Optimization of Convolutional Neural Networks on Resource Constrained Devices

Arish S
Nanyang Technological University
50 Nanyang Ave, Singapore,
sarish@ntu.edu.sg

Sharad Sinha
Indian Institute of Technology Goa
Farmagudi, Ponda, India
sharad@iitgoa.ac.in

Smitha K G
Nanyang Technological University
50 Nanyang Ave, Singapore
smitha@ntu.edu.sg

***Abstract*—Implementation of convolutional neural networks (CNNs) on resource-constrained devices like FPGA (example: Zynq) etc. is important for intelligence in edge computing. This paper presents and discusses different hardware optimization methods that were employed to design a CNN model that is amenable to such devices, in general. Optimization techniques such as adaptive processing, inter and intra-layer parallelism etc. are employed to show the superior performance of proposed methods over the state-of-the-art.**

***Index Terms*—FPGA, convolutional neural networks, hardware optimization, resource-constrained devices, automated code generation**

I. Introduction

Convolutional Neural Networks (CNN) have wide applications in almost all fields and getting enormously popular. Recent innovations in CNNs give rise to numerous intelligent applications in embedded vision, IoT, mobile systems, etc. Despite this popularity, the implementation of CNNs into hardware, especially resource-constrained devices, is still a cumbersome task. The resources in the embedded hardware platforms are very much limited to accommodate the complexity of CNNs having millions of parameters and billions of operations. Even MobileNets [2] and SqueezeNet [7], which are designed for mobile devices, are having a parameter count of 4.2M and 1.2M and the number of MAC operations of 569M and 352M respectively. The high complexity of CNNs makes Graphics Processing Units (GPUs) the obvious choice for hardware because of their high computation capability and the availability of many supported frameworks. Though, power consumption and the inability to work as a standalone unit other than a hardware accelerator pulls the GPUs back when it comes to embedded systems. The rapid improvement in FPGA technology paved the way to choose FPGAs as the preferred platform presently to implement machine learning algorithms. Along with its flexibility in design, ability to handle custom low-precision datatypes, and the ability to work as a standalone system are more advantageous for FPGAs. Still, resource constraints and the unavailability of a suitable hardware framework for CNNs are adversely affecting the implementation of CNNs on FPGAs. Hence it becomes very much important to optimize the network for inferencing on hardware.

In this work, a hardware-optimized design is proposed, targeting resource-constrained devices. Different optimization techniques in terms of data quantization, adaptive data processing, efficient use of on-chip BRAMs, and intra-layer and inter-layer pipelining along with limited channel level parallelism are employed to generate the hardware code to suit resource-constrained devices without affecting the performance adversely. An optimized Verilog library is created according to the proposed design. The hardware code for the CNN model is generated in Verilog with the help of python based generator scripts using the verilog library. The python-based script generates and trains the CNN with the help of Keras framework with Tensorflow backend and pre-processes the trained parameters and generates the Verilog code for hardware inference. This optimized hardware design produces better performance with respect to resource utilization compared to other related works.

II. Background

Increasing researches in the area of machine learning and deep neural networks provide much better neural network models with near-human accuracy and performance. Starting with LeNet [24], which had only less than 60K parameters, many improved and accurate models with increasing complexity were put forward. With the increase in complexity, the implementation on hardware becomes equally complex. Also, hardware-oriented approaches like MobileNets [2], SqueezeNet [7], Ristretto [17], etc. provided much compressed CNN models which suits hardware. Though, most of the hardware inference implementations like [3], [4], [14], [18] are focused on throughput and latency and not really targeted for resource-constrained devices. Work in [28] gives a very good implementation in terms of throughput/slices but usage of more number of DSP units and BRAM is a problem. Work in [19] reduces the usage of DSPs and BRAM but, lags behind in performance. Hardware frameworks such as [5], [8], [11], [22], [25], [26] also produces some of the best design in terms of throughput and latency, but not in terms of resources.

In order to reduce the hardware implementation complexity, it is important to do optimization in the algorithmic level also. In CNNs, the convolution layers of feature extraction layers constitute the least number of parameters whereas dense layers or classifier layers generate large parameter count and in terms of computational complexity, convolution layers constitute the most number of multiplications and additions. The size of the convolution kernel hence makes a huge difference in complexity. AlexNet [1] uses kernel sizes of 11x11 and 5x5, which creates a lot more multiply and accumulate operations

(MAC) compared to the same model with a filter size of 3x3. ResNet [12], Inception network [15] and GoogleNet [6] proved the same by introducing 3x3 and 1x1 filters, which reduced the complexity and parameter count significantly. Hence in this work, the size of the convolutional layer kernel size is chosen so that it will reduce the complexity of the network. The All Convolutional Net [21] eliminates all the dense layers and uses only convolutional layers for classification. This in fact can reduce the number of parameters. Hence, reducing the convolution kernel size and the number of dense layers can reduce the parameter count, which is very much advantageous for implementation on hardware.

Techniques like data quantization are very much effective in reducing the complexity at both algorithmic and hardware levels. Works in [9], [10], [18], [20] uses fixed-point numbers effectively to reduce the complexity of the network. Still, the disadvantages like underflow in fixed-point format demand to increase the number of bits for representing a wide range of values. Pipelining and parallelism are also important techniques to improve the performance of the implementation on hardware [10], [16]. Even though parallelism is very common, the advantages of pipelining are not fully explored yet. Work in [16] provides a very detailed explanation of intra and inter-kernel parallelism. But, complete parallelism actually increases resource utilization very much [10], [18]. Hence, it can be concluded that limited parallelism along with complete pipelining is the best solution for increasing the performance in resource-constrained devices. In this work, we are proposing optimized hardware models using sequential and pipelined implementation strategies along with limited channel-level parallelism to efficiently implement CNNs on resource-constrained devices. The sequential model utilizes pipelining within the layer (intra-layer pipelining) and the pipelined model utilizes both intra-layer and inter-layer pipelining. Both models use limited-level parallelism to increase the execution speed. The hardware optimization strategies used in this work are described in the following sections.

III. Proposed CNN model for hardware

Choosing a suitable hardware architecture is very much important to implement CNNs on hardware. In this work, the computational complexity and the size of the parameters are taken as two key factors while designing for resource-constrained devices. Hence, we use a customized model with a constant convolution kernel size of 3x3 and used only one dense layer as the classifier. This helps to reduce the computational complexity in convolutional layers and to reduce the parameter count in dense layers. The hardware architecture of the evaluation model is explained in detail in section 5. The optimization methods employed are explained in detail in this section.

A. Hardware optimization

Hardware optimization is one of the key factors while inferencing a CNN on to hardware. In this work, we have employed various optimization techniques for resource-constrained devices as explained in the following sections.

