

# Accelerating Depthwise Separable Convolution for TinyML Applications

Lohith Artham

*Electrical and Computer Engineering  
Georgia Institute of Technology  
Atlanta, USA  
lartham3@gatech.edu*

Sumedh Ravi

*Electrical and Computer Engineering  
Georgia Institute of Technology  
Atlanta, USA  
sravi71@gatech.edu*

**Abstract**—Convolutional neural networks (CNNs) are widely used in the fields of computer vision and pattern recognition because of their high accuracy. However, state-of-the-art models have a large number of convolution layers which cause the network to be compute intensive and often require a powerful computing platform such as a Graphics Processing Unit. This makes it impractical to apply standard CNNs to portable and edge devices. State-of-the-art CNN architectures, such as MobileNet and Xception, adopt depthwise separable convolution layers to replace standard convolution layers, which significantly reduces operations and parameters with only limited loss in accuracy. With advancements in ultra-low-power tiny machine learning (TinyML) systems, MLPerf Tiny provides a benchmarking suite to properly evaluate these systems. In this project, we propose to optimize the performance of Tiny-ML benchmark models for two tasks keyword-spotting and visual wake-words for implementation on an FPGA and compare its performance with implementation on a standard CPU and with a standard convolution layer.

**Index Terms**—Convolutional neural network, depthwise separable convolution, FPGA, MobileNet, TinyML

## I. INTRODUCTION

Convolutional neural networks (CNNs) have been widely used in computer vision tasks, but they are computationally intensive, making neural network compression and dedicated hardware accelerators necessary for real-time processing and memory resources in embedded systems. MobileNet is a popular compact model that uses depthwise separable convolutions (DSCs) to reduce parameters and increase accuracy. Field Programmable Gate Arrays (FPGAs) are ideal platforms for CNN acceleration, with recent MobileNet accelerators falling into two categories: general convolution engines and dedicated computation engines [1]. However, there is still room for improvement in hardware design and accuracy loss in current accelerators [2].

This paper is organized as follows. Section II introduces the DSC, MobileNet and Prior work talking about the two popular DSC accelerator architectures. Section III presents the methodology and the timeline for the proposed work. Proposed design solutions and evaluation methods are given in Section IV and Section V respectively.

## II. BACKGROUND

### A. Depthwise Separable Convolution

A standard convolution operation consists of sliding a filter across the input image, computing the dot product between the filter and the corresponding input pixels, and producing a single output value. The number of filters used in the convolution operation is typically much smaller than the number of input channels, resulting in a high number of parameters and computations. This can be a limiting factor when designing deep neural networks that require large amounts of memory and computational power.

DSC, on the other hand, is a factorized convolution operation that separates the standard convolution operation into two stages. The first stage is a depthwise convolution that applies a single filter per input channel. This means that the number of filters used in the depthwise convolution is equal to the number of input channels, resulting in a significant reduction in the number of parameters and computations required. The second stage is a pointwise convolution that applies a  $1 \times 1$  filter to the output of the depthwise convolution. This operation allows the network to learn linear combinations of the output channels from the depthwise convolution and generate the final output.

One of the key advantages of DSC over standard convolution is the reduction in the number of parameters and computations required, making it a useful technique for designing smaller and more efficient neural networks. In addition, DSC can lead to better generalization due to its ability to capture more spatial information and reduce overfitting. “Fig. 1” shows a pictorial representation of the same [3].

Consider the size of input feature map is considered as  $W \times W \times M$ , the kernel size is  $K \times K$  and the output feature size is considered as  $W \times W \times N$ ,  $W$  is the spatial width and height of a square input feature map,  $M$  is the number of input channels,  $N$  is the number of output channels. In case of stride length of 1, the amount of operations is computed as [4]:

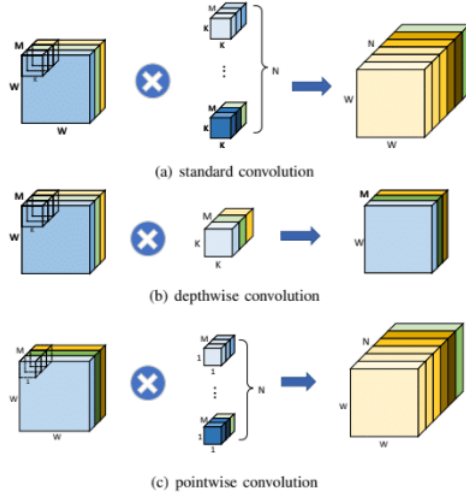


Fig. 1. Standard convolution and depthwise separable convolution

$$O_{STC} = W * W * K * K * M * N$$

$$O_{DSC} = W * W * K * K * M + W * W * M * N$$

$$F_O = \frac{O_{STC}}{O_{DSC}} = \frac{1}{N} + \frac{1}{K^2}$$

In fact, the reduction of parameters is also  $F_O$ .

