High-Performance Video Content Recognition with Long-term Recurrent Convolutional Network for FPGA

Xiaofan Zhang[1], Xinheng Liu[1], Anand Ramachandran[1], Chuanhao Zhuge[1], Shibin Tang[2], Peng Ouyang[3], Zuofu Cheng[4], Kyle Rupnow[4], and Deming Chen[1,4]
[1]Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign
[2]Institute of Microelectronics, Tsinghua University
[3]School of Electronic and Information Engineering, Beihang University
[4]Inspirit IoT, Inc.

***Abstract*— FPGA is a promising candidate for the acceleration of Deep Neural Networks (DNN) with improved latency and energy consumption compared to CPU and GPU-based implementations. DNNs use sequences of layers of regular computation that are well suited for HLS-based design for FPGA. However, optimizing large neural networks under resource constraints is still a key challenge. HLS must manage on-chip computation, buffering resources, and off-chip memory accesses to minimize the total latency. In this paper, we present a design framework for DNNs that uses highly configurable IPs for neural network layers together with a new design space exploration engine for Resource Allocation Management (REALM). We also carry out efficient memory subsystem design and fixed-point weight re-training to further improve our FPGA solution. We demonstrate our design framework on the Long-term Recurrent Convolution Network for video inputs. Our implementation on a Xilinx VC709 board 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.**

I. INTRODUCTION

Recent years have seen rapid development of DNNs for deployment on FPGAs [1-7]. DNNs are composed of layers of regular computations such as convolution and pooling. High level synthesis (HLS) is well suited to optimize the regular computations of network layers. However, there are significant challenges in managing computational complexity, on-chip memory limitation, and external memory bottlenecks. Each layer in a DNN features different computational and memory bandwidth demand; effective design of a network demands both different optimization strategies based on layer type as well as different optimization parameters between different instances of the same layer. To produce optimal network implementations under resource constraints, we must determine best on-chip memory usage and external memory access patterns, explore layer implementation options and determine how to best allocate limited FPGA resources among the layers in order to minimize overall latency. In this paper, we develop the Resource Allocation Management (REALM) framework to analyze resource requirement and perform resource allocation among the layers in order to minimize total network latency. We demonstrate our framework using the Long-term Recurrent Convolutional Network (LRCN) [8]. LRCN is among the most complex tools available today aiming to achieve cognitive intelligence in the context of video/image analysis. To summarize, the main features of this paper are:

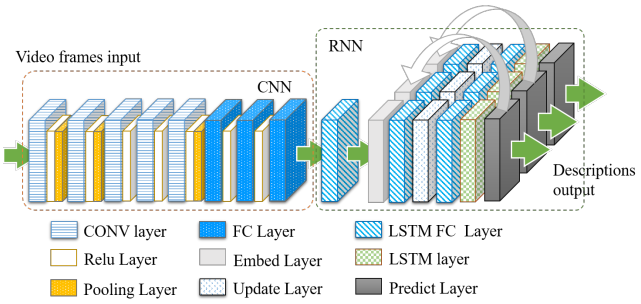

Fig. 1: LRCN structure

1) A flexible HLS IP for designing Recurrent Neural Network (RNN) and Convolutional Neural Network (CNN) optimally for a range of IP parameterizations. We use instances of this IP to implement the LRCN.

2) A resource partitioning solution that provides guidelines for resource allocation per layer for minimum overall latency. We call this solution, REALM.

3) An implementation of LRCN on the Xilinx VC709 board. We demonstrate the effectiveness of REALM and our HLS IP in the design process and achieve better performance than GPU and CPU solutions.

The rest of this paper is organized as follows. In Section 2, the LRCN is briefly introduced, and existing FPGA-based acceleration schemes for DNNs are discussed. In Section 3, the main challenges encountered in mapping the LRCN to the FPGA are presented. Section 4 describes our REALM framework, the HLS IP, and additional techniques for optimizing the design. The overall system implementation and comparison study are presented in Section 5. In section 6, we conclude this paper.

II. BACKGROUND

A. LRCN

A typical LRCN is implemented using AlexNet [9] for CNN and LSTM [10] for RNN. A LRCN uses 2.22 billion floating point operations and 86.56M synapse weights for processing just one video frame. Video frames are entered sequentially into the system and first processed by CNN (the

left side in Figure 1) for the extraction of visual features and the output is a vector with 1000 dimensions with each dimension representing a category of objects. This vector is passed to the RNN module (the right side in Figure 1) to generate proper descriptions (RNN produces a separate word in each iteration). By combining these different neural networks together, LRCN becomes an end-to-end model for video content description.