1) **Data Quantization:** Data quantization is employed to reduce the computation overhead. In order to improve the performance on hardware, either fixed-point format or binary integer format are used instead of floating-point numbers. Most of the research employs fixed-point format [3], [18], [28] to reduce the bit-width of the operands, but in this design, we employ signed binary integer format to perform all the computations. Even though fixed point format can provide better accuracy, the number of bits required to represent a wide range of numbers would be more and also would require pre or post-processing after multiplication similar to floating point operations. Another disadvantage with fixed-point format is underflow which can lead the result to be zero if the operands are very small fractional values. This would require scaling up before multiplication and this can in fact cause the other operands in the dataset to go out of range if the bit width is not large enough. In the signed binary integer format, the bit-width of the input data and trained parameters are scaled down to 8-bit by using a proper scaling factor and hence the size of the multiplier at the first convolution stage would be 8x8 only. Since the first convolution layer is the most computation-intensive layer compared to other convolution layers, keeping the bit-width of operands to 8-bit could save a lot of resources and time. The maximum size of the multiplication operation would be 16x8 at any stage of the model after the first convolution layer. The output of all the 16x8 multiplication operations of other layers are set to 16-bit by scaling down by 4-bit and then truncating the scaled down 32-bit result to 16-bit.

2) **Scaling:** Since this work uses signed binary integers, scaling is important to keep the numbers within the range. Unlike fixed-point format, scaling doesn't cause underflow or overflow but keeps the numbers within the range. Scaling is performed in every layer and is employed after every convolution operation in the first convolution layer, and after every multiply-accumulate (MAC) operation for all other layers. The result of first layer convolution operation is 16-bit and is scaled down by 16 (4-bit right shift) after multiplication and addition of bias. In all other convolution layers, the 16x8 multiplication gives a 32-bit result and the result is scaled down by 16 (4-bit right shift) and truncated to 16-bit. While performing truncation, if the 32-bit value is greater than 32767, the value is set to 32767 and if the value is less than -32768, the value is set to -32768. In other words, the value is truncated up or down to the maximum possible value a 16-bit number can represent if the values exceed the range. If no overflow occurs in the 32-bit scaled down value, the upper 16-bits are truncated and only the lower 16-bits are taken as output. Similarly, the bias in each of the consecutive convolution layers are scaled up by 16 (4-bit left shift) and changed to 16-bit from 8-bit. The scaling is performed to make the ratio of multiplication output and bias the same after the integer multiplication because integer multiplication doubles the range of numbers compared to normalized floating-point number multiplications.

After adding bias with the 16-bit output from the multiplier, the 18-bit sum is again truncated as done previously during multiplication. For the final dense layer, the scaling down and truncation is not performed and the output is kept as 32-bit to preserve the accuracy of the classifier. But, if there are multiple dense layers in the design, scaling and truncation are performed in all other dense layers except the final layer. The bias is scaled up by 64 (6-bit left shift) and changed to 16-bit from 8-bit to preserve the ratio of bias and the 16-bit result after multiplication.

*3) **Adaptive data processing:*** Most of the prior work [2, 4, 8, 14, and 15] concentrates on a single repeating processing element (PE) for all core operations like convolution and pooling. This can sometimes be inefficient. The convolution operation for multiple input channels may require more resources than a single-channel convolution operation as the sequence of MAC operations vary and for each output channel, a multi-input adder is required to add the convolved input channels. Using the same multi-input channel convolution module for single-channel input operation might cause a waste of resources and power for a convolution layer having multiple output channels if the channel calculations are performed in parallel. Also, the size of the multiplier cannot be varied if we are planning to use the same PE for all the layers. Adaptive selection of processing modules would save a lot of resources with multiple channels executing in parallel, especially in the first convolution layer, which is the most computation-intensive convolution layer. Another important thing to be mentioned is that in this proposed design, the ReLU operation is performed within the convolution layer itself, which would save the temporary storage buffers and data load time while using a separate ReLU layer.

Similarly, the size of the max-pool kernel is always fixed to 2x2 in almost all the state-of-the-art designs [1]–[3], [19], [27], [28]. One disadvantage of this design is that if the input dimension is odd, the last column of the input feature map will be ignored by the kernel window. This can cause loss of information and accuracy, especially at the deeper layers where the feature maps have fewer dimensions. Hence, in order to avoid data loss during max-pooling, dynamic selection of 2 different kernel sizes is employed based on the input dimension. If the input dimension is even, the pooling kernel size is selected as 2x2 and if the input dimension is odd, kernel size is selected as 3x3. Also, this work uses different dense layer modules for pipelined and sequential models and is explained in detail in the following sections.

Convolution Layer: For single channel input, the convolution kernel - where the convolution operation (multiply-accumulate) is performed- loads the input data directly from the BRAM each time the convolution kernel is called. The advantage of direct load is that the input registers which are required to store the preloaded data can be avoided. The biases and weights are loaded only once as they remain constant, which avoids unnecessary delay. Figure 1 shows the convolution module for a single channel input. The scaling and truncation of the output and ReLU operation is also performed

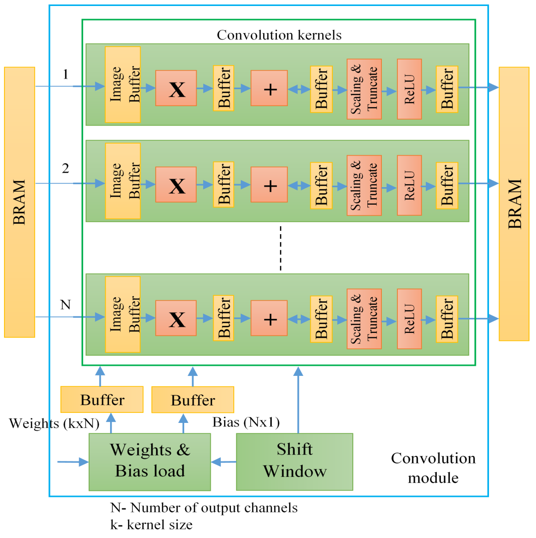

Fig. 1. Convolution module for single channel input for all output channels

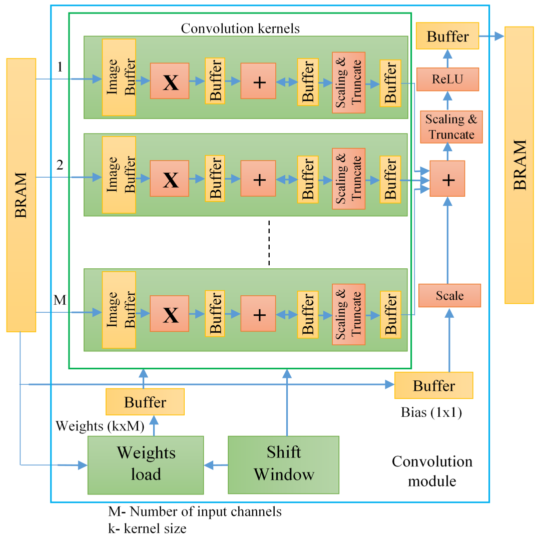

Fig. 2. Convolution module for multi-channel input for a single output channel

in the convolution kernel itself which will help to minimize the temporary storage registers. The result is directly stored in the BRAM which can be accessed by the subsequent max-pool layer.

