FPGA-Optimized Parallelism in Deep Neural Networks

*University of Illinois at Urbana-Champaign, CS 533 Course Project

Yanzhuo Chen
Electrical and Computer Engineering
yanzhuo2@illinois.edu

Mingkai Miao
Electrical and Computer Engineering
mmiao2@illinois.edu

***Abstract*—Deep Neural Networks (DNNs) are gaining increasing popularity in artificial intelligence due to its excellent capacity on interpreting complicated data patterns. However, despite the advantage of better capturing features, applications based on DNN models usually take more execution time in comparison with traditional algorithms, arousing a trend for optimizing the parallelism on its computation to support some real-time applications. Field-Programmable Gate Arrays (FPGAs) are often utilized as a suitable platform for the acceleration of DNNs due to its great performance and high reconfigurability. High-Level Synthesis (HLS) in FPGAs further provides developers a convenient way to optimize performance by using C/C++ and pragmas instead of writing hardware description languages. However, for resource-limited scenarios, hardware optimizations only cannot expose adequate parallelism and also needs software level support for optimal performance.**

In this project, we leverage advantages brought by FPGAs and HLS for the optimization of DNNs, particularly the famous LeNet model, on Xilinx PYNQ-Z2 platform to achieve acceleration on hand-written digit classification task on MNIST dataset. We delicately implement our accelerator with model architecture optimization techniques like quantization and pruning and software interrupt support for function-level pipeline, which helps achieve a speedup of 63.7x. The relevant project files could be found at: https://github.com/tracymiao111/CS533-FinalProject.git.

I. Introduction

Rapid development of computational devices allows researchers to design and train more complicated deep learning models in order to obtain greater performance on target applications [1], which in turn poses pressures on edge devices to accomplish the task within a satisfactory time period, especially for some workloads with real-time demands. It becomes an issue of how to do accelerations on the computation of DNNs when low latency is essential. Traditional CPUs and GPUs could not be very suitable under this case since the former is inefficient on matrix computation while the latter can be power consuming, limiting their scenarios if the application is sensitive to the corresponding disadvantage [2], [3].

FPGAs provide a potential solution on this dilemma due to its flexibility on software side and well-known parallel processing capabilities with desirable efficiency on hardware side, which is ideal for custom DNNs acceleration tasks [2], [4]. In addition, HLS offers developers a simplified approach to implement complicated algorithms in HDLs via writing C/C++ and HLS pragmas. For some cases, designs written by HLS can be comparable with manual RTL designs but with much less time consumption [5].

In our project, we focus on the acceleration of the pioneering model LeNet with the hand-written digit classification task on MNIST dataset. Our previous attempts have been solely on the hardware-side acceleration, achieving a speedup of approximately 14x on the single image processing task with a less optimized implementation [6]. During this midterm, we try to further explore the HLS optimization space to obtain a higher speedup (roughly 20x) on the single image inference and find it tough to design an efficient acceleration solely with hardware optimization [7]. Given the circumstances, in addition to some adjustment on our HLS accelerator design, we implement techniques including quantization, pruning and make the system asynchronous by enabling interrupt on the software side, which helps us achieve a **63.7x** speedup on single image inference on average.

II. Background

A. Deep Neural Networks and LeNet

Recently, the breakthrough of Sora, an AI model with abilities of creating realistic and artificial scenes from text input [9], surging a great interest even among non-relevant practitioners due to its superior performance demonstration. As the precursor of Sora, becasue of its powerful feature representation capacity, Deep Neural Networks (DNNs) have been gaining popularity for years in various areas including computer vision and natural language processing [1].

The desirable high accuracy comes from complicated model configurations with substantial computational complexity, presenting a challenge on applications sensitive to forward propagation latency that have real-time processing demand [4]. For instance, the true-time image processing task while autonomous driving and scenes generation on models like Sora.

Our project focuses on the acceleration of LeNet [8], a classic pioneering of DNNs proposed by Yann LeCun, well known for its elegant simplicity and impressive performance. LeNet has been successfully applied to many practical tasks related to image processing and recognition. The detailed configuration is demonstrated in **Figure 1**, in which the major structure are first two convolution and maxpooling layer pairs followed by two fully-connected layers.