B. Related Work

FPGA-based acceleration has achieved very high performance for DNNs. The work presented in [1] explores the design space of loop optimizations in a CNN implementation. In [2], an efficient design is presented with a weight quantization method. An OpenCL-based design method and an associated design-space exploration for system-level throughput optimization is carried out in [3] to produce a CNN implementation. Memory access times are considered in [4] to achieve comprehensive optimization goals. The paper [5] exhaustively analyzes loop optimizations and data movement patterns in CNN loops. An FPGA accelerator for LSTM is designed in [6] which explores both computation and communication optimizations. In [7], LSTM model compression and an associated accelerator design are presented and the results surpass CPU and GPU solutions. Compared to these methods, we design a parameterized HLS IP for implementing neural networks and we introduce a resource allocation strategy called REALM, for achieving minimizing overall latency.

III. DESIGN CHALLENGES

The memory space and computational complexity of LRCN are very high. Table I summarizes the detailed requirement with CNN (AlexNet) and 15 iterations of RNN representing 15 output words. In total, 2.22 Giga (billion) floating-point operations are necessary during inference while 411970 inputs are distributed to different layers and 659290 outputs are generated. 86.56 million weight data are needed for video description which occupies 346.24MB of memory. Layers in LRCN show different characteristics regarding computation and memory requirements. The computational demand of convolutional layers, fully-connected layers and RNN are respectively 60.06%, 5.29% and 34.65% while the memory space requirements are 2.69%, 67.73% and 29.58% respectively. The complex structure and large

TABLE I: Space and Computation Complexity of LRCN

Layers	# of Weights(M)	# of Input data(K)	# of Input data(K)	# of Output data(K)
Conv1	0.03	0.21	150.53	290.40
Conv2	0.31	0.45	69.98	186.62
Conv3	0.89	0.30	43.26	64.90
Conv4	0.66	0.22	64.90	64.90
Conv5	0.44	0.15	64.90	43.26
FC1	37.76	0.08	9.22	4.10
FC2	16.79	0.03	4.10	4.10
FC3	4.10	0.01	4.10	1.00
RNN: 15 iterations	25.61	0.77	1.00	0.015
Total	**86.56**	**2.22**	**411.99**	**659.29**

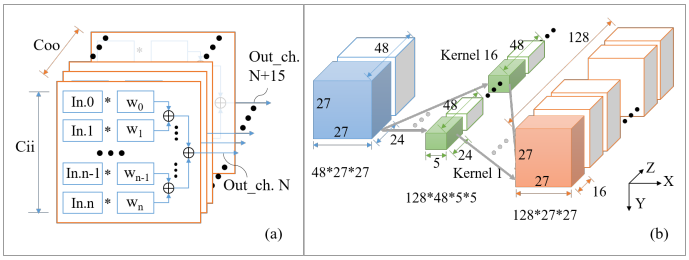

Fig. 2: (a) The parameterizable HLS IP design and (b) computation in blue, green, and orange blocks carried out by a single IP

variation in the computation and communication characteristics of different layers in LRCN present the following challenges.

1) Resource allocation and partitioning. Currently, HLS relies heavily on pragmas. Although using HLS pragmas can improve the performance of loops of DNN layers, it is not straightforward to relate such HLS pragmas directly to the performance because of the possible data dependencies. Second, LRCN consists of multiple loop structures across different layers, a homogeneous resource allocation for these layers will not produce the best results.

2) Memory limitation. The large size of the LRCN weight data forces us to use external memory to store these weights. The frequent access of external memory easily becomes a bottleneck, which means that the performance of a critical loop is not affected so much by its computation demand and resource allocation, but by how frequently it needs to access weights from the memory. While modeling the performance, it is much simpler if this memory bottleneck can be dealt with so as to reduce its impact on the system's performance. In this paper, we explore techniques in order to improve memory performance, simplify the overall performance model, and enable an effective resource allocation scheme for minimizing overall latency.