For multi-channel input, the addition of bias is not performed in the convolution kernel, as it is required to have the convolution results of all the input channels before accumulation and bias addition. Hence the multi-channel convolution operation uses multiple convolution modules equal to the number of output channels. Each of these modules calculates the output for the corresponding output channel. This module uses a multi-input adder having inputs equal to the number of input channels plus the bias. During pipelined execution, these modules are used in parallel and hence these adders will consume unnecessary resources if used for single-channel input. The weights are preloaded for each channel directly

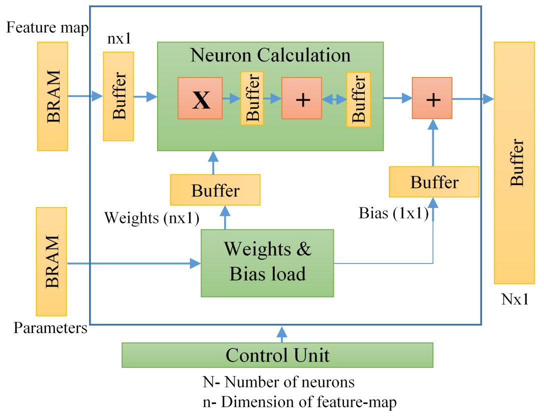

Fig. 3. Dense layer calculation module for sequential model

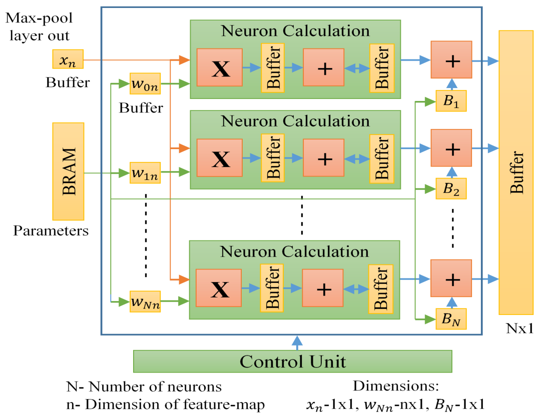

Fig. 4. Dense layer calculation module for pipelined model

from BRAM only at once in the same way as in single-channel convolution, whereas the bias for all output channels is preloaded before the shift window operation. The convolution operation for each input channel is performed at once and then the elements are added together along with the bias. Unlike single-channel convolution, scaling and truncation are performed twice after each addition operation. The output after the ReLU operation is recorded to BRAM. The block diagram of multi-channel convolution module for a single output channel is shown in figure 2.

Max-pooling layer: Max pooling operation also uses adaptive processing where the kernel size is determined by the dimension of the input to ensure best possible accuracy. The kernel size is selected to be either 2x2 or 3x3 based on the input dimensions and has a constant stride-width of 2. If the input dimension is even, the kernel size is taken as 2x2 whereas it is 3x3 when the input dimension is odd.

Dense layer: Adaptive selection of dense layer is used based on whether the hardware optimization technique employed is either sequential or pipelined version of the CNN hardware model. In sequential execution, the complete data to the dense layer is already available and hence the computation module takes the flattened input data and computes each neuron's operation one after another. In pipelined execution, as soon as the first output element of preceding max-pool layer becomes available, the dense layer starts computation. Hence, the dense layer computation module for pipelined versions takes a single element as input and computes all the neuron operations at once and the result of each neuron is accumulated and stored in a temporary buffer until next input is available. These neuron operations can be performed either in parallel or sequentially by re-using the multiplier depending on the user requirements. Figures 3 and 4 show the different dense layer computation modules for pipelined and non-pipelined versions.

In the non-pipelined version, the neuron calculation (NC) module has an input size equal to nx1 where n is the feature-map dimension, whereas in pipelined version, the input size of NC module is 1x1, i.e. a single element. Similarly, the input weights and bias size is nx1 and 1x1 respectively for the non-pipelined and pipelined version. The non-pipelined NC module will have n multiplications and n-1 additions whereas the pipelined NC module will have only one multiplication and one addition in each cycle. The NC module of the non-pipelined version will be executed either sequentially or in parallel depending on the hardware platform and resource availability. It can use either multiple NC modules in parallel or multiple multipliers inside the NC module to speed up the execution. Similarly, in the pipelined model, the parallel NC modules can also be executed sequentially by reusing a single NC module. These configurations will depend on the performance requirements.

4) **Use of on-chip BRAM as storage registers:** In convolutional neural networks, the storage of intermediate results between different layers would require a considerable amount of memory bandwidth as the size of the output feature-maps of layers is much larger. Using distributed memory for storing intermediate results would consume a lot of FPGA resources. In most of the works, on-chip BRAM is used to store only the parameters [3], [28], because of the limited memory available and is not often used to store intermediate results because of the delay associated with memory access also. Some of the works use off-chip RAM to store the intermediate results or parameters [10], [14], [18]. The delay associated with the storage and retrieval from external RAM will be more compared to retrieval from on-chip memory. In this work, the storage and retrieval time is negligibly reduced by pipelining the operations within the layers and hence, no additional clock cycles would be required for memory write and read operations. Intra-layer pipelining is employed in all layers and is explained in detail later. Though both convolution and max-pooling layers would cause retrieval delay from RAM during sequential execution, the dense layer operations can be pipelined in sequential execution also to produce zero retrieval delay as the dense layer module can process one operand at a time unlike convolution or pooling modules which require an array of elements in each operation. So, as soon as the results are written to the RAM, the dense layer can start processing at the same time and hence the time taken to retrieve the data

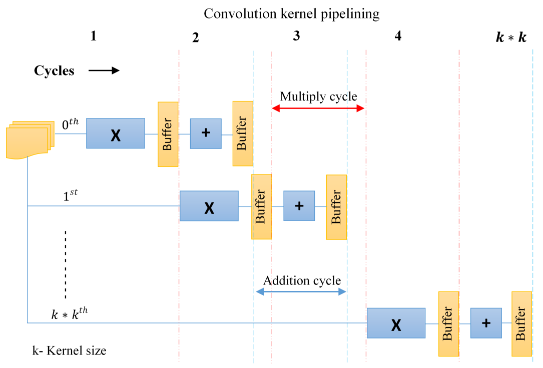

Fig. 5. Pipelining inside the convolution kernel

can be merged with the execution time of the previous layer.

5) **Zero skipping:** The convolution and dense layer modules skips the multiplication operation and assert the result as zero if any of the operands is zero. Most of the values of the input feature map of the image are actually zeros and hence, skipping multiplications when operands are zero saves a significant amount of time and energy during inferencing.

6) **Pipelining of operations:**

a. Intra-layer pipelining Pipelining is employed in all layers to limit resource utilization, reduce latency, and increase throughput. One of the most important advantages of intra-layer pipelining is that it can save the time delay associated with the memory write and read operations to RAM in each layer.

The pipelining method in convolution layers is shown in figures 5 and 6. The convolution module employs multiple pipelining strategies within the convolution module and within the convolution kernel. Within the convolution kernel, the multipliers and adders are pipelined in such a way that as soon as the multiplication operation is completed, the next multiplication operation starts as shown in figure 5. The convolution kernel loads all the input data and weights required for the whole convolution kernel operation at once and hence no waiting time is required to load data from memory at each multiplication operation. The temporary results of multipliers are stored in buffers associated with each multiplier and the accumulated results after addition are stored in a common buffer. Note that the buffer used after accumulation is common for all the addition operations and the results of each multiplication are added with the previously stored result in the buffer.

Each convolution kernel operation is followed by Scaling and ReLU operation, and then a write operation to BRAM. This operation is also pipelined within the convolution module as shown in figure 6. As soon as a convolution operation is completed, the next convolution starts and the results of the first convolution are scaled, performed ReLU, and then written to BRAM in parallel with the second convolution operation.

