Machine Learning on FPGAs to Face the IoT Revolution

Xiaofan Zhang[1*], Anand Ramachandran[2*], Chuanhao Zhuge[1*],
Di He[2], Wei Zuo[2], Zuofu Cheng[1,2], Kyle Rupnow[2], and Deming Chen[1,2]
[1]University of Illinois at Urbana-Champaign
[2]Inspirit IoT, Inc.
{xiaofan3, zhuge2, zcheng1, dchen}@illinois.edu, {anand.ramachandran, di.he, wei.zuo, kyle.rupnow}@inspirit-iot.com

ABSTRACT

FPGAs have been rapidly adopted for acceleration of Deep Neural Networks (DNNs) with improved latency and energy efficiency compared to CPU and GPU-based implementations. High-level synthesis (HLS) is an effective design flow for DNNs due to improved productivity, debugging, and design space exploration ability. However, optimizing large neural networks under resource constraints for FPGAs is still a key challenge. In this paper, we present a series of effective design techniques for implementing DNNs on FPGAs with high performance and energy efficiency. These include the use of configurable DNN IPs, performance and resource modeling, resource allocation across DNN layers, and DNN reduction and re-training. We showcase several design solutions including Long-term Recurrent Convolution Network (LRCN) for video captioning, Inception module for FaceNet face recognition, as well as Long Short-Term Memory (LSTM) for sound recognition. These and other similar DNN solutions are ideal implementations to be deployed in vision or sound based IoT applications.

KEYWORDS

Machine Learning, FPGAs, Internet of Things

1 INTRODUCTION

In recent years, there has been rapid advancement in the quality of DNNs, and correspondingly rapid adoption of DNN-based applications including image classification [1], face recognition [2], video content analysis [3], and language translation [4]. These applications provide machine cognitive intelligence, but DNNs' overwhelming computation and memory consumption limit applicability in real deployments that need to balance energy consumption, resource overhead, and overall performance. FPGAs are promising candidates to accelerate DNNs due to improved latency and energy consumption compared to CPU and GPU-based implementations [5] [6]. FPGAs' high energy efficiency make them ideal for cognitive intelligence under strict energy limits such as in Internet of Things (IoT) devices. FPGA design is complex and time-consuming, but the advent of high level synthesis (HLS) has significantly reduced design and verification effort [7]. HLS-based design techniques not only improve the design productivity, but also improve the ability to implement and explore architecture optimizations including data quantization, parallelism extraction, pipelining, unrolling and memory partitioning [8].

Despite the design exploration advantages of HLS-based design flows, there are significant challenges in managing computational complexity, memory consumption and resource allocation. Furthermore, recent trends in state-of-the-art neural networks include irregular, parallel network structures such as the Inception module [9] or residual connections instead of simple feed-forward networks. These new topologies bring additional challenges for FPGA-based implementations because each layer features different computational and memory bandwidth demand, and a single complex Inception or Residual layer may contain multiple convolutions and activation functions; with limited FPGA resources, this affects exploration of layer implementation options, layer resource sharing and analysis to determine how to best allocate FPGA resources.

In this paper, we present a series of effective design techniques for mapping machine learning workloads to FPGAs. We first implement network pruning and quantization for reducing the size of given network with comparable accuracy so that we improve the resource utilization and feasibility of targeted FPGAs. We then model the relation between computational demand and latency. To determine the parameter options, we develop the Resource Allocation Management (REALM) to analyze resource allocation among the layers and give theoretical guidance to achieve minimized total network latency. After that, we apply a parameterized IP infrastructure that allows instantiation of well-optimized regular network layers (e.g. convolutional layers, fully-connected layers). This proposed IP also employs fast convolution algorithms to accommodate branched structures in the emerging neural networks. We then present several case studies with our design solutions using the proposed techniques above.

2 RELATED WORK

A generic IoT workflow is roughly constructed by object sensing, which offers information, both diverge in type and large in throughput, feeding smart system/device, in ideal case with ubiquitous processing capability [10], the latter triggers and offers a wide variety of services including decisions and actuations [11]. In most case the complex machine learning algorithms and models, such as DNNs, run on the back-end of the workflow, these systems are also physically separated from the front-end sensors. However, a clean cut between sensor and processing platform drains communication and computation resource [12]. This limitation asks for the distribution of intelligence [13], pushing machine learning tasks to the sensor or sensor nodes at the front-end of the workflow. In this case, FPGAs' high energy efficiency makes them prefect choices to enable cognitive intelligence in IoT devices. Meanwhile, large FPGAs can be deployed in the cloud for various types of DNN accelerations.

*These authors made equal contributions.

ICCAD'17, November 2017, Irvine, CA USA
2017. ACM ISBN 123-4567-24-567/08/06…$15
https://doi.org/10.475/123_4

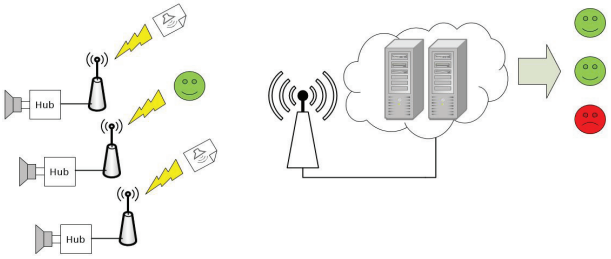


Figure 1: Distributing AED across the network

FPGA-based DNN implementations have demonstrated their fast total turn-around time compared to ASICs and low power consumption and fast speed compared to both CPUs and GPUs. The work presented in [14] explores the design space of loop optimizations in a CNN implementation to locate the best implementation point. In [15], an efficient design is presented for all layers of the CNN, and coupled with a weight quantization method. An overall implementation is built that delivers state-of-the-art results. An OpenCL-based design method and an associated design-space exploration for system-level throughput optimization is carried out in [16] to produce a CNN implementation. Memory access times of on-chip logic and off-chip memory are considered in [17] to achieve comprehensive optimization goals. In [18], a Binary Neural Network (BNN), a version of CNN with binarized weights, is implemented on FPGA using HLS [19] and introduces methods to optimize the layers in a BNN to meet throughput targets. The paper, [5], exhaustively studies loop optimizations and data movement patterns in CNN loops to obtain very high throughput and low latency. In [6], LSTM model compression and an associated accelerator design for LSTM are presented and the results surpass CPU and GPU solutions.