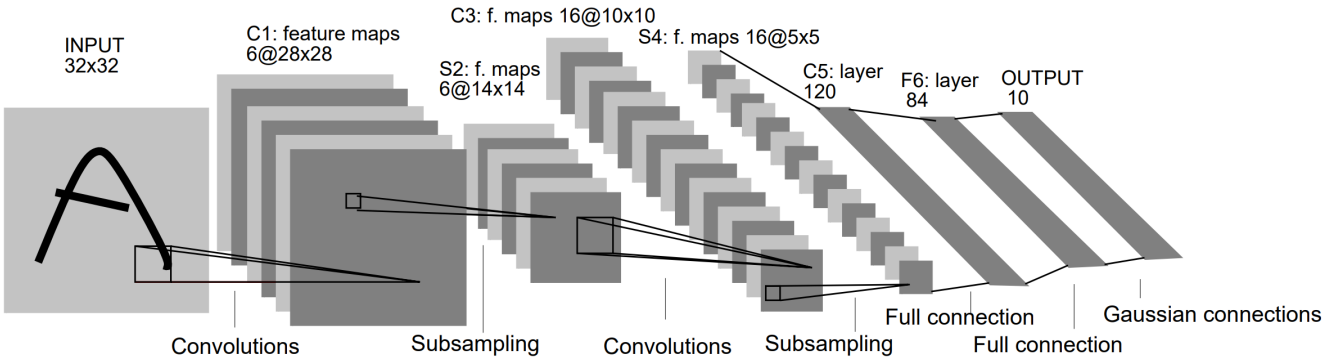

Fig. 1. Overall Archituecture of LeNet [8]

In addition, LeNet is a suitable choice for FPGA based acceleration implementation not only due to its relatively small size in comparison with modern huge deep models but also the satisfactory effectiveness. Its wide applicability and historical significance also become major factors why LeNet is selected as our target model. We think the experience and strategies for accelerating DNNs are generally interconnected, and the valuable insights as well as methodologies obtained from accelerating LeNet could also be applied to other modern DNN models.

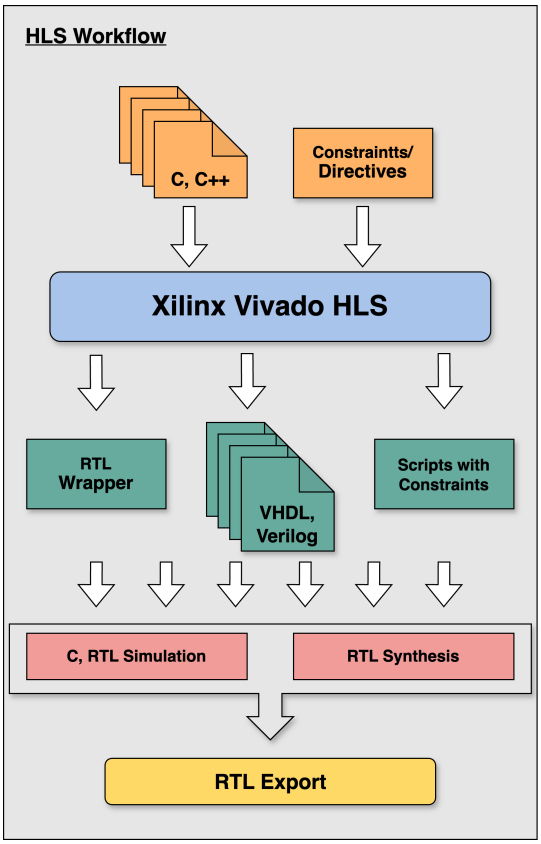

Fig. 2. High-Level Synthesis Work Flow in Vivado HLS

B. High-Level Synthesize

Traditionally, people design hardware accelerators via writing hardware description languages (HDL), resulting in complicated and time-consuming developing process. High-Level Synthesize [10] in FPGAs provides a potential solution towards this issue by enabling an automated design process which does a transformation from high-level functional specifications into optimized register-transfer level (RTL) descriptions for convenient and efficient hardware implementation. Due to this, developers are allowed to use C or C++ programming languages instead of HDL like Verilog, decreasing the required time for hardware design then speeding up the overall iterations of the accelerator. In addition, HLS pragmas make the accelerator design further convenient, by which delicately designed parallelism, data movement, resource utilization management are allowed to be implemented. Selected examples of HLS pragmas include [5]:

(1) **#pragma HLS pipeline**: Accelerates a loop by allowing multiple iterations to be executed in parallel, reducing loop latency.
(2) **#pragma HLS unroll**: Expands a loop, creating multiple copies of the loop body, which can enhance parallel processing.
(3) **#pragma HLS array_partition**: Divides arrays into smaller, independently accessible partitions, enabling parallel access to the data and improving data throughput.
(4) **#pragma HLS dataflow**: Allows functions or loop bodies to execute concurrently, optimizing the overall data path and potentially reducing the latency.

As shown in **Figure 2**, the overall workflow begins with the design code written in C or C++, along with the corresponding constraints and directives used for guiding the HLS process. After that, the HLS tool can help generate the RTL wrapper, HDL codes as well as scripts with constraints, which is followed by the simulation and synthesis. If the simulation and synthesis are all passed successfully, there will be a HLS synthesis results which illustrates the performance as well as the resource utilization of the exportable hardware design.