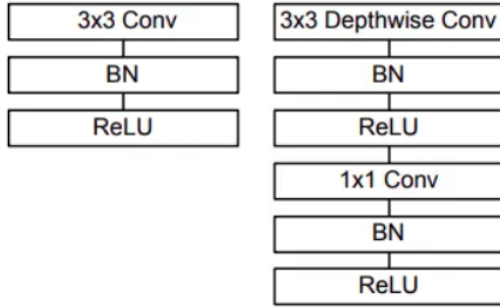


Fig. 2. Standard Convolution followed by batch normalization and RELU. Right: Depthwise convolution layer and pointwise convolution layer, each followed by batch normalization and RELU.

However, there are some limitations to DSC. One of the main limitations is that it may not capture certain spatial relationships between features that standard convolution can. This is because the depthwise convolution applies a single filter to each input channel, which may not be sufficient for capturing more complex feature interactions.

### B. MobileNet

MobileNet [5] is a lightweight convolutional neural network designed for mobile and embedded devices. It uses depthwise separable convolutions to reduce the number of parameters

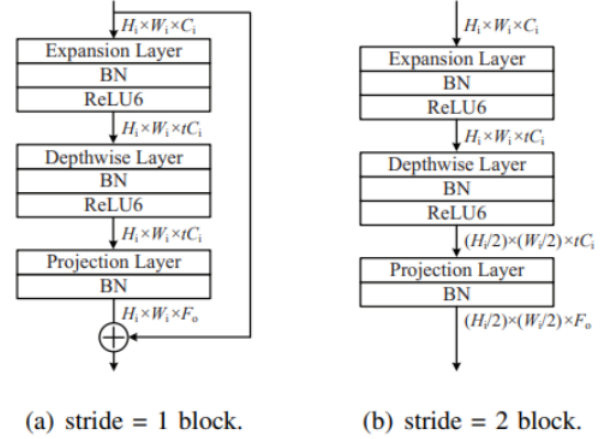


Fig. 3. Bottleneck block

and operations, a width multiplier to control the number of channels, and a global depthwise convolution to reduce spatial resolution. MobileNetV2 uses an inverted residual structure with bottleneck depthwise separable convolutions and residuals for improved accuracy and reduced computation compared to MobileNetV1. It has achieved state-of-the-art performance on various image classification tasks while maintaining a small model size and low computational cost. As shown in “Fig. 3”, the bottleneck block consists of an expansion layer, depthwise layer, and projection layer, with pointwise convolution and an expansion factor  $t$  controlling feature map dimensions. BN and ReLU6 activation are used after each layer, with a shortcut connection between the expansion and projection layers when stride is 1. No shortcut is added otherwise.

### C. Prior Work

DSC based lightweight CNN accelerator architectures can be roughly classified into two categories [6]:

- 1) Unified Engine Architecture(UE): treats DWC and PWC as independent convolutional layers, cannot utilize the potential to parallel DWC and PWC, read input feature maps from off-chip memory and write back output feature maps: High off-chip bandwidth
- 2) Separated Engine Architecture(SE): separate engines for DWC and PWC, DWC has separable channels so directly transferred to PWC, can be pipelined/parallelized, reduced off-chip bandwidth but more on-chip buffers to save the intermediate layers

Resource constraints in TinyML application may not allow to have enough area for separate dedicated engine kind of architecture. We can try working on a on-the-fly reconfigurable unified engine. The primary issue is that memory access traffic of intermediate feature maps is higher for DSC. To reduce the off-chip memory traffic, there is a need for a large amount of on-chip buffer. In the Separated

Engine version, ensuring high utilization and balanced computation is a challenge (non uniform computation of DWC and PWC)