One of the most common organizations of IoT devices is an IoT sensing network [10], usually made up of widely distributed, sensors nodes, connected to complex machine learning models running in the cloud. Audio Event Detection [20] has shown promising capability in security surveillance networks and IoT in general. Many works [21, 22] in this area distributed their models and embedded cognitive pre-process capabilities into sensing node or front-end of the systems. They demonstrate promising potential to the IoT framework. However, to handle readings from large amount of sensors, pre-filtering, demonstrated by Figure. 1 from [23], is needed. The pre-filtering mechanism implemented into sensor nodes can effectively reduce false alarm while minimizing miss detection; most importantly, it reduces the required communication and computational load on the network. In terms of mapping machine learning algorithms into sensor nodes for pre-filtering and other applications, the limited computational resource and tight energy budget become obstacles. Therefore, it is important to reduce the DNNs so they can be fit into smaller FPGAs.

3 DNN DESIGN OPTIMIZATIONS

3.1 IP-based Design

In this section, we introduce a flexible HLS IP for designing Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN) for a range of IP parameterizations. We use instances of this IP to implement the LRCN [3] and Facenet [2].

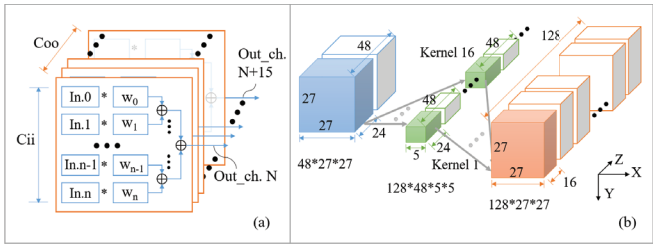


Figure 2: (a) The parameterizable HLS IP design and (b) computation in blue, green, and orange blocks carried out by a single IP

3.1.1 IP for regular DNN layers. The IP-based design methodology provides the opportunity to quickly implement a high-quality FPGA design. In order to leverage its benefits, the proposed HLS IP covers the most critical and universal operations (multiply-accumulations) in DNNs. With this parameterized IP, we are able to efficiently implement DNNs. In the critical loops representing the regular DNN layers, we moved the loop iterations with minimal dependency inwards, so that the inner loops in the transformed source code may be unrolled for maximum parallelization and resource utilization. We abstracted this loop structure as an HLS IP, and use it to construct the network.

As shown in Figure 2(a), the IP consists of Coo multiply-accumulate units of dimension C_{ii} each. In other words, it represents a two-dimensional, unrolled, loop tile of multiply-accumulate operations along one dimension, where C_{ii} and C_{oo} represent the unroll factors along the two dimensions of the tile. To illustrate this, Algorithm 1 shows the HLS code for a convolutional layer. By increasing C_{oo}, we can increase the number of operations executed by the HLS IP per second since there are no dependencies in the C_{oo} dimension. On the other hand, it doesn't seem like increasing C_{ii} would linearly improve the performance of the tile, because the following adder tree whose depth increases with increasing C_{ii}, thus worsening the latency. We would like the performance to increase linearly with C_{ii} as well. Hence we use Vivado's #pragma HLS PIPELINE to introduce pipeline stages into the adder tree. The depth of this pipeline is logarithmic with respect to C_{ii} due to its tree structure. The latency of using the adder tree is visible only when the tile is used the first time within the outer loop. Thereafter, the pipeline stages in the adder will be fully occupied until the completion of the layer computation. Figure 2(b) helps visualize how a complete convolutional layer is built using the IP. One blue block and sixteen green blocks are processed by the IP which generates partial sum of the orange block (one eighth of the layer's output). In Figure 2(b), C_{ii}=24 and C_{oo}=16, and this tile is reused 27×27×8×2 times to obtain all the outputs of the layer.

3.1.2 IP for Inception Module. One Inception module contains a composition of pooling and 1×1, 3×3, and 5×5 convolutions in parallel. The diversification of these irregular network structure creates challenges while using the HLS IP mentioned in section 3.1.1. So, we propose the IP working for the Inception module. This IP considers hardware-specific optimizations, as well as algorithmic improvements to accelerate convolution computation. Researchers [24] exploit the well-known convolution theorem, which states that convolutions in spatial domain are equivalent in frequency domain. Thus, we are able to mathematically reduce the computation

complexity with FFT-based convolution. More recently, Winograd minimal-filter based convolution algorithm has been introduced [25], and it is suitable for small kernel sizes and strides. In our work, we analyze and explore the properties of both FFT and Winograd-based convolution algorithms, and bring up a heuristic idea of designing a hybrid accelerator for different convolutions in Inception module.