The execution time for a convolution layer and ReLU layer can be expressed as,

$$T_{conv} = N_{ic} * N_{oc} * T_{ck} * {D_{oc}}^2 \tag{1}$$

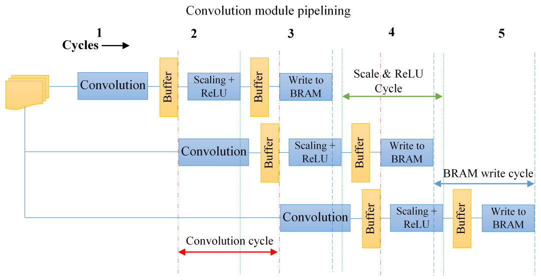

Fig. 6. Pipelining operations in the convolution module

$$T_{ReLU} = N_{oc} * T_{rk} * {D_{oc}}^2 \tag{2}$$

Where, the convolution kernel execution time,

$$T_{ck} = k^2 * (T_{mr} + T_{mul} + T_{add} + T_{mw}) \tag{3}$$

and the ReLU kernel execution time,

$$T_{rk} = T_{mr} + T_{or} + T_{mw} \tag{4}$$

Where, T_{mr}-memory read time, T_{mul}-multiplication time, T_{add}-addition time, T_{mw}-memory write time, N_{ic}-number of input channels, N_{oc}-number of output channels, D_{oc}-dimension of the convolution layer output feature map, k-convolution kernel size, T_{or}-ReLU operation time.

Since the ReLU operation is associated within the convolution kernel itself it would save the memory read and write times and also would save the temporary storage space. Then the total execution time of the convolution layer can be expressed as,

$$T_{conv} = N_{ic} * N_{oc} * T_{ck} * {D_{oc}}^2 + (N_{ic} * T_{or} * {D_{oc}}^2) \tag{5}$$

i.e.;

$$T_{conv} = N_{ic} * N_{oc} * k^2 * (T_{mr} + T_{mul} + T_{add} + T_{mw}) * {D_{oc}}^2 + (N_{oc} * T_{or} * {D_{oc}}^2) \tag{6}$$

To minimize the execution time in convolution kernels, the best possible way is to use the number of multipliers equal to k^2, which would help to compute the output in one calculation cycle. Along with pipelining, the time required for addition and memory operations can also be saved to provide the best possible execution time. But, this in fact will cause drastically increased resource utilization especially if the layer has multiple input and output channels. This is entirely dependent on the embedded hardware resources available. In order to cop up with this, we only use pipelining in kernels, but we use parallel modules for each channel operation. This would speed up the execution time significantly compared to the sequential model, but would not consume much of the resources and memory.

The total execution time of the convolution layer considering the parallelism changes, but not considering the pipelining can be expressed as,

$$T_{conv} = (k^2 * (T_{mr} + T_{mul} + T_{add} + T_{mw}) + T_{or}) * {D_{oc}}^2 \tag{7}$$

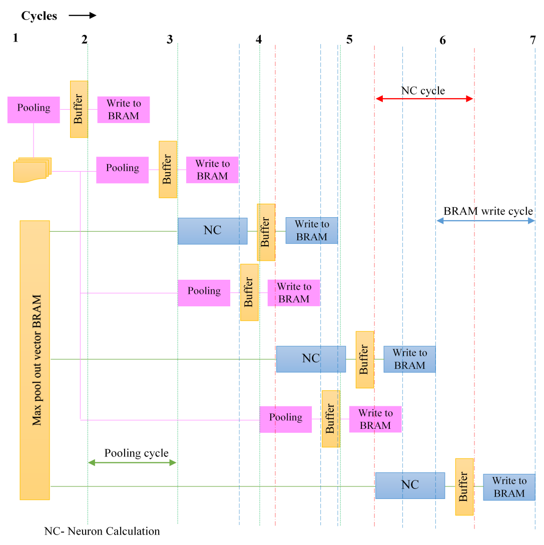

Fig. 7. Pooling layer and dense layer pipelining operation

Considering intra-layer pipelining,

$$T_{conv} = (k^2 * T_{mul}) * {D_{oc}}^2 + T_{mr} + T_{add} + T_{or} + T_{mw} \quad (8)$$

For a LeNet-5 CNN, for the convolution layer where N_{ic}=6, N_{oc}=16, k=5, D_{oc}=8, the intra-layer pipelining along with channel level parallelism provides ∼240x reduction in execution time compared to the non-optimized baseline model[9].

Max-pooling operation is also pipelined so that pooling and memory write can be performed simultaneously along with the convolution layer. Similarly, the NC operation along with memory write is also pipelined in the dense layer as in figure 7. The calculation time can be reduced further in the dense layer to almost none if the user can afford the number of multipliers in parallel equal to the number of neurons and separate BRAM units. In such case, as soon as the max-pooling operation starts writing the results to memory, the dense layer can access the data at the same time and start the MAC operation, which means no additional time is required for max-pool operation and data access from BRAM, as the execution time required for dense layer calculation module is more compared to max-pool kernel. It is shown in figure 7. As in figure 7, the max-pooling and dense layer calculations run in parallel. Since both operations require different execution times during sequential execution, the NC operation has to wait until the previous NC is completed even if the max-pool output is available. The dense layer pipelining also employs multiple pipelining strategies within the layer similar to the convolution layer. Each neuron calculation module is pipelined in sync with the pooling layer. Also, within the NC module, each multiplication operation is also pipelined exactly as in the convolution kernel shown in figure 5.

The pooling layer execution time can be expressed as,

$$T_{pool} = N_{ip} * T_{pk} * {D_{op}}^2 \quad (9)$$

Where the pooling kernel execution time,

$$T_{pk} = m^2 * (T_{mr} + T_{op} + T_{mw}) \quad (10)$$

Where, N_{ip}-number of input channels to the pooling layer, T_{op}-max-pool operation time, D_{op}-dimension of the pooling layer output feature map, and m-maxpool kernel size. Considering both channel level parallelism and intra-layer pipelining, equation 9 becomes,

$$T_{pool} = (m^2 * T_{op}) * {D_{op}}^2 + T_{mr} + T_{mw} \quad (11)$$

For LeNet-5 CNN, for the pooling layer where N_{ip}=16,m=2,D_{op}=4, the intra-layer pipelining along with channel level parallelism provides ∼45x reduction in execution time compared to the baseline model.

The best possible way to speed up the dense layer calculation is to employ the number of multipliers equal to the number of neurons. This wouldn't be wise in terms of resources even though the whole dense layer calculation can be completed in a single calculation cycle. This work proposes limited channel level parallelism where it employs the number of multipliers equal to the number of output channels of the preceding layer of the dense layer, which would calculate the dense layer operations in parallel for each neuron in each channel along with pipelining operations. This would save a huge amount of resources and the dense calculation time is reduced by a factor equal to the number of input channels to the dense layer compared to the baseline model.

The dense layer execution time,

$$T_{den} = N_{id} * {D_{od}}^2 * T_{dk} * N_n \quad (12)$$

where, the dense kernel execution time,

$$T_{dk} = T_{mr} + T_{mul} + T_{add} + T_{mw} \quad (13)$$

Where, N_{id}-number of output channels in the preceding layer, N_n-number of neurons, D_{od}-dimension of the preceding layer output feature map.