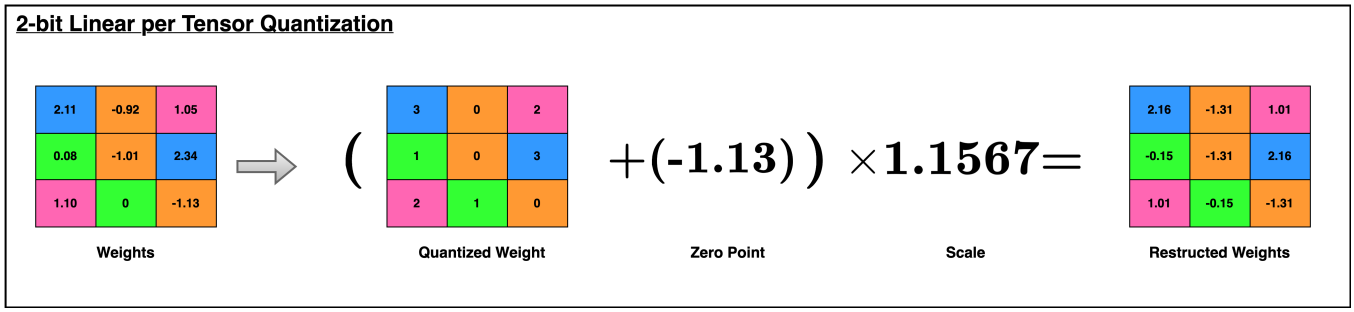

Fig. 3. 2-bit Linear per Tensor Quantization

C. Quantization

Quantization, in mathematics and digital signal processing, is the process of mapping input values from a large set to output values in a smaller set, often with a finite number of elements [11]. Similarly, model quantization, is a technique that reduces the memory and computational cost during inference by representing weights or activations with lower bit data format (4-bit or 8-bit integer) instead of higher bit format (16-bit or 32-bit floating point), which further would reduce the total amount for energy for a single iteration of inference because the computation is in lower-bit format. The comparison of energy consumption on computation of different data types and operations is illustrated in **Table I** below.

TABLE I
ENERGY CONSUMPTION FOR COMPUTATION IN DIFFERENT FORMAT [12]

Integer Format		**Float Format**	
Add		**FAdd**	
8 bit	0.03 pJ	16 bit	0.4 pJ
32 bit	0.1 pJ	32 bit	0.9 pJ
Mult		**FMult**	
8 bit	0.2 pJ	16 bit	1.1 pJ
32 bit	3.1 pJ	32 bit	3.7 pJ

Numerous innovations in both software and hardware domain have emerged to support quantization in models.

From hardware perspective, Nvidia introduces a new computation unit called Tensor Core since Volta Architecture [13], which computes a small-sized matrix multiplication and speedup the overall performance when doing tiled matrix multiplication but only supports FP16 in Volta. After Turing architecture, tensor core also supports lower bits integer data type (INT8) so that some integer-only quantization strategy can benefit not only on software level but also with specialized hardware support.

From software perspective, different strategies on how to do quantization have been discussed. Among them all, symmetric linear quantization [14] is most widely used, which maps a group of floating point numbers into integer format representation with a single zero point and scaling factor, and recover it during inference time.

To be more specific, suppose a given weight matrix of size 3x3 in 32-bit floating point as below. We can quantize this matrix into a 2-bit unsigned int matrix using the following equation.

$$Zero = min(W) \tag{1}$$

$$Scale = (max(W) - Zero)/(2^{nbits} - 1) \tag{2}$$

$$W^Q = (W - Zero)/Scale \tag{3}$$

By subtracting the minimum number of the weight matrix, we map all the weights into positive range, and to store it in n bits (which is 2 in our given scenario), we cut the positive range of $(max(W) - min(W), 0)$ into $2^{nbits} - 1$ bins, and recover the original weight matrix using the below equation during inference.

$$W^* = (W^Q + Zero) * Scale \tag{4}$$

In this case, we can reduce the memory to $(2 * 3 * 3 + 32 + 32) = 82$ bits, which is only 28.5% of original $32 * 3 * 3 = 288$ bits. Since this method needs to recover the weight matrix back to its original data format, it can not benefit from computation or energy, however, it is still widely used because its easy computation process and high compression ratio. Still, as you can see, there is numerical loss from original weight and covered weight matrix, and in some cases, it would lead to accuracy loss. In order to recover accuracy, sometimes it would need further retraining of the model to fix the accuracy loss. The computation process is illustrated in **Figure 3**.

D. Pruning

The model size nowadays is getting larger and larger, however, the distribution of the model weights often reveals that not all weights matter, and in some cases most of the weights do not contribute much to the accumulative result because their magnitude is close to zero. Therefore, pruning away those unimportant weights can further reduce memory and computational cost [15].