IV. DESIGN METHODOLOGY

A. IP for LRCN Design

The IP-based design methodology provides the opportunity to quickly implement a high-quality FPGA design with customized IPs. 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 can fit it into the LRCN implementation. In the critical loops representing the LRCN 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 2a, the IP consists of C_{oo} multiply-accumulate units of dimension C_{ii} each. It can represent a two-dimensional, unrolled, loop tile of multiply-accumulate operations. The proposed IP source code is shown in Algorithm 1. Figure 2b 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

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

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

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

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

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

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

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

Fig. 3: REALM equations

generates partial sum of the orange block (one eighth of the layer's output). In Figure 2b, 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.

B. Resource Allocation for Minimal Latency

Given that the computational demand of layer i is C_i and the resource consumed by that layer is R_i, the latency of the layer is proportional to C_i/R_i. R_{total} represents resources to implement the complete network. Under these conditions, the equations in Figure 3 hold (α is a constant of proportionality). Equations (1) and (2) specify the latency and resource calculations. 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.

Algorithm 1 Pseudocode of Proposed IP

```
1: for Ci → InChannel , Ci+ = Cii
2:   for Co → OutChannel , Co+ = Coo
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 → Coo , coo ++ // Output traversal
11:              for SelBuf → 1, 2 // Sel. Ping-pong buf.
12:                for cii → Cii , cii ++ // Input traversal
13:                  Out[SelBuf][Co + coo][h][w]
                       += weight[SelBuf][coo][cii]
                       ×In[SelBuf][Ci + cii][h + i][w + j]
```

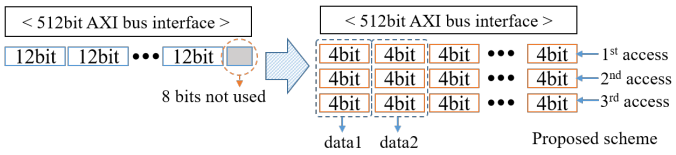

Fig. 4: Memory-Bus Organization for Minimizing Wastage

C. Network Pruning and Quantization

One of the conditions that can invalidate the assumptions behind REALM is that memory access time dominates the layer latency instead of computations. This can happen in the case of FC layers and LSTM which have a very low computation-to-communication ratio. Second, FPGAs perform favorably with fixed-point DSP units but not as well when using floating point.

To address these concerns, we prune the original LRCN network to reduce the number of output nodes in the fully-connected layers, FC1 and FC2, from 4096 to 256. In addition, two LSTM layers with 1000 hidden units each are converted to one LSTM layer with 256 hidden units. We then change the weights, bias and intermediate data to use fixed point numbers (shown in Table II) to improve DSP utilization and reduce the memory bandwidth pressure. We re-train the modified LRCN network using Caffe for maintaining accuracy. The accuracies of the networks are summarized in Table III. After pruning and quantization, the LRCN network occupies 11.08 million weight and requires 1.45G operations. These updated numbers are used for setting up REALM.

D. Memory Management

In addition to network pruning and quantization, we take further action to maintain the validity of the assumptions under REALM by reducing the impact of communication costs. Using a 12-bit format means that for a bus-width of 512 bits (this applies also to other bus-widths which are usually power-of-2), a few bits in each access may have to be discarded (512 is not divisible by 12). To prevent this, we collect bits from three bus accesses for regrouping into weights. The scheme is shown in Figure 4. To further improve the memory access efficiency, we need to ensure that the data access patterns exploit the maximum DDR memory

TABLE II: Layer-wise Bit-width Quantization

Layers	Output data (total bits, frac. bits)	Weight and Bias data (total bits, frac. bits)
Conv1	16, 4	12, 11
Conv2	16, 7	12, 11
Conv3	16, 8	12, 11
Conv4	16, 9	12, 11
Conv5	16, 10	12, 11
FC1∼FC3	16, 11	12, 11
LSTM	16, 11	12, 11

TABLE III: Accuracy after Re-training

Network	Accuracy
LRCN - original (AlexNet + 2 LSTM layers)	43.0%
LRCN pruned (AlexNet + 1 LSTM layer)	41.8%
LRCN pruned, fixed-point (AlexNet + 1 LSTM layer) *implemented on FPGA*	42.0%

Fig. 5: Left: Front-end (Tegra TK1 and webcam); Right: Back-end (Xilinx VC709 FPGA board, host PC)

bandwidth. We re-order the multi-dimensional weight data into a linear sequence that follows the order of computation. This ensures that data locality is exploited when the HLS IP instance accesses weight data thus improve the throughput of access. To further reduce the memory access latency, we instantiate FIFOs outside the LRCN layers in the path connecting the layers to external memory. In addition, we use Vivado HLS to synthesize ping-pong buffers at the input of each layer. This is intended to hide memory latency between the layer's computational unit and the FIFO feeding the layer.