Considering both channel-level parallelism and intra-layer pipelining,

$$T_{den} = {D_{od}}^2 * T_{mul} * N_n + T_{mr} + T_{add} + T_{mw} \quad (14)$$

For LeNet-5 CNN, for the dense layer where N_{id}=16, N_n=120, D_{od}=4, the intra-layer pipelining along with channel level parallelism provides ∼40x reduction in execution time compared to the baseline model.

b. Inter-layer pipelining: Pipelining along with limited-level parallel execution is employed between the layers to reduce the latency at the expense of an increased number of BRAM and DSP blocks. In sequential execution, each channel is executed one at a time and the results are accumulated after the execution of the final channel. Whereas for inter-layer pipelining, all the channels in each layer is executed together and hence making it possible to pipeline the layers.

i) Sequential execution: During sequential execution, only a single computation module is used in each layer and it is reused for each channel calculation. Fig. 8 shows the

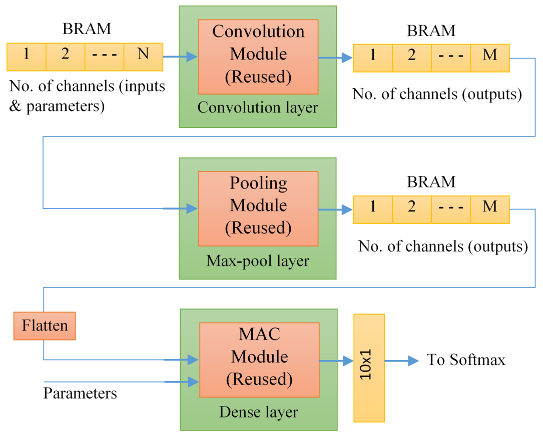

Fig. 8. Sequential execution model

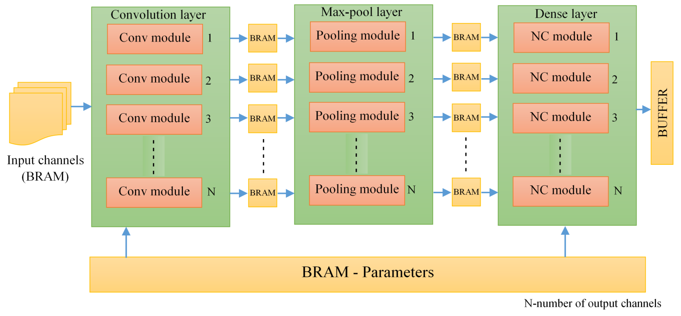

Fig. 9. Inter-layer pipelined model showing convolution, pooling and dense layers

sequential execution model. In the Convolution layer, only one channel is executed at a time and once the execution is finished, the next channel execution is started. The computed results of each channel are stored in BRAM in a sequential order one after the other. Once the convolution layer completes its execution, the max-pool layer loads the data from the BRAM and starts computation. Also for the max-pool layer, pooling operation is performed sequentially for each input channel. The result is again stored into BRAM which is then loaded to the dense layer. For the dense layer, the computation module is executed as many times as equal to the number of neurons, where the flattened output of the max-pool layer is taken as a single vector. The execution time in the sequential model increases as the number of layers and channels increases but ensures a minimal increase in resources.

For convolution layers having multiple input channels, each input channel is multiplied with corresponding weights and then each element of the outputs are added together to get the final output. Since this accumulation can only be started after the completion of all input layer calculations, we can use parallel adders to compute all the output elements together as we have all the inputs to the adders available at the same time. Using the adders in parallel, the execution time for addition operations can be eliminated. But, the execution time for additions at the output is very much less compared to the convolution operations. Hence sequential addition wouldn't cause comparable delay and also would save a lot of resources. A single adder can be reused to save resources with a negligible increase in execution time.

Pipelining of sequential model: Non-pipelined sequential model is the standard base model of the convolutional neural network. Inter-layer pipelining of this model will minimize the execution time than the non-pipelined version with a negligible increase in resource utilization. Also, note that the non-pipelined version still has the intra-layer pipelining as the basic layers are already pipelined internally. Since the processing elements and execution pattern do not vary much, it is always better to use the inter-layer pipelined version than the non-pipelined version of the sequential model. The only difference would be the execution pattern of the dense layer. Though, there are limitations as all layers cannot be pipelined completely because of the sequential execution pattern.

ii) Fully pipelined execution: In inter-layer pipelining, all layers start execution at the same time as soon as a possible computation input is available from the previous layer. To exploit the maximum possible reduction in execution time, all the channels must be executed together so that the next layer does not have to wait until all the channel executions are completed to get the final output. This model will require only the time required to calculate the last row of the activation map for all the other layers except for the first layer. Figure 8 shows the fully pipelined execution model. Such a fully-pipelined execution model also demands and exploits parallelism within the layers. This model effectively make use of on-chip BRAM as intermediate storage registers for each output channel of each layer. One important thing to be mentioned is that the size of the BRAM of each layer is dynamically selected based on the input specifications of the model to ensure that the least possible size is selected to minimize the BRAM usage.

Intra-layer Parallelism: Each layer uses a dedicated computation module for each channel and hence the execution is performed in parallel. The pipelining operation is shown in figure 10. This type of execution can reduce the size of temporary storage buffers. In sequential execution, the size of the temporary storage buffers will be the same as the size of the output of each channel whereas in the pipelined parallel model, the size of buffers will be equal to the size of a single element. This makes a huge difference in the reduction of buffer sizes. But pipelined model makes use of block RAMs for each channel output as shown in figure 9 and this can increase the number of BRAMs required compared to the sequential model. In the sequential model, the size of the BRAM increases as the number of channels are increased and the number of BRAMs remains constant. Whereas in the pipelined parallel model, the number of BRAMs will be equal to the number of channels but the size remains the same. The intra-layer parallelism is dynamically configurable according to the user-given performance metrics. Parallelism will be applied to the computation modules or to the multipliers inside the computation module or a combination of both depending on the best possible configuration to achieve the required performance.

Each layer starts execution as soon as the inputs from previous layers are available for the computation of at least one element. The max-pool layer 1 requires the first two rows from convolution layer 1 to start the computation of its first row as the kernel size of the max-pool layer is 2x2. As soon as the first 2 rows are computed in the first convolution layer, the max-pool layer starts its computation. Similarly, second convolution layer starts computation as soon as the third row is computed is max-pool layer 1. The dense layer can start its computation when the first element of max-pool layer 2 output is available.