There are mainly two types of pruning schemes, fine-grained and structured pruning, as is shown in **Figure 4**. For fine-grained pruning, the weights that have the least importance

would be pruned away no matter of their structural information. However, this introduces complexity for computation during inference time because sparsity of the weight matrix is random if their locations are not kept with extra memory, then the sparsity cannot be leveraged and does not bring any benefits. Therefore, in production environment, structured pruning is more often used.

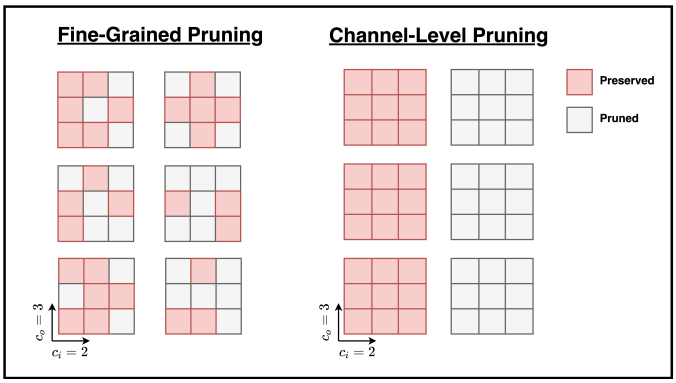

Fig. 4. Fine-Grained Pruning vs Channel-Level Pruning

Structured pruning enforces the pruned weights to have a specific structured format so that there would be no or little memory overhead to memorize its sparsity locations and benefit computation. For example, for convolutional neural network, pruning per channel scheme is often used because when a whole channel of weights is pruned, and the overall computation pattern is still a matrix multiplication only with a different size and it also satisfies the intuition that some channels may contain relatively redundant information covered in other channels already.

III. Methodology

To achieve better performance and utilization of FPGA resources, we design our accelerator for LeNet from hardware, model architecture and software perspective.

For hardware, we would use HLS for accelerator design and the provided HLS primitives to tune the accelerator for better performance, use it as a baseline and see what further optimizations are enabled by above model structure changes.

For model architecture, we would try the model compression technology including quantization and pruning discussed above to see what benefits they bring on an end-to-end inference performance by simplifying hardware design with fewer memory and computation cost.

For software, we would utilize the module pipeline on the hardware by enabling software level interrupt and designing its corresponding handler for a pipelined inference process.

The **Figure 5** demonstrates the overall hardware block design. Basically, AXI buses take the responsibility of data movement while AXI-Lite ones are utilized to send and receive control signals according to their differences on port bandwidth. In addition, parameters are transferred from SD to BRAMs for more efficient data accesses.

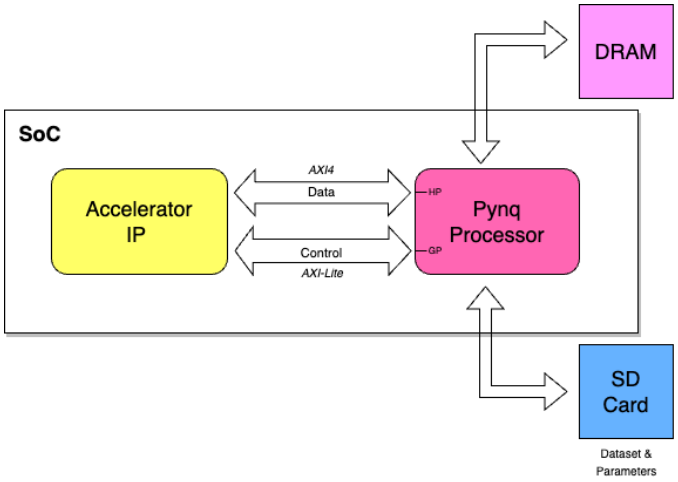

Fig. 5. Overall Hardware Block Design

A. Hardware Optimization

We use HLS to delicately design the accelerator. Various HLS pragmas including pipelining, array partitioning and dataflow are implemented strategically layer by layer to achieve better customized performance. Weights and biases for each layer are stored in BRAM instead of storage for better latency elimination. In addition, layer fusions and adder trees are applied to selected layers for optimized balance between resource utilization and overall performance. These techniques with corresponding figures for demonstration are discussed in details in following subsections.

1) Layer Fusion: Effective loop fusion could improve the loop parallelism as well as the data locality [16], [17], with higher DSP and BRAM utilization. As demonstrated in **Figure 6**. We implement fusions of all the ReLU activation functions with its preceding layers.

In order to fuse the maxpooling layer with the corresponding convolutional layer, an intermediate array is allocated to hold the temporary results between these two layers, eliminating the intermediate buffering demand and reduce the latency as a result. However, under the consideration of the balance between performance increase and limited resources, only the initial convolutional layer with its maxpooling layer are fused mainly due to its moderate amount of parameters and the remarkable reduction on latency it could bring.