3.1.3 Convolution in Frequency Domain. Based on the convolution theorem, spatial convolutions are equivalent to pair-wise multiplications in the frequency domain. Assume the convolution input is a $L \times M \times M$ feature map, and there are K kernels with size $L \times Q \times Q$. The spatial convolution's computation complexity is then $C_{spatial} = (KLQ^2M^2)$. During inference, weights usually only needs to be loaded once, therefore the FFT for kernels can be done offline. Thus, the computation complexity for FFT-based convolution consists of three parts: 2D FFT for feature map, pair-wise complex number multiplication, and inverse-FFT for the result. The overall computation complexity is given by:

$$C_{fft\_conv} = LM^2 logM^2 + KL4M^2 + KM^2 logM^2 \quad (1)$$

According to the above computation complexity analysis, larger kernel size leads to more significant speed up. We apply Cooley-Tukey Radix-2 algorithm as the core of our FFT convolution. The 2D FFT IP is implemented by using multiple parallel 1D FFT cores, as illustrated in Figure 3. With the 2D FFT IP we then implement the FFT-based CONV IP. There are three main points to exploit parallelism. Firstly, for each FFT to a single channel of the input feature map, we reuse the FFT result to compute all the output channels. Secondly, in the element-wise multiplication loop, we use HLS PIPELINE and UNROLL directives, together with manual unrolling of independent loops, to process multiplications in parallel. Thirdly, we greatly reduce the number of IFFT calls by executing IFFT after the summation of all the products. The result is still valid thanks to the linearity of the inverse FFT.

3.1.4 Winograd Minimal Filtering Algorithm for Convolution. Another fast convolution is based on the Winograd minimal filtering algorithm[25]. The algorithm can reduce the number of multiplications. Although this reduction comes with the expense of additional addition and constant multiplication, they can be implemented as bit-shift in hardware which are much cheaper. The 2D Winograd algorithm is implemented from nesting the minimal 1D algorithm. In general, a 2D Winograd algorithm $F(m \times m, r \times r)$ can

Algorithm 1 Pseudocode of Proposed IP

```
1: for C_i → InChannel , C_i+ = C_ii
2:   for C_o → OutChannel , C_o+ = C_oo
3:     for i → KernelHeight , i ++
4:       for j → KernelWidth , j ++
5: #pragma HLS dataflow // Loading from Ping-pong buf.
6:         Load_Data_Func( );
7:         for h → OutHeight , h ++
8:           for w → OutWidth , w ++
9: #pragma HLS pipeline // HLS IP starts below
10:            for coo → C_oo , coo ++ // Output traversal
11:              for SelBuf → 1, 2 // Sel. Ping-pong buf.
12:                for cii → C_ii , cii ++ // Input traversal
13:                  Out[SelBuf][C_o + coo][h][w]
                       += weight[SelBuf][coo][cii]
                       ×In[SelBuf][C_i + cii][h + i][w + j]
```

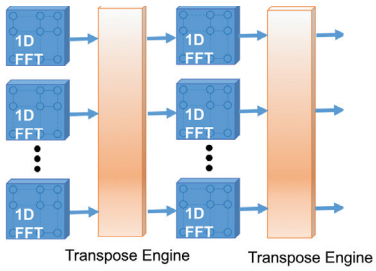


Figure 3: 2D FFT IP constructed from multiple 1D FFT cores

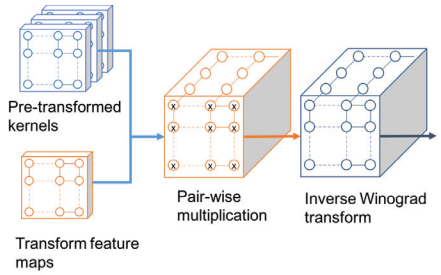


Figure 4: Winograd IP with parallelism and data reuse

be represented by the following equations. $U = GgG^T$, $V = B^TdB$, $Y = F(m \times m, r \times r) = A^T(U \odot V)A$, where G, B, and A are transform matrices, generated by Cook-Toom algorithm. $F(m \times m, r \times r)$ consumes 16 multiplications, at the expense of additional 84 operations. The Winograd IP implementation is illustrated in Figure 4, where we take advantage of data reuse opportunity and reduce number of transformations required.

3.1.5 Trade-off study for FFT and Winograd based convolution, and the Inception IP. Theoretical analysis and previous experiments [24, 25] have shown that these two algorithms are best suited for different convolution types. FFT-based method in theory provides greater speed up when kernel size is larger. This opinion is supported by Nicolas et al.'s implementation on GPU [24]. On the other hand, study claims that Winograd algorithm's improvement on speed wind down quickly when kernel size becomes larger because the number of additions and constant multiplications required by the transformation increases quadratically, offsetting the savings in the multiplications [25].

In order to find an ideal strategy of using different algorithms, we carry out design space explorations on FPGA, with the configurations shown in Table 1. One observation is that the kernel size does not affect FFT's performance in general because the zero-padding leads to the input data being the same size. The exploration result for kernel size 5×5 is shown in Figure 5. The baseline is implemented using a conventional loop-optimization method. In the figure, orange curve represents Winograd-based convolution's speed up against the baseline, and grey curve represents FFT-based method's speed up compared to the baseline. We learn that in small kernels, Winograd's algorithm dominates the performance. For larger kernel sizes and large input/output depth, FFT-based convolution starts to catch up in speed, and eventually outperforms Winograd's method. In the study, We try to keep the same parallel factor so that different algorithms use similar amount of resource.