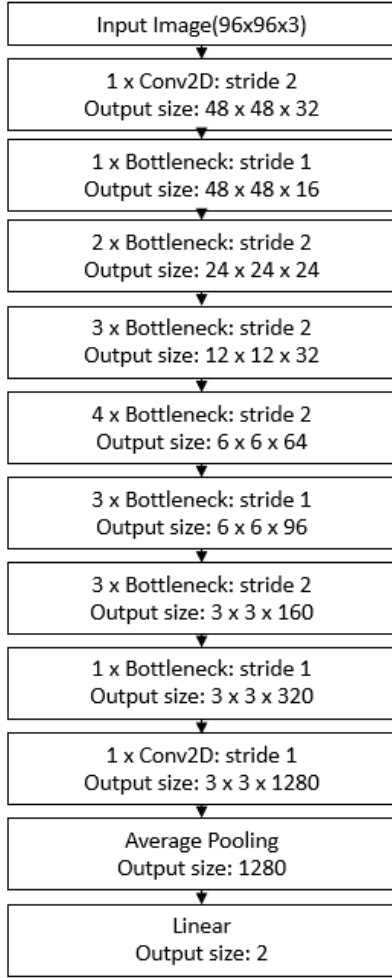


Fig. 4. MobileNet entire network

“Fig. 4” shows the MobileNet Architecture including the bottleneck layer.

### III. METHODOLOGY

The code framework for network acceleration will be implemented using Vitis HLS, while PyTorch will be utilized for golden files and profiling in order to comprehend the computation of different blocks. Our focus will not be on optimizing training in PyTorch since we require the files for comparison in HLS. The architecture and acceleration are independent of the model’s accuracy, thus it is prudent to not allocate time towards improving the model’s accuracy.

- ① DWC input feature maps. ② PWC output feature maps.  
③ DWC output & PWC input feature maps. ④ Weights of convolutional kernels.

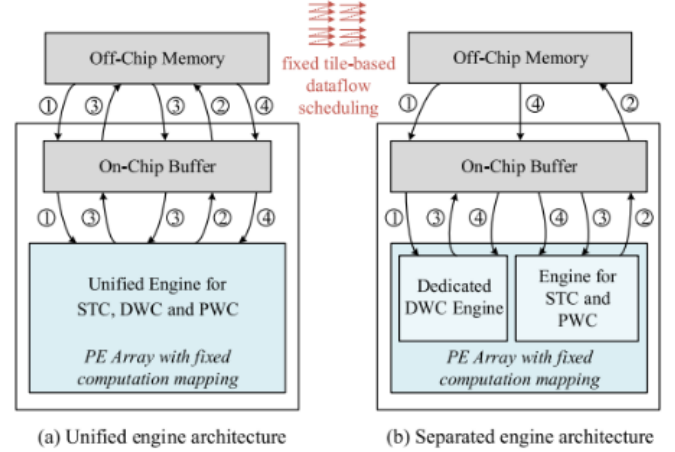


Fig. 5. Architecture diagrams of two existing designs for DSC-based lightweight CNNs

#### A. Python Model

MobileNet v2 is a popular deep learning model that is commonly used for computer vision tasks such as image classification, object detection, and segmentation. It is designed to be lightweight and efficient, making it well-suited for use in mobile and embedded systems.

In this scenario, the MobileNet v2 model was implemented on PyTorch using the TensorFlow implementation available on the TinyML benchmark resource for visual wake words dataset. This means that the PyTorch implementation was based on the original TensorFlow implementation, which was likely adapted to work with the PyTorch framework.

The primary focus of the training was not to achieve high accuracy, but rather to provide some results that could be used as a benchmark for the FPGA implementation. This suggests that the goal of the training was to obtain a model that was efficient enough to run on an FPGA and still produce reasonable results, rather than a model that achieved state-of-the-art accuracy on the dataset.

It is worth noting that accuracy is not the only metric that is important when evaluating a deep learning model. Other factors such as inference time, memory usage, and power consumption are also important considerations when deploying a model on a resource-constrained device such as an FPGA. By focusing on efficiency rather than accuracy, model was optimized for these other metrics, which could be more important in a real-world application.

#### B. Binary File Generation

In order to prepare the MobileNetv2 model for the TinyML benchmark, a Python script is used to extract the parameters

from the model. This involved loading the pre-trained MobileNetv2 model into memory using a deep learning framework such as PyTorch, and then extracting the weights and biases for each layer of the model. Once the parameters were extracted, they were converted to binary format, which is a more compact and efficient way of representing numerical data than text-based formats such as CSV or JSON.

