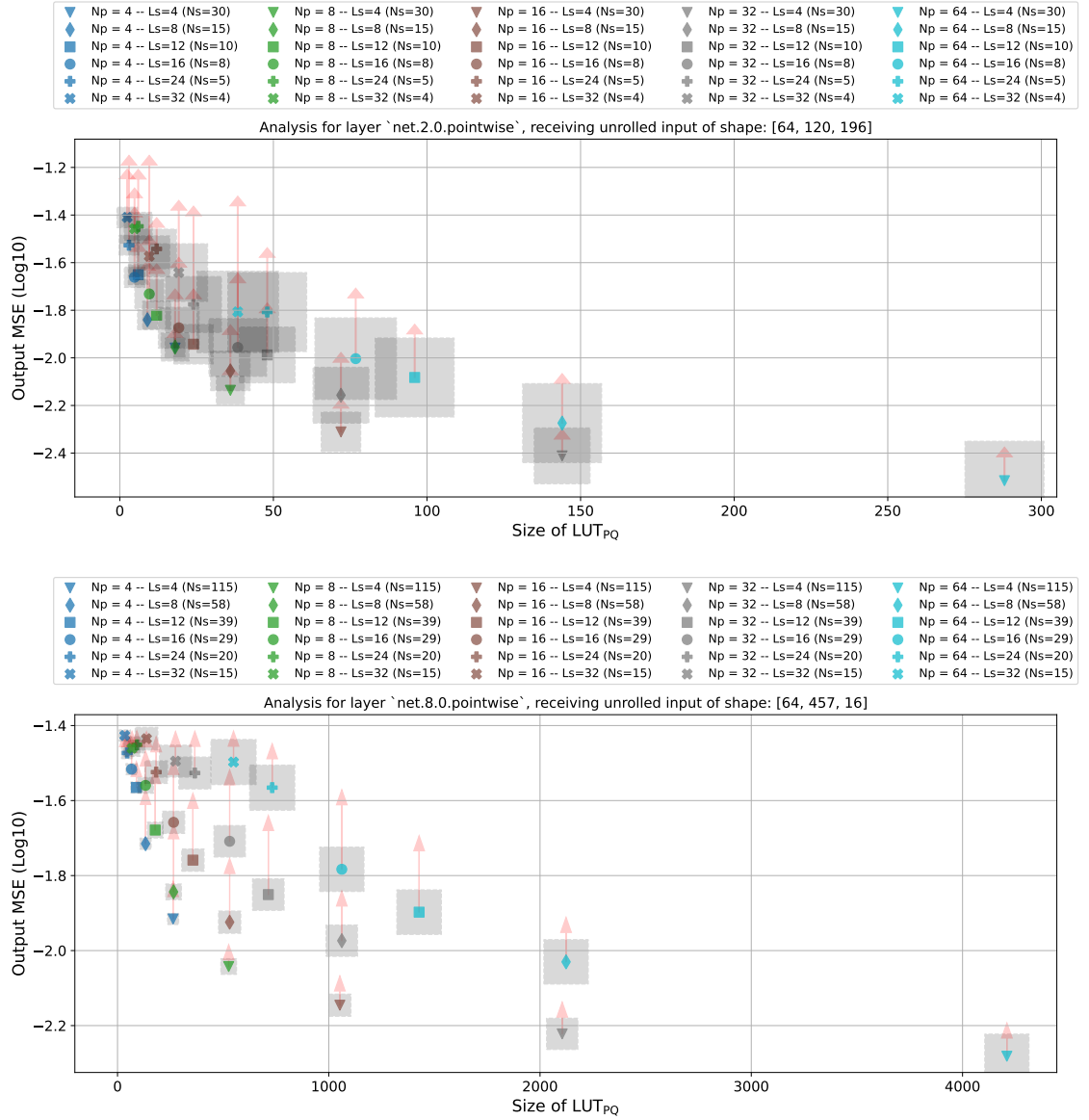
D   MANHATTAN DISTANCE

Figure 13 shows the impact of choosing Manhattan (L1) distance instead of Euclidean distance, which does not require multiplications. However, the impact on accuracy is not acceptable.

E   ACCURACY AND HARDWARE PERFORMANCE

To link our hardware efficiency study with PQ accuracy, Fig 14 plots the the mean square error (MSE) of the output vs. the number of PQA cycles for different PQ parameters for PointW-9 layer described in Table 4. It can be seen that increasing N_p increases the accuracy marginally while it has a clearer effect on the cycle count especially for larger L_s. It is worth noting that in this case, the efficiency of the hardware is limited by the external memory bandwidth so increasing N_p increases the size of the lookup table leading to more memory bandwidth requirements.
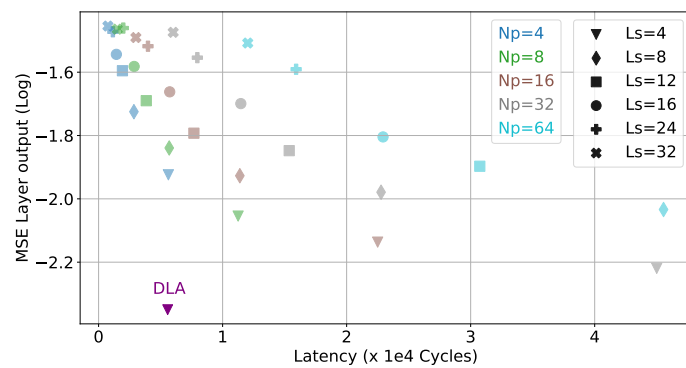
F   COMPARISON WITH OTHER ACCELERATORS

In the main text, a comparison with a conventional DLA [3] is presented. Here, comparison is extended to include another DLA [49] in Table 7. It's clear that PQA with the correct configuration outperforms both which indicates that PQ running on a customized hardware can indeed have a benefit over conventional convolution running on a customized hardware. It's worth mentioning that to be able to do the comparison with the information provided in the literature about the conventional accelerator [49], some assumptions were done. We're not taking into account the number of cycles taken by their engine to load data into the memory and we're assuming it is always compute bound but we don't make the same assumption in PQA (i.e. we count those cycles in PQA). One of the paper's optimizations is starting the calculation of the depthwise layer before the convolution layer before it is already done, they use the resultant outputs to kick off the computation in a pipeline manner. Given that there are no depthwise layers in ResNet20, we're not overlapping any of the layers calculation with the other layers.

Fig. 13. When replacing Euclidean distance with the more lightweight Manhattan distance, the error introduced to the output (represented as a red arrow) is not negligible with longer prototypes. The size of LUT$_{PQ}$ is not affected by the choice of distance metric.

Table 7. Comparison between the proposed accelerator and conventional accelerators from the literature.

Accelerator	Cycles per Image
PQA	11776
[3]	17664
[49]	19584

Fig. 14. MSE in a sample layer output vs number of cycles needed to process the input.