Even though the max-pool layer starts its execution after the completion of the first two rows, it is possible to start when the first two elements of row 2 are available given that the max-pool kernel size is 2x2. But, this is not really required as the max-pool execution time is very much less compared to convolution time. Also, if the pipelining operation is scheduled element-wise instead of row-wise, it would increase the number of states in the finite state machine (FSM) for pipelining. The number of states in the FSM required for pipelining depends on the number of layers and the number of possible conditions we consider in each layer. For a 5-layer (not considering ReLU) CNN, the number of states required for a non-pipelined version would be 7 or in general, $n+2$ where n is the number of layers. For a fully pipelined version, the number of states would increase to 34, or in general 2^n+2. Also, if we consider multiple conditions, for example, considering both row and element-wise pipelining in the single layer, it would increase the number of states exponentially and each condition would be added to n. i.e.; one more condition in a layer means the number of states would increase to $2^(n+1)+2$. Hence, it is better to consider only the row-wise pipelining as it wouldn't cause any delay because of the lower execution time of the max-pool layer. To employ inter-layer pipelining for deeper networks, it is advisable to apply inter-layer pipelining in multiple sections of the network separately. A group of 5 or fewer layers will be taken as a section and parallel computation modules can be applied in the intermediate layer between two sections to reduce the execution time of the intermediate layer. In this way the complexity of the pipelining FSM can be reduced and also BRAMs can be reused in each section.

The pipelining method used in this work reduces the execution time significantly as shown below, without using many parallel blocks and without using too much of resources. Let's consider T_{total} as the total execution time.

$$T_{total} = T_{conv} + T_{ReLU} + T_{pool} + T_{den} \tag{15}$$

Where T_{conv}, T_{ReLU}, T_{pool}, and T_{den} are given in equations 2, 3, 9, and 12. Considering intra-layer pipelining and inter-layer pipelining along with channel level parallelism, all the layers except the first convolution layer and dense layer would require only the time required to calculate the last row of the feature map. The dense layer calculation starts as soon as the max-pool operation computes its first element and continues until the output is calculated as the neuron calculation time is more compared to the pooling operation. Then, equation 15 can be written as,

$$\begin{aligned} T_{total} = (k^2 * T_{mul}) * {D_{oc}}^2 + T_{mr} + T_{add} + T_{or} + T_{mw} \\ + m^2 * T_{op} * D_{op} + T_{mr} + T_{mw} + {D_{od}}^2 * T_{mul} * N_n \\ + T_{mr} + T_{add} + T_{mw} - m^2 * T_{op} * D_{op} + T_{mr} + T_{mw} \end{aligned} \tag{16}$$

$$\begin{aligned} T_{total} = k^2 * T_{mul} * {D_{oc}}^2 + 2T_{add} + 2T_{mr} + T_{or} + 2T_{mw} \\ + {D_{od}}^2 * T_{mul} * N_n \end{aligned} \tag{17}$$

For LeNet-5 CNN, for the final convolution, max-pool, and dense layers, where k=5, D_{oc}=8, N_n=120, D_{od}=4, the intra-layer and inter-layer pipelining along with channel level parallelism provides ~132x reduction in execution time compared to the baseline model.

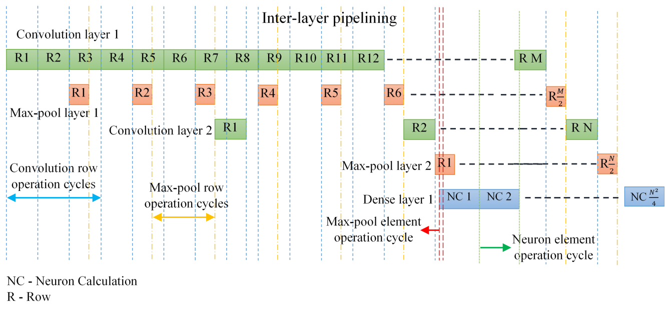

Fig. 10. Inter-layer pipelining operation

IV. Experiments & Evaluation

A. Experimental setup

The evaluation of the hardware design is done on Zedboard which uses Xilinx Zynq Z-7020 FPGA. The evaluation dataset used is the MNIST dataset for handwritten digit recognition. For the evaluation on hardware, different configurations of the proposed hardware design are set up. For inference testing and evaluation purpose, different versions of LeNet [23], [24] architecture is used. The proposed design is optimized to reduce the computational complexity and parameter count to support resource-constrained devices. To leverage the best performance from the proposed design, a custom-modified architecture of LeNet-1 [23] is implemented and is termed model 1. Model 1 has two versions, 1.a and 1.b, which are the pipelined and sequential versions of model 1 respectively. Model 1 consists of 2 convolution layers, 2 ReLU, 2 max-pool, and a single dense layer as shown in figure 11. The convolution layers have a constant filter size of 3x3 and a stride width of 1. The max-pool layers can have two different filter sizes, either 2x2 or 3x3, depending on the input to minimize the loss in accuracy. The dense layer has neurons equal to the number of classifiers. While training, the dropout layer is also introduced before the dense layer. For inference implementation, the softmax layer is replaced with a simple maximum function which would serve the purpose without using many resources

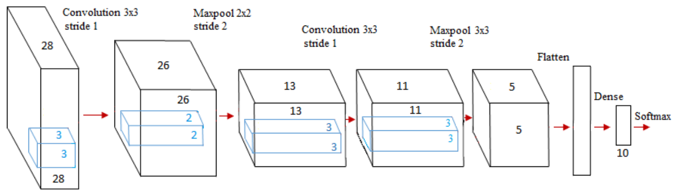

Fig. 11. Model 1 - CNN evaluation model

Model	Model 1.a	Model 1.b	Model 2.a	Model 2.b	Model 3.a	Model 3.b
Version	Pipelined	Sequential	Pipelined	Sequential	Pipelined	Sequential
Convolution layers	2	2	2	2	2	2
Conv kernel size	3x3	3x3	5x5	5x5	5x5	5x5
Filter channels	variable	variable	6, 16	6, 16	6,16	6,16
Pooling layers	2	2	2	2	2	2
Pooling kernel size	2x2 or 3x3	2x2 or 3x3	2x2	2x2	2x2	2x2
Dense layer	1	1	3	3	2	2
Number of neurons	10	10	120,84,10	120,84,10	100,10	100,10

TABLE I
Details of evaluation models used for comparison

or degrading timing performance. This model is best suited for the proposed design because it contains only a single dense layer, which is sufficient to provide comparable accuracy [21], and most importantly reduces the parameter count.

Even though increasing the dense layer count in the proposed design would deteriorate the performance for this design in terms of resources, to perform a fair comparison with the existing works which uses different hardware architectures, two other models, model 2 and 3 are also implemented by using the proposed design method. The features of all the 3 models used for evaluation are shown in table I. Model 2 is implemented by using the proposed design method which follows the architecture of LeNet-5 [24]. Model 2 has two variants, 2.a and 2.b, which are the pipelined and sequential counterparts of the same. Model 3 follows LeNet-4 architecture, which is also implemented by using the proposed design method. Model 3.a and 3.b are the pipelined and sequential versions of model 3. In dense layers, models 2 and 3 use BRAMs for temporary storage whereas model 1 uses distributed memory as the output of the dense layer of model 1 is directly fed to the softmax layer.