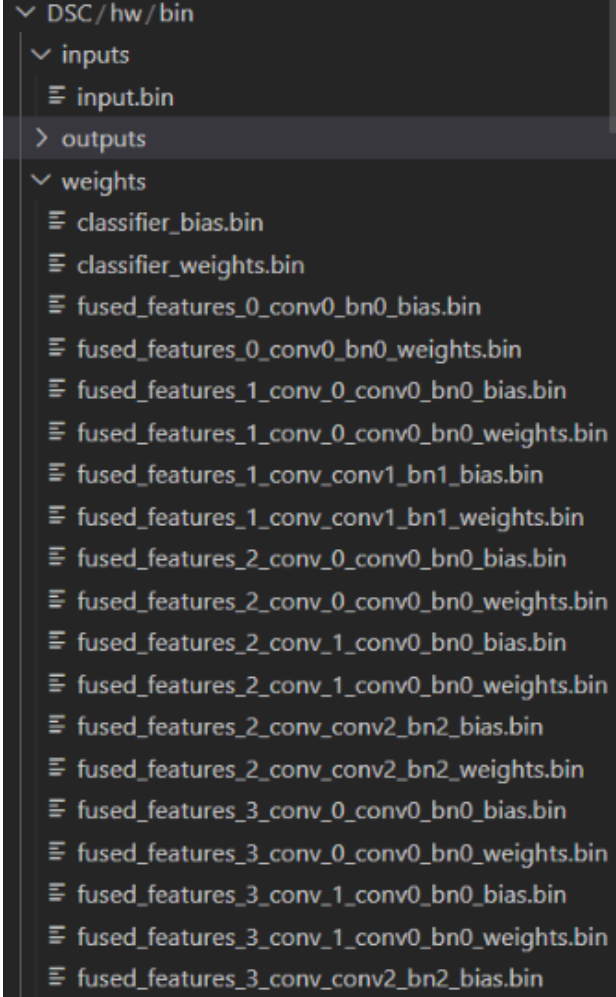


Fig. 6. Generated Binary Files

The binary files were then used to input the input features, weight and bias parameters for the High-Level Synthesis (HLS) code, which is a tool used for designing and optimizing hardware circuits. Output binary files were generated after each block of convolution, which allowed for easy debugging during individual block implementation in HLS. This involved running the HLS code and using the output binary files to verify that each convolution block was functioning correctly. Weights and bias binary files were generated for each convolution block, with the parameters from the convolution and batch normalization block fused to simplify the HLS implementation to only require the convolution block. This optimized the HLS code by fusing certain operations together to reduce the overall complexity of the design.

### C. Fusing Batch Normalization

Simplify the network architecture by merging the frozen batch normalization layer with a preceding convolution. This saves computational resources and simplifies the network architecture at the same time. We can replace these two layers by a single convolutional layer with the following parameters where  $W$  is filter weights and  $b$  is bias.

$$W = W_{BN} * W_{conv}$$

$$b = W_{BN} * b_{conv} + b_{BN}$$

### D. CModel

The implementation of MobileNetv2 in C++ was optimized for performance and resource utilization, as it serves as a baseline for future comparisons. To ensure synthesizability and greater control over the implementation and potentially optimize it for the specific hardware platform, each layer of the model was written without the use of external libraries. Parameterized functions for loading and storing buffers to DRAM were implemented to allow for varying sizes within the MobileNet layers and sub layers. The template method in C++ helped making the function parameterized. Additionally, the convolution function like conv2D, depthwise convolution and pointwise convolution were implemented to support different input and output channels by handling the boundary conditions while tiling the layers. This implementation was modular and reusable across varying input and output sizes. This could be especially useful for deploying the model on different hardware platforms or for different applications.

```
//-----
// Function to store output feature map tile block from BRAM to DRAM.
//-----
template <int out_channel, int inp_height, int inp_width>
void store_output_tile_to_DRAM {
    fm_t out_fm[out_channel][inp_height][inp_width],
    fm_t out_fm_buf[OUT_BUF_DEPTH][OUT_BUF_HEIGHT][OUT_BUF_WIDTH],
    int ti,
    int tj,
    int b,
    int stride
}
```

Fig. 7. Parameterized code using template

However, the use of fixed-size BRAM buffers for the convolution functions could limit the flexibility of the implementation in some scenarios. Pointwise and Depthwise convolution functions can do with different buffer sizes for further optimization but is not carried out for the time being. Overall, the C++ implementation of MobileNetv2 appears to be designed for efficiency, with custom code and parameterized functions contributing to its performance. However, the fixed-size BRAM buffers used for convolution could be a limitation in some cases.