By implementing these layer fusions, it could obtain a roughly 18% overall improvement on its latency according to Vivado HLS Synthesis results.

2) Pipelining and Array Partition: Inserting #pragma HLS pipeline before the loop can automatically unroll the inner loops and allow multiple iterations to be executed in a parallel way, which could reduce the loop latency to a large extent. However, merely applying HLS pipelining is not able to achieve the optimal result since the pipeline rate is essentially limited by the number of BRAM ports which could burden the optimization of the initiation interval. For example, there are multiple reads on a particular array but the memory is

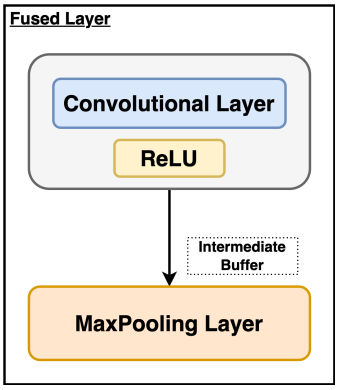

Fig. 6. Implementation of Layer Fusion

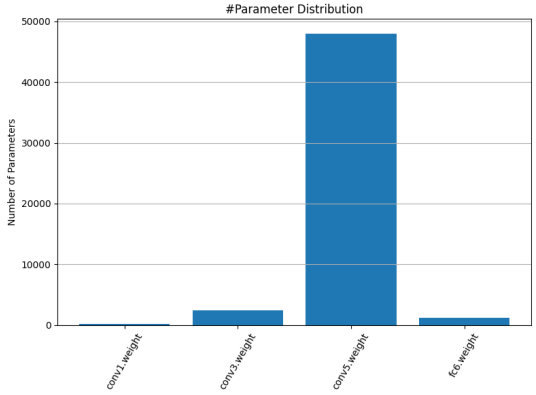

Fig. 8. Number of Parameters Per LeNet Layer

only equipped with one or two available ports, resulting in a bottle neck. To tackle this problem, array partition technique is proposed to independently access partitions, enabling parallel access to the data. In terms of the large utilization on LUTs consumed by array partition, we only apply array partitions to three convolutional layers because of their large weights arrays with either cyclically or completely according to the resource and latency concerns.

3) Dataflow: To further exploit the parallelism, the #pragma HLS dataflow is implemented to enable task-level pipelining, which allows individual functions and loops can overlap during their execution. The HLS tool takes the responsibility of examining the dataflow possiblity between sequential functions or loops and then establish channels using RAMs or FIFOs, allowing consumer's execution to begin before that of producer has accomplished.

4) Adder Tree: To reduce the depth of addition loop, adder trees are implemented in selected layers for parallel arrangement of adders. It is found to be highly effective on the computation speed improvement if adder trees are integrated into add operations inside a layer. Therefore, considering the remarkable elimination on latency and enough resource, we apply adder trees to all the convolutional layers, of which the architecture is shown in the **Figure 7** below:

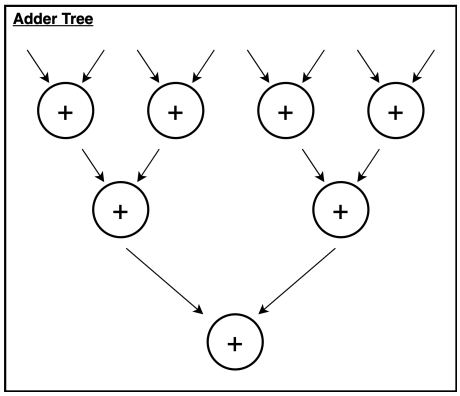

Fig. 7. Architecture of Adder Tree

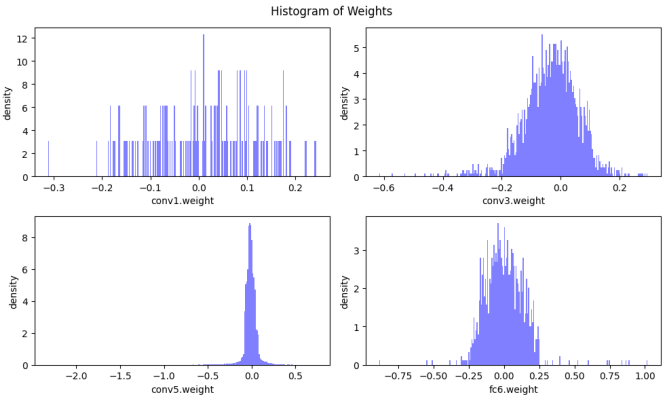

Fig. 9. Weight Distribution for Each Layer

5) Hardware Optimization for Each Layer: The **Table II** below illustrates the hardware optimization configuration for each layer.