We use Vivado HLS to implement a C++ template based reconfigurable Inception module IP, which includes all the techniques of

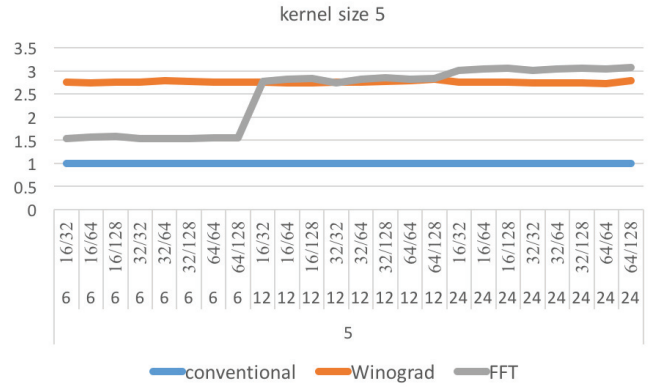
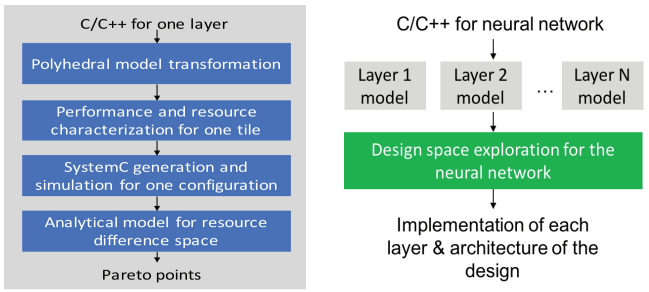


Figure 5: Speed up comparison when kernel size is 5×5



(a) Performance and resource modeling for each layer

(b) Design space exploration for neural network

Figure 6: Modeling and design space exploration

optimizations discussed above. The Inception module IP is implemented as a template function, with the *cccp* (which means the 1×1 CONV), *conv*3, *conv*5, and *pool* sub-functions that represent the parallel branches in the Inception module. Although there is no data read/write dependencies between each sub-function, by default, Vivado HLS won't schedule sub-functions to execute concurrently if sub-functions read from or write to the same array (even when they are writing to different location of the array). To solve this problem, we have to explicitly implement *split_input* and *combine_output* functions to copy the input feature maps to different buffers, and write to isolated buffers for different sub-functions, and then concatenate the result at the end. With the Inception module IP, we are able to instentiate Inception modules that fit different input/output and CONV sizes by passing template parameters, and build the face recognition FaceNet system.

3.2 Performance and resource modeling

The hardware structure of neural network layers is highly configurable, the best structure of each layer considering the resource and performance varies on different implementation platforms. To obtain a good structure configuration of each layer for overall performance and resource optimization on FPGAs involves exploring a huge design space and therefore extremely difficult. We cannot resort to synthesis and RTL simulation to solve this problem because it is too time consuming. Alternatively, in work [26] we developed a model based mechanism to accurately and effectively estimate the performance and resource of different structures of neural network layers (Figure 6a). Leveraging this model, we then developed a design space exploration framework [27] to decide the structure of the entire neural network (Figure 6b). We describe each step in detail in following paragraphs.

3.2.1 Software transformation using Polyhedral model framework. HLS provides a powerful automation from software to hardware. To generate efficient hardware, however, requires a good mapping of the input software to the hardware architecture. Optimizations such as data locality, pipelining and course-grained parallelism must be considered for improved performance. Automating these optimizations, however, are very challenging. One advanced compiler technique name Polyhedral model transformation has shown great promise in tackling this problem [28]. It targets a subset of loop structure named affine loops, where the data- and control-flow can be expressed used affine functions of the surrounding loop iterations and variable invariants in the program. Operating on this class, Polyhedral model carries out powerful transformations to enable automatic data locality and parallelism extraction via loop tiling, which results in significant improvement of performance. Neural networks are well suited for polyhedral model transformation thanks to their regular affine loop based structures. Therefore, in work [26], we used Polyhedral model to automatically detect affine regions and perform transformation to expose tiling and unrolling. One key observation of our modeling method is the static control flow property of affine programs which enables the behavior of the program to be essentially captured by iterating the same computation chunks (tiles), and this precise set of iterations can be exactly modeled at compile time.

Table 1: Design space explorations for FFT and Winograd based convolutions

Dimensions	Sizes evaluated
kernel sizes	3, 5, 7
feature map sizes	6, 12, 24
input/output dimensions	16, 32, 64, 128 (combinations)

3.2.2 Resource and performance characterization. Since the Polyhedral model transformation enables the whole design to be represented by regular tile structures, at this step, we focus exclusively on one tile, and construct a detailed performance and resource characterization. To obtain the associated resource and performance information, one tile is fed into the HLS tool to generate RTL code, and generate corresponding latency information and resource information. These results are used to compute the total application performance and resource. One thing worth noticing is that this step factors in the underlying implementation platform (e.g, the selection of FPGA).

3.2.3 SystemC generation and simulation for performance and resource computation. The previous step generates the information for one tile. To obtain the performance and resource usage of the whole design, we need to connect the information of one tile to the whole design, considering the configuration among different tiles. To avoid running HLS and RTL simulation for the whole design, we developed an automatic SystemC code generation framework to generate SystemC code to capture the structure of the design with a given configuration. Then a fast and accurate SystemC simulation is launched with the characterization data obtained in last step as

annotation, to compute the latency and resource information of the whole layer.

3.2.4 Analytical modeling to extrapolate the design space. So far, we can acquire the power and latency associated with a given architecture. However, the overall design space covers all combination of the configurations, and it is infeasible to exhaustively iterate the design space even using SystemC simulation. In work [26] we recognized relationships between code configuration and the resulting resource and latency can be roughly extrapolated by an analytical model, we therefore developed following method to obtain this model to describe the design space. First, for each loop being considered, we indicate the range of the valid iteration tile sizes, unroll factors and the maximum loop depth for which unrolling should be applied. Next, we sample the parameter dimension with a stride that increases exponentially along each dimension, to greatly condense the design space in logarithmically order. Third, for each of the sampled point, we generate the SystemC and run simulation to collect resource and latency information. Fourth, we use the sampled points with the resource and latency information to build the analytical model to extrapolate the design space. For each point in the space, we use the sampled points to perform a triangulation based linear interpolation to obtain the latency and area information. This model estimates the latency and resource for the each design choice within the design space, hence it outputs a set of Pareto points with the optimal trade-off between the two.