B. Results & Comparison

Various configurations of model 1.a and model 1.b are tried by altering the number of filter channels for each convolution channel. For sequential and pipelined models, the number of parameters and both hardware and software accuracy increases with an increase in the number of channels as shown in table II and III. For model 1.b, delay and LUT and FF utilization increase with the increasing number of channels whereas the increase BRAM is minimal as shown in table II. DSP units remain constant in model 1.b irrespective of the number of channels as the processing modules are re-used in sequential execution, which gives the user the freedom to use parallel multipliers in convolution or dense modules,

	C1-1, C2-1 channel (Model 1.b)	C1-2, C2-2 channel (Model 1.b)	C1-3, C2-3 channel (Model 1.b)	C1-4, C2-4 channel (Model 1.b)	C1-6, c2-8 channel (Model 1.b)
LUT	2549	3140	3729	4417	6576
FF	848	1177	1505	1836	4482
BRAM	2.5	3.5	3.5	4.5	5.5
DSP	2	2	2	2	2
Clock cycles required	42387	95571	160703	238317	505004
Hardware accuracy (software accuracy in bracket) (%)	76.0 (84.75)	92.0 (93.62)	94.67(95.63)	96.67(97.23)	(97.85)98.49
Parameter count	280	568	874	1198	2510

C1-convolution layer 1, C2- Convolution layer 2

TABLE II
Non-pipelined model with different filter channel configuration

and this would increase the speed of execution. In model 1.a, LUT utilization, as well as the usage of both BRAMs and DSPs, increase with an increase in the number of channels as shown in table III because of parallel execution of channels. There is only a negligible increase in execution time in model 1.a with the increasing number of channels. The percentage saving of execution time of model 1.a shows an average of 87.5% saving in time after the first convolution layer which is very much significant. It can be seen from tables II and III that for smaller networks model 1.a leverages the best performance compared to model 1.b and as the number of channels increases, the combined resource utilization of DSPs, BRAMs and LUTs of model 1.a increases rapidly. By varying pipelining and parallel execution strategies within the layers and in between the layers, different design configurations in terms of execution time and resource utilization which fall in between the sequential and pipelined models can be implemented according to the user requirements. Compared with the works in [3], [14], [19], [28] as shown in table IV, both models 2.a and 2.b provide better optimization in terms of resources. Model 2.b uses only 16% of LUTs and 5.5% of DSPs compared to form 2 of [19] and still provides a 35x reduction in delay, and model 2.a uses only 1.27x DSPs compared to form 2 and provides 509x reduction in delay. Work in [3] shows 5.75x and ∼84x speedup compared to models 2.a and 2.b respectively but shows poor resource utilization which uses 20x and 460x more DSPs compared to model 2.a and 2.b respectively. Compared to the work in [28], model 2.a shows almost similar execution time with 5x and 10x reduction in DSPs and BRAMs respectively. Though the design in work [14] shows 6.5x increase in throughput compared to model 2.b, it uses 5.7x, 12x and 41.2x LUTs, BRAMs and DSPs respectively compared to model 2.b.

The reduction in latency is very much significant after the first convolution layer with inter-layer pipelining as shown in table III. When only intra-layer pipelining is considered, the convolution layer, pooling layer and dense layer provide a speedup of ∼240x, ∼45x and ∼40x compared to the baseline model. Inter-layer pipelining along with intra-layer pipelining provides a speedup of ∼132x compared to the baseline model. For model 1.a, the execution times of all the layers after the first convolution layer got reduced by a maximum of 88.19% as shown in table III. Also, the execution times of other pooling and convolution layers got reduced by 90.65%.

	C1-1, C2-1 channel (Model 1.a)	C1-2, C2-2, channel (Model 1.a)	C1-3, C2-3, channel (Model 1.a)	C1-4, C2-4, channel (Model 1.a)	C1-6, C2-8 channel (Model 1.a)
LUT	2113	3718	5753	7550	15526
FF	1367	2208	3115	4090	7941
BRAM	3	5	7	9	16
DSP	4	10	18	28	72
Clock cycles required	33495	33555	33615	33675	33975
Hardware accuracy (software accuracy in bracket) (%)	76.0 (84.75)	92.0 (93.62)	94.67(95.63)	96.67(97.23)	97.85(98.49)
% time saving after 1st convolution layer	88.19%	87.93%	87.72%	87.50%	86.44%
Parameter count	280	568	874	1198	2510

C1-convolution layer 1, C2- Convolution layer 2

TABLE III
pipelined model with different filter channel configuration

TABLE IV
Resource utilization, delay and performance comparison with different hardware designs

	Platform	clock	Precision	DSP	BRAM	LUT	FF	GOP/s	GOP/s/DSP	Delay
[14] (LeNet-5)	Zynq Z-7020	100 MHz	16-bit fixed	206	144	38136	42618	39.78	19.3x	
[28] (LeNet-5)	Virtex7 485t	NA	16-bit	564	571	15285	2074	5.2	9.22x	1.1ms
[28] (LeNet-5)	Virtex7 485t	NA	8-bit	574	343.5	7204	1316	6	10.45x	0.96ms
[3] (LeNet-5)	Zynq UltraScale+	50MHz	4-8bit	2300	466	30984	63555	NA	NA	0.173ms
[19] form1 (LeNet-5)	Zynq Z-7020	100MHz	25 bit	20	27	14832	54075	NA	NA	26.36ms
[19] form1 (LeNet-5)	Zynq Z-7020	100MHz	25 bit	90	3	39879	35399	NA	NA	506.93ms
Model 2.a (LeNet-5)	Zynq Z-7020	100 MHz	8-16-bit	115	54	40897	21285	2.25	19.56x	0.996ms
Model 2.a (LeNet-5)	Virtex7 485t	100 MHz	8-16-bit	115	54	40930	21290	2.25	19.56x	0.996ms
Model 2.a (LeNet-5)	Zynq UltraScale+	100 MHz	8-16-bit	115	54	39919	21345	2.25	19.56x	0.996ms
Model 2.b (LeNet-5)	Zynq Z-7020	100 MHz	8-16-bit	5	20.5	6678	8574	0.148	29.6x	14.471ms
Model 2.b (LeNet-5)	Virtex7 485t	100 MHz	8-16-bit	5	20.5	6678	8574	0.148	29.6x	14.471ms
Model 2.b (LeNet-5)	Zynq UltraScale+	100 MHz	8-16-bit	5	20.5	6685	8575	0.148	29.6x	14.471ms

This shows how effectively we can reduce the execution time without increasing the resources. The comparison of prediction times with varying parameter counts is shown in table V. Compared to [13], all the models presented in this work show at least thousands of fold speedups even with increased parameter count as shown in table V. Compared with the percentage of prediction misses in [13], all 3 models shown in table V shows better accuracy and performance in terms of execution time for the same parameter count.

While considering the resource constraints, this work also ensures to have a more satisfactory performance by targeting resource-constrained devices. The throughput of a CNN design is very much dependent on the number of DSP units in use. The complexity of the network is expressed in terms of Giga Operations (GOP). As shown in table VI, model 2.b shows $\sim$3x throughput density in terms of GOP/s/DSP with $\sim$115x less DSP slices and $\sim$48x less BRAM units compared to [28] and can process 69.1 frames per second (FPS) at 13.82 FPS/DSP which is $\sim$8x more compared to [28]. Both models 2.a and 2.b have the best FPS/DSP value compared to [3], [4], [27], [28]. Model 1.a is configured for the same accuracy as LeNet-5 for the comparison in table VI and it shows the best throughput of all the models in terms of FPS/DSP with at least $\sim$6x times less parameter count compared to all the models shown in table VI. Only [27] shows better throughput in terms of GOP/s/DSP compared to our models but models 1.a, 2.a and 2.b has 228x, 98x and 155x increase in FPS/DSP value respectively compared to the best possible value of [27].