B. Model Architecture Optimization

1) Quantization: Since the third layer of convolution constitutes the majority of LeNet, we focus on how to do quantization on the weights of this single layer and what further hardware design optimizations are enabled.

We would use the symmetric linear quantization discussed above for the quantization process. However, we would still need to decide on what granularity to perform the quantization. For example, as demonstrated in **Figure 8** and **Figure 9** this conv_5 layer has a weight size of 48000 and most of the weights are within the range of $(-0.5, 0.5)$, and therefore, it is possible to have a relatively small accuracy loss on tensor level quantization with a larger bitwidth and channel level quantization with a smaller bitwidth.

The original accuracy is 98.39%, and as is shown in the **Table III**, for per-tensor 4-bit linear quantization can achieve an acceptable result with merely no accuracy loss while for per-channel 2-bit linear quantization can achieve an acceptable result with roughly 4% accuracy loss. However, as is shown in **Table IV**, per-channel quantization scheme would have more memory overhead because it needs a zero point and scaling

TABLE II
HARDWARE OPTIMIZATIONS FOR EACH LAYER

Layers	Configurations
Fused_1	Array Partition; Pipeline; Layer Fusion (Conv_Relu_1, MaxPool_2); Adder Tree
Conv_ReLU_3	Array Partition; Pipeline; Adder Tree
MaxPool_4	Pipeline
Conv_ReLU_5	Array Partition; Pipeline; Adder Tree
Dense_ReLU_6	None
Overall	Dataflow

TABLE III
ACCURACY AFTER QUANTIZATION

Number of Bits	Per Tensor	Per Channel
1	0.1032	0.5849
2	0.0982	0.9479
4	0.9770	0.9822
8	0.9836	0.9837

factor for every channel while for per-tensor quantization there would only be one zero point and one scaling factor.

TABLE IV
QUANTIZATION COMPRESSED RATIO

Number of Bits	Per Tensor	Per Channel
1	0.0313	0.1113
2	0.0625	0.1425
4	0.1250	0.2050
8	0.2500	0.3300

Therefore, considering the overall compression ratio, it would be wise to choose 4-bit per-tensor quantization for this project. However, in HLS any data format that is not supported in C/C++ would need a special type of ap_int, which cannot be fused together with primitive integer data type. Therefore, due to the project complexity, the following part would use 8-bit per-tensor quantization scheme to illustrate what hardware optimizations are further enabled.

For a unrolled loop, it would create multiple read & write to the same memory, however, for limited number of ports of a given synthesized memory, it might serialize the memory access which in theory could be parallelized. For example, for the fused module of convolution_5_ReLU, the given pipeline would unroll the inner loop and create multiple parallel memory access to weights exceeding the total number of memory ports. In normal cases, we would need to partition the weight memory to create more ports. However, since weight array is of a relatively large size, array partitioning would introduce a large overhead of LUTs exceeding the amount of LUTs available. However, if we can compress the weight matrix of this convolution to 8-bit unsigned integer instead of 32-bit floating point, this would enable us to do array partition on the weight array without exceeding the number of LUTs.

2) Pruning: As discussed above, channel pruning is more friendly for hardware implementation because it keeps the original computation pattern. However, running a sensitivity analysis using random pruning could help us decide which layer is more sensitive to pruning.

As is shown in the **Figure 10**, convolution_1 and convolution_3 are more sensitive to sparsity, while convolution_5 and fc_6 are less sensitive. Meanwhile, convolution_5 constitutes the majority of the weight, and thus we would still only prune the convolution_5 layer.

Still, we need an algorithm to decide which channels to prune away given the sparsity ratio. One naive way is to leave out the fixed last few channels, however, this ignores the information of the pretrained weight importance and may lead to a severe accuracy loss as well as longer time to retrain. Therefore, we would evaluate the importance of each layer based on the Frobenius norm of the weight array. The result is shown in **Table V**.

TABLE V
ACCURACY AFTER PRUNING

Sparsity	Naïve Last Channels	Frobenius Norm
0.5	0.2775	0.971
0.7	0.2775	0.906
0.8	0.1005	0.839
0.9	0.1005	0.3265

Even though the accuracy drops a lot for a relatively high sparsity, this often can recover after a few epochs of retraining, as is shown in **Table VI**.