3.2.5 The design space exploration among different DNN layers. Step 1 to 4 above provide an effective solution for design space exploration of one layer. Next, we try to explore the design space of the whole neural network. Given the Pareto points of implementation choices for each layer, there is still a challengingly huge design space to explore to (1) decide the architecture among different layers, and (2) select the implementation of each layer given the constraints of an implementation platform, to achieve overall (near) optimal performance. In work [27], we developed an automated tool to solve this problem. The tool first builds a graph-based model to capture the behavior of the system, which factors in the computation of each layer and communication between layers. Then the tool samples N points from the curve of each layer and attaches them to the graph. Next, the tool formulates an integer linear programming (ILP) problem to select one of the sampled points for each layer for best performance under resource constraints. This step factors in the different parallelism schemes among different layers. According to the selected point of each layer, the tool zooms into the surrounding regions of that point on each curve to form a new (sub) curve, on which the sampling and ILP process is repeated. This framework shrinks the design space in a logarithmic manner and therefore is very efficient.The whole process terminates when only one point is left for each layer, which forms the final solution.

Experimental results demonstrate the effectiveness of the methods. We observed that the the latency and area error of the hardware model (Step 1 to 4) are 3.24% and 2.10% away from the actual Pareto points with 2091x faster design-space exploration time on average. We also observed our design space exploration framework (Step 5) is capable to consider complex designs.

$$latency = \alpha \sum_i \frac{C_i}{R_i} \quad (1) \qquad R_{total} = \sum_i R_i \quad (2)$$

$$\left[\sum_i \left(\sqrt{\frac{C_i}{R_i}}\right)^2\right]\left[\sum_i \left(\sqrt{R_i}\right)^2\right] \geq \left[\sum_i \sqrt{C_i}\right]^2 \quad (3) \text{ Cauchy inequality}$$

$$\sum_i \frac{C_i}{R_i} \geq \frac{\left[\sum_i \sqrt{C_i}\right]^2}{\sum_i R_i} \quad (4) \text{ Simplify (3)}$$

$$latency_{\min} = \alpha \frac{\left[\sum_i \sqrt{C_i}\right]^2}{\sum_i R_i} \quad (5) \text{ Use (1) and (4)}$$

$$\text{REALM:} \quad \frac{R_i}{R_j} = \frac{\sqrt{C_i}}{\sqrt{C_j}} \quad (6) \text{ Satisfies (4) with equality}$$

$$\text{notice:} \quad \left(a_1^2 + a_2^2 + \cdots a_n^2\right)\left(b_1^2 + b_2^2 + \cdots b_n^2\right) \geq \left(a_1 b_1 + a_2 b_2 + \cdots a_n b_n\right)^2$$

$$\text{when} \quad \frac{a_1}{b_1} = \frac{a_2}{b_2} = \cdots \frac{a_n}{b_n} \quad \text{satisfies with equality}$$

Figure 7: REALM equations

3.3 Efficient resource allocation

With proper configurations of the HLS IPs (mentioned in section 3.1), we can control the resource consumption and the performance gains on the FPGA implementation. But first, we need a guidelines for resource allocation per layer to balance the resource overhead and the overall performance.

Given that the computational demand of layer i is C_i and the resource consumed by that layer is R_i. An increase in R_i results in a proportional increase in the tile size of the HLS IP used in building the layer. Hence a certain quantity of resource allocated to the layer results in a proportional decrease in the latency of the layer. In addition, a layer that requires more computations, C_i, will involve more outer loop iterations involving the HLS IP. A larger C_i hence results in a higher latency. Combining these two, the latency of the layer is proportional to C_i/R_i. It is assumed that the designer has allocated a quantity, R_{total}, of resources to implement the complete network. Under these conditions, the equations in Figure 7 hold.

Equations (1) and (2) specify the latency and resource calculations (α is a constant of proportionality). Equations (3), (4) and (5) find a lower-bound for the latency of the overall network. Equation (6) lists the condition for the minimum latency. Equation (6) is hence the essence of REALM (Resource Allocation Management), which can be used to budget resources among different layers in the network. Once we obtain the ratio of resource allocation per layer from REALM, we can set the tile-sizes of the HLS IP appropriately to reflect this ratio and reach minimum latency. The condition for minimum latency is different from that of maximum throughput of a pipelined implementation of the DNNs. By a pipelined implementation, it will have maximum throughput when all the pipeline stages (layers) are perfectly load-balanced. In other words, this happens when $C_i/R_i = C_j/R_j$ or $R_i/R_j = C_i/C_j$, which doesn't always satisfy Equation (4) with equality (though, this is an altogether more intuitive result).

3.4 Network reduction

Neural Networks are trained on high-performance computing systems which have high throughput floating-point (FP) units, and off-chip memory (e.g. GPU clusters). The resultant inference models

naturally rely on similar system parameters for fast execution (referred to as "performance" in the sequel; network fidelity measures are referred to as "accuracy").