TABLE V
Delay and accuracy comparison with increase in parameters

Parameter count		[13] (LeNet-4)	Model 1.a	Model 3.a	Model 3.b
1000	Prediction time	1.097s	0.336ms	0.761ms	1.217ms
	Misses (%)	5.27	3.69	3.91	3.91
2500	Prediction time	1.108s	0.339ms	0.763ms	3.425ms
	Misses (%)	1.67	1.51	1.66	1.66
5000	Prediction time	1.125s	0.342ms	0.770ms	5.938ms
	Misses (%)	1.24	1.26	1.26	1.26
10000	Prediction time	1.160s	0.351ms	0.784ms	7.386ms
	Misses (%)	1.14	0.96	1.11	1.11
$\sim$30000	Prediction time	1.375s	Nil	0.906ms	13.723ms
	Misses (%)	1.12	Nil	1.07	1.07

TABLE VI
Comparison of throughput with existing works

	FPGA	logic	Network	GOP/s	GOP/s/DSP	FPS	FPS/DSP
[27]	Stratix-V GSD8	OpenCL	AlexNet	72.4	99.58x10^{-3}	49.75	6.84x10^{-3}
[27]	Stratix-V GXA7	OpenCL	AlexNet	31.8	129.27x10^{-3}	21.88	8.89x10^{-3}
[28]	Virtex7 485t	HLS	LeNet-5	5.2	9.22x10^{-3}	909.09	1.61
[28]	Virtex7 485t	HLS	LeNet-5	6.0	10.45x10^{-3}	1041.66	1.81
[3]	Zynq UltraScale+	HLS	LeNet-5	NA	NA	5780.34	2.51
[4]	Virtex7 VX485T	HLS	AlexNet	61.62	22.0x10^{-3}	46.27	1.65x10^{-3}
Model 1.a	Zynq Z-7020	RTL	LeNet	2.99	20.78x10^{-3}	2923.97	20.30
Model 2.a	Zynq Z-7020	RTL	LeNet-5	2.25	19.56x10^{-3}	1004.02	8.73
Model 2.b	Zynq Z-7020	RTL	LeNet-5	0.148	29.6x10^{-3}	69.10	13.82

V. Conclusion

In this work, a hardware design is proposed to optimize the hardware implementation on resource-constrained devices and an automated hardware code generator is implemented. This design is able to ensure the best throughput and reduced execution time without adversely affecting resource utilization, with the help of intra and inter-layer pipelining, limited channel level parallelism, efficient use of on-chip BRAMs, adaptive data processing, and bit-width quantization strategies. Experimental results show that this design shows better performance in terms of throughput and resource utilization compared to the related works.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. 'ImageNet Classification with Deep Convolutional Neural Networks'. Advances in neural information processing systems.

[2] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, TobiasWeyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861.

[3] C. Y. Lo, F. C. M. Lau and C. Sham. 2018. Fixed-Point Implementation of Convolutional Neural Networks for Image Classification. International Conference on Advanced Technologies for Communications (ATC), Ho Chi Minh City.

[4] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. FPGA.

[5] C. Zhang, Z. Fang, P. Zhou, P. Pan, C. Jason. 2016. Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks. Proceedings of the 2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2016).

[6] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going deeper with convolutions. arXiv:1409.4842.

[7] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. 2017. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡ 1MB model size. ICLR.

[8] H. Sharma, J. Park, E. Amaro, B. Thwaites, P. Kotha, A. Gupta, J.K. Kim, A. Mishra, H. Esmaeilzadeh. 2016. Dnnweaver: from high-level deep network models to fpga acceleration. the Workshop on Cognitive Architectures.

[9] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, J. K. Kim, V. Chandra, and H. Esmaeilzadeh. 2018. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks. ISCA.

[10] Jiantao Qiu, JieWang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. 2016. Going deeper with embedded FPGA platform for convolutional neural network. ACM International Symposium on FPGA.

[11] Jinwei Xu, Zhiqiang Liu, Jingfei Jiang, Yong Dou, and Shijie Li. [n. d.]. CaFPGA: An automatic generation model for CNN accelerator.

[12] 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), 770–778.

[13] L. B. Saldanha and C. Bobda. 2016. Sparsely connected neural networks in FPGA for handwritten digit recognition. 17th International Symposium on Quality Electronic Design (ISQED), Santa Clara, CA.

[14] L. Gong, C. Wang, X. Li, H. Chen and X. Zhou. 2017. Work-in-progress: a powerefficient and high performance FPGA accelerator for convolutional neural networks. International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Seoul.

[15] Lin M, Chen Q, and Yan S. 2014. Network in network. In Proc. ICLR.

[16] Motamedi M, Gysel P, Akella V, and Ghiasi S. 2016. Design Space Exploration of FPGA-Based Deep Convolutional Neural Networks. 21st Asia and South Pacific Design Automation Conference (ASP-DAC).

[17] P. Gysel, M. Motamedi, and S. Ghiasi. 2016. Hardware-oriented Approximation of Convolutional Neural Networks. ICLR.

[18] R.A. Solovyev, A. A. Kalinin, A. G. Kustov, D. V. Telpukhov, V. S. Ruhlov. 2018. FPGA Implementation of Convolutional Neural Networks with Fixed-Point Calculations. arxiv: 1808.09945.

[19] S. Ghaffari and S. Sharifian. 2016. FPGA-based convolutional neural network accelerator design using high level synthesize. 2nd International Conference of Signal Processing and Intelligent Systems (ICSPIS), Tehran.

[20] S. Tripathi, G. Dane, B. Kang, V. Bhaskaran, and T. Nguyen. 2017. LCDet: Low-Complexity Fully-Convolutional Neural Networks for Object Detection in Embedded Systems. 2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW).

[21] Springenberg Jost Tobias, Dosovitskiy Alexey, Brox Thomas, and Riedmiller Martin. 2014. Striving for simplicity: The all convolutional net. arXiv preprint arXiv:1412.6806.

[22] Stylianos I. Venieris and Christos-Savvas Bouganis. 2017. fpgaConvNet: A Toolflow for Mapping Diverse Convolutional Neural Networks on Embedded FPGAs. Workshop on Machine Learning on the Phone and other Consumer Devices (MLPCD), NIPS.

[23] Y. LeCun et al. 1995. Comparison of learning algorithms for handwritten digit recognition. F. Fogelman and P. Gallinari, editors, International Conference on Artificial Neural Networks, Paris.

[24] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient based learning applied to document recognition. Proceedings of the IEEE.

[25] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA.

[26] Ying Wang, Jie Xu, Yinhe Han, Huawei Li, and Xiaowei Li. 2016. DeepBurning: Automatic Generation of FPGA-based Learning Accelerators for the Neural Network Family. Proceedings of the 53rd Annual Design Automation Conference (DAC'16). ACM, New York, NY, USA, Article 110.

[27] Yongmei Zhou and Jingfei Jiang. 2015. An FPGA-based accelerator implementation for deep convolutional neural networks. 4th International Conference on Computer Science and Network Technology (ICCSNT), Harbin, China.

[28] Z. Li et al. 2017. Laius: An 8-Bit Fixed-Point CNN Hardware Inference Engine. IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC), Guangzhou.