TABLE VI
ACCURACY BEFORE AND AFTER 5-EPOCHS OF RETRAINING

Sparsity	Before Retraining	After Retraining
0.7	0.9060	0.9795
0.8	0.8390	0.9705
0.9	0.3265	0.9665
0.95	0.2680	0.9355

Based on the accuracy after retraining, we would choose sparsity of 0.9, which in our case would reduce the number of input channels of convolution_5 from 16 to 2, and also implicates that the former convolution can also reduce its number of

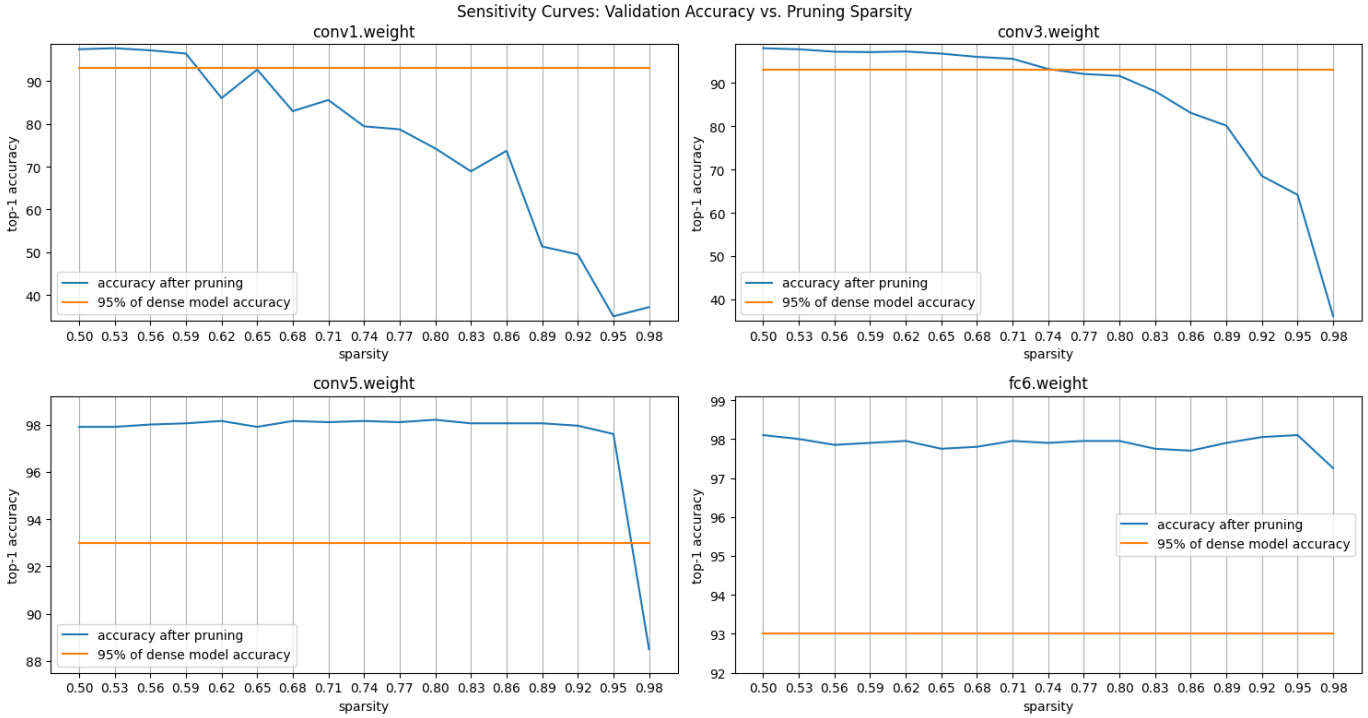

Fig. 10. Pruning Sensitivity Curve

output channels from 16 to 2. Unlike the quantization scheme we use, pruning not only reduces large amount of resources for memory but also reduces large amount of computation, further leading to a great end-to-end inference speedup. Our overall pruning configuration could be demonstrated in **Figure 11**.

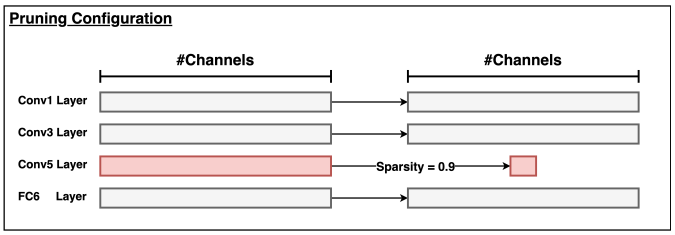

Fig. 11. Pruning Configuration

C. Software Optimization

Even though on hardware level, pipelined executions are enabled between different functions (hardware modules in HLS form), still, to utilize this pipeline we need to enable interrupt in our software implementation and block design.

Formerly, a new frame could only be input when the whole execution completes. However, with this interrupt enabled, we can feed the input to accelerator as soon as it is ready. By moving the read logic to the software interrupt handler, output read would not block the execution of image input. The **Figure 12** shows the difference between dataflow model with or without interrupt support in DNNs acceleration.

IV. Result Analysis

Comparison for results are modeled from two perspectives:

- We compare the inference speedup with the baseline model running on PYNQ-Z2 CPU without any optimizations.
- We compare the inference speedup with the baseline* model (only hardware side optimization is implemented) to further demonstrate the effectiveness brought by model architecture and software side support.