However, IoT devices using FPGAs do not offer the same luxuries even at the very high-end. There exist works on DNN compression [29] [6], fixed point approximation [30] and retraining [31] [32] that reduce the size of the network in terms of the size of the weights and activations with very little loss of accuracy. Such application-level engineering can potentially improve performance because the resultant networks play to the strengths of the underlying FPGA hardware. For example, Xilinx Ultrascale devices provide higher computational throughput when using 8-bit arithmetic [33], and reducing the size of the fully-connected layers can ease the demands on external memory bandwidth. Hence, these methods have opened-up new ways to deploy state-of-the-art DNNs on FPGAs more efficiently than before. At the same time, we can build further upon these techniques to provide additional tools to the hardware designer.

In [29] and [6], the parameters for network reduction are set based on experiments and empirical observations. In [34], only fixed-point approximations of convolutional networks are considered. In addition, the optimizations do not consider the underlying hardware behaviour. Taking cues from these approaches we have built a network approximation framework that proceeds through two main steps. In the first step, the network's weight footprint is reduced through pruning, clustering and fixed-point quantization. This leads to a smaller network which can potentially achieve higher performance on FPGAs. In the second step, if these modifications result in a more than tolerable loss of accuracy, the network is retrained under the constraints imposed in the first step to recapture the loss. This framework depends on a manual specification of parameters for network reduction. Specifications can be made for each layer independently resulting in a large parameter space, and both accuracy and performance of the network can vary in this space. Naturally, the question arises as to how this space may be explored efficiently, to provide a point of optimal performance under an accuracy constraint.

Table 2 (column "Independent") summarizes an effort to optimize VGG16 [35] for FPGA. Only three layers and 6 parameters are considered for tuning. When each parameter is individually set to the values shown under column "Independent", in each case, the network achieves a top-5 accuracy of over 73% for a small set of 100 examples. However, when all the parameters are applied simultaneously, the accuracy drops to unacceptable levels. Hence, not all parameters should be stressed to this extend. Determining the optimal solution that meets the 73% accuracy constraint will involve careful considerations of trade-offs between accuracy-loss and speed-improvement. To make parameter choices automatically, we believe, a framework must involve not only the DNN inference models, but also performance models of the approximated DNN which considers the characteristics of the underlying hardware platform.

Towards this goal, we are building a design-space explorer on top of our network reduction framework. The aim is to trade network accuracy for network performance, under constraints of tolerable accuracy loss, to achieve higher performance than the original network. The design-space will include a comprehensive set of parameters such as fixed-point quantization of weights and activations, and pruning and clustering of network weights, for each layer. The explorer will be able to analyze the accuracy and performance impact of each of these parameters. The explorer will be able to treat multiple types of network layers: multi-layer perceptron, CNNs - with inception units, residual units, Winograd filters and FFT - and LSTM.

The explorer uses "soft" empirical rules to aid network reduction (e.g., one such rule maybe that layer pruning and layer clustering can be independently applied across layers for a range of settings, without a large drop in accuracy; this rule is similar to observations in [29]), treating them as guideposts providing an initial direction for design-space exploration (DSE). A direct employment of such rules may not provide the best results, as demonstrated in Table 2. In the first phase of DSE, independent exploration is performed on each dimension of the design space. Compared to an exhaustive search which is exponential in the number of design-space dimensions, this initial phase is only of linear complexity. The information gathered in this phase is used, in concordance with the soft empirical rules mentioned above, to locate a design point that provides improved performance, but one with likely less than desired accuracy characteristics (e.g., top-5 accuracy). If, in the first phase, the network's accuracy is found to be less than ideal, then in the next phase, it will be further improved, by relaxing the approximation parameters. This means bit-widths and number of clusters can increase and the number of weights pruned away can decrease during this phase. Each such change may or may not result in a performance reduction of the network. While making these decisions, the explorer will consider the performance and accuracy impact of each such change, and make choices that result in no, or very little performance loss, and high accuracy gains. In the final phase, the framework will check whether any parameters can be further relaxed without affecting performance. This provides a further increase in accuracy, as well as a better chance of making further gains through retraining. Once this step is completed, a retraining run is launched, which can potentially recapture any accuracy tolerated during the DSE phase.

4 EXPERIMENTAL RESULTS

4.1 Results on network reduction

To illustrate the operation of our DSE we present some initial results on small experiments (using 100 examples) on VGG16 in Table 2. Under the column "Independent", six parameters are tested that individually provide the desired accuracy, however, when they are

Table 2: Network Reduction : experiments with VGG16

Feature	Independent		DSE	
	Value	top-5	Value	top-5
conv1_1 weight bit-width	4	≥ 0.73	4	≥ 0.73
conv1_1 number of clusters	8	≥ 0.73	off	≥ 0.73
conv1_2 weight bit-width	4	≥ 0.73	8	$\geq$0.73
conv1_2 number of clusters	4	≥ 0.73	4	≥ 0.73
conv2_1 weight bit-width	4	≥ 0.73	8	≥ 0.73
conv2_1 number of clusters	4	≥ 0.73	4	≥ 0.73
Simultaneous application		0.04		≥ 0.73

*the following values hold when not mentioned for all layers: 8-bit weights, 18-bit MM, 8-bit activations

applied simultaneously, the accuracy drops precipitously. Hence, some of the parameters should potentially have less stringent settings. If there is some loss of performance as a result, that loss should be minimal. We examine the same parameters as tuned through our DSE framework, as shown under the column "DSE". The simultaneous application of the DSE parameters provide the desired accuracy. Compared to the "Independent" setting, in the "DSE" setting, the first convolutional layer has clustering turned off, and the remaining layers have higher weight bit-width. We can qualitatively reason that the impact of these relaxations on performance is small. First, the cluster setting does not affect DSP throughput. Similarly, Xilinx Ultrascale FPGA (for which this case was targeted), provides the same DSP throughput for all bit-widths below 8. Thus the computational throughput of the design is not affected by the differences between the "Independent" and the "DSE" settings. Second, it is known that conv layers are compute bound [15]. In addition, turning off clustering for the first layer results in only one extra bit per weight, and the cluster size isn't changed for the other layers, meaning their weight volumes remain the same. It is fair to argue that the minor increase in the overall weight footprint (and associated extra memory operations), due to the differences between the "Independent" and "DSE" columns add very little delays to the network performance. Thus overall, we can expect a small, if any, impact on performance due to the DSE settings compared to the "Independent" column. At the same time, these settings provide much higher accuracy and potentially better retrainability. Future experiments will run on larger samples as our platform becomes more scalable.