V. IMPLEMENTATION AND COMPARISON

A. Overall System Setup

Xilinx Virtex-7 VC709 evaluation platform with XC7VX690T FPGA is used for our design with optimizations mentioned in Section 4, and Vivado HLS 2016.2 is used for high-level synthesis.

We build an end-to-end, real-time, video content description system that can directly process frames from a commercial webcam. We use Tegra TK1 and a Logitech C920 full-HD webcam as the front-end. We down-sample the captured frames to the size that fits the LRCN network and stream the image frame over the internet. On the back-end side, the host PC receives the frames and pre-process the image, including re-ordering of pixels and fixed-point conversion, before off-loading the data to our LRCN kernel implemented on the FPGA. After computation, the kernel sends back an index vector which is used for dictionary look-up to produce a sentence. The complete system is shown in Figure 5.

B. Comparison

Resource consumption of proposed LRCN is shown in Table IV, and the maximum frequency is 100MHz in our board level implementation. We compared the performance of the pruned LRCN model on CPU, GPU, and FPGA. The floating-point version of the pruned LRCN was run on GPU and CPU and the fixed-point version of the pruned LRCN was run on the FPGA. In addition, two GPUs in the NVIDIA K80 were used to map the LRCN network (using cuDNN) under Caffe framework. The CPU version is an optimized implementation (using BLAS) from the Caffe framework.

The performance and power comparisons are provided in Table V. 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.

TABLE IV: Resource Consumption

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

TABLE V: LRCN Performance Comparison

	Freq.	Latency	Speedup	Power	Efficiency
This 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.

VI. CONCLUSIONS

In this paper, we presented an implementation of LRCN, using an HLS-based design flow for FPGA. We introduced a resource allocation strategy called REALM, which drove theoretical guidelines for per-layer resource allocation for minimum overall latency. We implemented methods including network pruning, weight quantization, and retraining, as well as efficient memory system design. Using our resource allocation guidelines, we tuned the parameters of the proposed HLS IP instances to implement the LRCN to obtain a design whose power and latency performance surpassed those of GPU and CPU implementations.

ACKNOWLEDGMENT

This work is partly supported by the IBM-Illinois Center for Cognitive Computing Systems Research (C^3SR) and IBM Faculty Award.

REFERENCES

[1] Chen Zhang, et al. Optimizing FPGA-based accelerator design for deep convolutional neural networks. Proc. of the Intl. Symposium on Field-Programmable Gate Arrays. ACM, 2015.

[2] J. Qiu, et al. Going deeper with embedded FPGA platform for convolutional neural network. Proc. of the Intl. Symposium on Field-Programmable Gate Arrays. ACM, 2016.

[3] N. Suda, et al. Throughput-optimized OpenCL-based FPGA accelerator for large-Scale convolutional neural networks. Proc. of the Intl. Symposium on Field-Programmable Gate Arrays. ACM, 2016.

[4] M. Peemen, et al. Memory-centric accelerator design for convolutional neural networks. Intl. Conference on Computer Design. IEEE, 2013.

[5] Y. Ma, Y. Cao, S. Vrudhula, J. Seo, Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. Proc. of the Intl. Symposium on Field-Programmable Gate Arrays. ACM, 2017.

[6] Yijin Guan, et al. FPGA-based Accelerator for Long Short-Term Memory Recurrent Neural Networks. The Asia and South Pacific Design Automation Conference, IEEE, 2017.

[7] S. Han, et al. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. Proc. of the Intl. Symposium on Field-Programmable Gate Arrays. ACM, 2017.

[8] J. Donahue, et.al. Long-term recurrent convolutional networks for visual recognition and description. Proc. of the conference on computer vision and pattern recognition, IEEE, 2015.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, Imagenet classification with deep convolutional neural networks. Advances in Neural Information Processing Systems, 2012.

[10] A. Graves and N. Jaitly, Towards End-To-End Speech Recognition with Recurrent Neural Networks. Proc. of the Intl. Conference on Machine Learning, 2014.

[11] E. Del Sozzo, et al. On the Automation of High Level Synthesis of Convolutional Neural Networks. The Intl. Parallel and Distributed Processing Symposium Workshops, IEEE, 2016.