As is shown in **Table VII**, quantization slightly increases the speedup compared to the baseline* implementation while pruning has a larger speedup increase compared to quantization. The most effective improvement is brought by enabling software interrupt, which almost doubles the speedup compared to the same implementation without it.

This satisfies our expectation because our quantization method only brings benefits to memory resources because we would recover it back to FP32 without reducing the total amount of computation, leading to the limitation on the improvement. However, pruning reduces not only the memory requirement but also the total amount of computation so that it can achieve a much higher speedup improvement. As for software interrupt, it allows functional level pipeline on hardware that helps to overlap large proportion of computation and data movement between different function modules.

Still, we wonder what is the theoretical maximum speedup of our overall system, an experiment is conducted in which the accelerator only does two memory read, a computation, and one memory write in order to see the extreme speedup by

TABLE VII
RESOURCE UTILIZATION & SPEEDUP RESULTS

	LUT	FF	BRAM	DSP	Speedup
Baseline	-	-	-	-	1x
Baseline*	42980	39877	140	156	21.5x
Baseline* + Quantization	45453	46660	73	184	24.3x
Baseline* + Pruning	29549	31348	50.5	159	30.7x
Baseline* + Pruning + Interrupt	29549	31348	50.5	159	**63.7x**

reducing the execution time of parallel proportion to almost 0. We get a speedup of 125.6x with this implementation, which infers that in order to further speedup our implementation, it is supposed to run some additional software profiling and improve the sequential proportion of our software implementation.

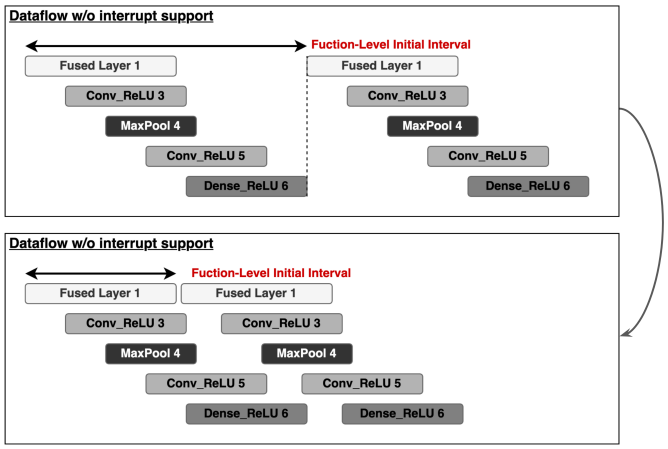

Fig. 12. Interrupt Handler

V. CONCLUSION

In this project, we explore how to design a DNN accelerator from perspectives of hardware, model architecture and software. It is discovered that for scenarios like FPGA with limited resources, it would be hard to obtain an ideal speedup with hardware optimizations solely due to limited parallelism with constrained hardware resources. With model architecture optimization techniques like quantization and pruning, we can reduce the amount of hardware resources required and further enable other hardware optimizations. Lastly, a well-implemented software is also essential for high hardware utilization during runtime. Therefore, by combining hardware, model architecture and software optimizations, we can leverage limited hardware resources and efficiently parallelize our DNN models.

REFERENCES

[1] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "Fp-bnn: Binarized neural network on fpga," *Neurocomputing*, vol. 275, pp. 1072–1086, 2018.

[2] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, 2015, pp. 161–170.

[3] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA international symposium on field-programmable gate arrays*, 2016, pp. 26–35.

[4] Z. Li, Y. Zhang, J. Wang, and J. Lai, "A survey of fpga design for ai era," *Journal of Semiconductors*, vol. 41, no. 2, p. 021402, 2020.

[5] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[6] Y. Chen and M. Miao, "DNN SoC Design [ECE527]," 2023, university of Illinois at Urbana-Champaign.

[7] Y. Chen and M.Miao, "FPGA-Optimized Parallelism in Deep Neural Networks [CS533]," 2024, university of Illinois at Urbana-Champaign.

[8] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[9] "Sora model." [Online]. Available: https://openai.com/sora

[10] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A study of high-level synthesis: Promises and challenges," in *2011 9th IEEE International Conference on ASIC*, 2011, pp. 1102–1105.

[11] "Quantization explanation from wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Quantization_(signal_processing)

[12] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014, pp. 10–14.

[13] "Nvidia tensor cores." [Online]. Available: https://www.nvidia.com/en-us/data-center/tensor-cores/

[14] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," *CoRR*, vol. abs/1712.05877, 2017. [Online]. Available: http://arxiv.org/abs/1712.05877

[15] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," 2015.

[16] K. Kennedy and K. S. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 1993, pp. 301–320.

[17] A. Darte, "On the complexity of loop fusion," in *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00425)*. IEEE, 1999, pp. 149–157.