4.2 Results on LRCN Implementation

To demonstrate our proposed design techniques, we map a LRCN (which is implemented using AlexNet [1] for CNN and LSTM [36] for RNN) into FPGA [37]. We apply the proposed REALM equations and obtain the ideal allocation of computation resources for layers in LRCN. We strive to allocate resource to different layers following the guidance of the REALM scheme. In this case, Xilinx Virtex-7 VC709 evaluation platform with XC7VX690T FPGA is used for our design. After synthesis with Vivado HLS 2016.2, placement and routing are completed subsequently showing resource consumption in Table 3, and the maximum frequency is 100MHz in our board level implementation. The performance and power comparisons are provided in Table 4. For the FPGA version, a power meter was used to measure the consumption of the entire evaluation board during the execution of the kernel. For the GPU version, power was measured using the command `nvidia-smi`, and for the CPU version, power was measured using a power meter. In summary, our implementation achieves 3.1X speedup compared to an NVIDIA K80 and 4.75X speedup compared to an Intel Xeon with 17.5X lower energy per image.

Table 3: Resource Consumption

BRAM	DSP	Flip-flop	LUT
1508	3130	321165	316250
51%	87%	37%	73%

Table 4: LRCN Performance Comparison

	Freq.	Latency	Speedup	Power	Efficiency
Our work	100MHz	0.040s	4.75X	23.6W	0.94J/pic.
NVidia K80	562MHz	0.124s	1.53X	133W	16.49J/pic.
Intel Xeon E5-2630	2.6GHz	0.19s	1.00X	88W	16.72J/pic.

4.3 Results on FaceNet Implementation

We used the proposed Inception IP with hybrid algorithms to build FaceNet, which is a face recognition system based on GoogLeNet-BN. The design was implemented on a Xilinx Ultrascale+ VU9P device. We implement Winograd $F(4\times4, 3\times3)$ in earlier modules and Winograd $F(2\times2, 3\times3)$ for later modules (where feature maps are small). We adopt FFT-based for 5×5 CONV in *inception 3b* and *inception 4a* because the 5×5 branch becomes critical path after 3×3 branch is well optimized. CONVs in *inception 3c* and *inception 4e* have stride 2, which the fast methods don't support, so we optimize them with conventional methods. The resource consumption and performance are shown in Table 5.

We compare our implementation on FPGA with GPU result. We use a cutting-edge Pascal-based NVidia GTX 1080 GPU, which has a 8.9 TFLOPS peak performance. The GPU implmentation is on Torch, with CUDA 8.0. The single image inference latency is shown in Table 6.

4.4 Results on sound recognition

We participate in the Hardware Design Contest at Design Automation Conference (DAC) 2017. The theme of the contest this year focuses on FPGA for IoT. We present the design of a robust, low cost, and low power audio system to quickly react to potential violent events in a public institution setting. Using an array of four microphones and a Lattice XO3L-6900 FPGA, our design, as presented in Fig 8, can detect, classify, and locate gunshot, screaming and affective speech. At the center of the implementation, we have a classifier conducting frame-wise event detection. A tiny Neural Network (NN), consists of 3 fully connected layers of 16 neurons, 4 neurons and 1 neuron, has demonstrated accuracy advantage over a Decision Tree classifier. The system can act as a detection pre-filtering front-end, similar to that of [23]; or it can function as a totally stand-alone device, without Ubiquitous cloud resource backing the system up. We simulate the full system performance on a subset of Google Audio Set [38], the Receiver Operating Characteristic (ROC) curve is presented in Figure 9. In this figure, we can see NN offers higher true-positive rate for the same false alarm rate across the full ROC range when compared to Decision tree.

Table 5: GoogLeNet-BN resource usage on Virtex VU9P

BRAM	DSP	FF	LUT
3067	2041	539422	938159
71%	32%	23%	79%

Table 6: GoogLeNet-BN latency comparison with GPU

	Our work	NVidia GTX 1080
latency	23.7ms	89ms

Table 7: IoT AED baseline resource usage on Lattice XO3

BRAM	DSP	PLL	LUT
7	0	2	4623
27%	None	100%	67%

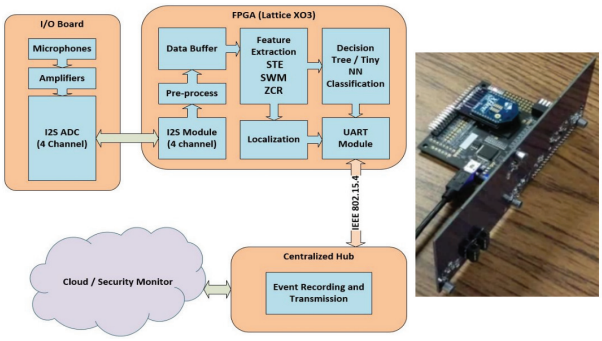


Figure 8: Block Diagram for IoT AED system.

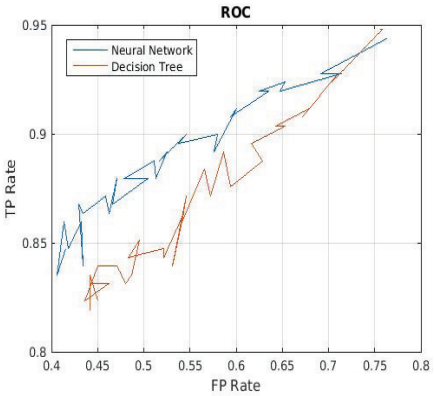


Figure 9: IoT AED Classification Accuracy Comparison.

Despite X03L-6900 having no DSP slices, we are able to implement the tiny NN, through HLS, with roughly 2000 LUTs. The resource utilization of the baseline system with Decision Tree is presented in Table 7, respectively. The design provided a small yet effective answer to the calling of embedding "smartness" into IoT sensor nodes, with which we were awarded first place in the contest.

5 CONCLUSIONS

In this paper, we presented a series of design techniques for FPGA-based implementation in order to meet the extremely high demand for energy efficiency and high performance. Among these technologies, we implemented methods for network reduction including network pruning, weight quantization, and network re-training in order to adapt the limited resources of IoT devices. We provided an effective tool for accurately estimating the performance and resource of different neural networks. It was able to capture the characters of target DNNs layer by layer based on given hardware platform. Also, we presented a resource allocation strategy which drove theoretical guidelines for per-layer resource allocation for minimum overall latency. To build the whole neural network, the configurable HLS IPs were designed for providing optimized implementations of the DNNs layers. Using these proposed techniques, we implemented the LRCN and the FaceNet on FPGA for video captioning and face recognition, as well as the LSTM for sound recognition and demonstrated very promising results.

REFERENCES

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*.
[2] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *CVPR*, 2015.
[3] Jeffrey Donahue et al. Long-term recurrent convolutional networks for visual recognition and description. In *CVPR*, 2015.
[4] Dzmitry Bahdanau and ohters. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.
[5] Yufei Ma et al. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *FPGA*, 2017.
[6] Song Han et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *FPGA*, 2017.
[7] Jason Cong et al. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Tran. on Computer-Aided Design of Integrated Circuits and Systems*, 2011.
[8] Xilinx. *UltraFast High-Level Productivity Design Methodology Guide.*
[9] Christian Szegedy et al. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, 2017.
[10] Jayavardhana Gubbi et al. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 2013.
[11] Rafiullah Khan et al. Future internet: the internet of things architecture, possible applications and key challenges. In *FIT*, pages 257–260. IEEE, 2012.
[12] M.A. Alsheikh et al. Machine learning in wireless sensor networks: Algorithms, strategies, and applications. *IEEE Commun. Surveys Tuts.*, 16(4):1996–2018, 2014.
[13] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.
[14] Chen Zhang et al. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *FPGA*, 2015.
[15] Jiantao Qiu et al. Going deeper with embedded fpga platform for convolutional neural network. In *FPGA*, 2016.
[16] Naveen Suda et al. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *FPGA*, 2016.
[17] Maurice Peemen et al. Memory-centric accelerator design for convolutional neural networks. In *ICCD*, 2013.
[18] Ritchie Zhao et al. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *FPGA*, 2017.
[19] Yaman Umuroglu et al. Finn: A framework for fast, scalable binarized neural network inference. In *FPGA*. ACM, 2017.
[20] Shawn Hershey et al. Cnn architectures for large-scale audio classification. *arXiv preprint arXiv:1609.09430*, 2016.
[21] Talal Ahmed, Momin Uppal, and Abubakr Muhammad. Improving efficiency and reliability of gunshot detection systems. In *ICASSP*, 2013.
[22] Giuseppe Valenzise et al. Scream and gunshot detection and localization for audio-surveillance systems. In *AVSS*, 2007.
[23] Di He et al. Using approximated auditory roughness as a pre-filtering feature for human screaming and affective speech aed.
[24] Nicolas Vasilache et al. Fast convolutional nets with fbfft: A gpu performance evaluation. *arXiv preprint arXiv:1412.7580*, 2014.
[25] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *CVPR*, 2016.
[26] Wei Zuo et al. A polyhedral-based systemc modeling and generation framework for effective low-power design space exploration. In *ICCAD*, 2015.
[27] Wei Zuo et al. Accurate high-level modeling and automated hardware/software co-design for effective soc design space exploration. In *DAC*, 2017.
[28] U. Bondhugula et al. Pluto: A practical and fully automatic polyhedral program optimization system. In *PLDI*, 2008.
[29] Song Han et al. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, 2015.
[30] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Low precision arithmetic for deep learning. *CoRR*, abs/1412.7024, 2014.
[31] Suyog Gupta et al. Deep learning with limited numerical precision. In *Proc. of the 32Nd Int. Conf. on Machine Learning*, ICML, 2015.
[32] Darryl Dexu Lin and Sachin S. Talathi. Overcoming challenges in fixed point training of deep convolutional networks. *CoRR*, abs/1607.02241, 2016.
[33] Yao Fu et al. Deep learning with int8 optimization on xilinx devices. *White Paper.*
[34] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented approximation of convolutional neural networks. *CoRR'16.*
[35] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR'14.*
[36] Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *ICML*, 2014.
[37] Xiaofan Zhang et al. High-performance video content recognition with long-term recurrent convolutional network for fpga. In *FPL*, 2017.
[38] Jort F Gemmeke et al. Audio set: An ontology and human-labeled dataset for audio events. In *IEEE ICASSP*, 2017.