---

```

(0): Conv2dNormActivation(
  (0): Conv2d(16, 96, kernel_size=(1, 1),
    stride=(1, 1))
  (1): BatchNorm2d(96, eps=1e-05,
    momentum=0.1, affine=True)
  (2): ReLU6(inplace=True)
)

(1): Conv2dNormActivation(
  (0): Conv2d(96, 96, kernel_size=(3, 3),
    stride=(2, 2), padding=(1, 1), groups=96)
  (1): BatchNorm2d(96, eps=1e-05,
    momentum=0.1, affine=True)
  (2): ReLU6(inplace=True)
)

(2): Conv2d(96, 24, kernel_size=(1, 1),
  stride=(1, 1))
(3): BatchNorm2d(24, eps=1e-05, momentum=0.1,
  affine=True)

```

---

The above bottleneck layer with a stride of two was a critical part of the MobileNetV2 model that we aimed to optimize. This layer consisted of a pointwise convolution with a ReLU6 activation function, followed by a depthwise convolution with ReLU6 and another pointwise convolution without ReLU6. Since the pointwise convolution had a kernel size of (1, 1), it could be implemented using four for loops. However, the depthwise convolution required five for loops since the input and output channels were equal. This made the depthwise convolution more complex to implement and potentially more computationally expensive. Overall, optimizing the bottleneck layer was an important objective of the project, given its critical role in the MobileNetV2 model.

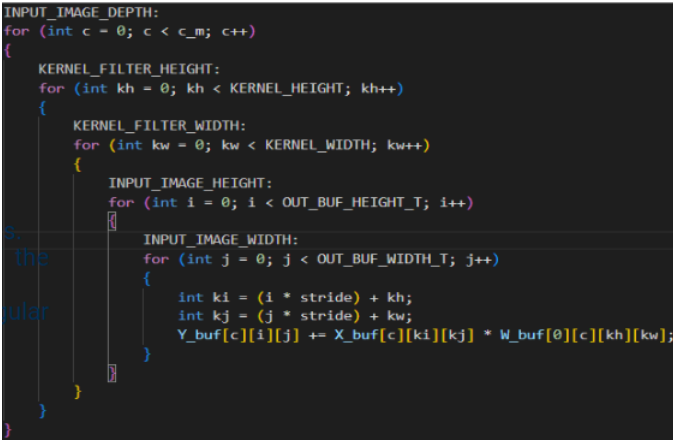
---

```

foreach output channel c_out do
  foreach pixel location (h, w) do
    foreach input channel c_in do
      output_tensor[c_out, h, w] +=
        input_tensor[c_in, h, w] *
        pointwise_kernel[c_in, c_out];

```

---



```

INPUT_IMAGE_DEPTH:
for (int c = 0; c < c_m; c++)
{
  KERNEL_FILTER_HEIGHT:
  for (int kh = 0; kh < KERNEL_HEIGHT; kh++)
  {
    KERNEL_FILTER_WIDTH:
    for (int kw = 0; kw < KERNEL_WIDTH; kw++)
    {
      INPUT_IMAGE_HEIGHT:
      for (int i = 0; i < OUT_BUF_HEIGHT_T; i++)
      {
        INPUT_IMAGE_WIDTH:
        for (int j = 0; j < OUT_BUF_WIDTH_T; j++)
        {
          int ki = (i * stride) + kh;
          int kj = (j * stride) + kw;
          Y_buf[c][i][j] += X_buf[c][ki][kj] * W_buf[0][c][kh][kw];
        }
      }
    }
  }
}

```

Fig. 8. Depthwise Convolution code

## E. HLS Code

Tiling-based convolution involves breaking down the convolution operation into smaller, more manageable pieces called tiles. Each tile is then processed independently, which can result in improved resource utilization and lower latency. By using defines for BRAM buffer sizes, the MobileNetV2 implementation could easily experiment with different tile sizes and adjust resource utilization and latency accordingly.

Loop unrolling is a technique that involves replicating loop iterations to reduce the overhead of the loop control logic. By doing this, the number of instructions executed per loop iteration is reduced, which can improve the overall performance of the code. In the convolution functions of MobileNetV2, loop unrolling was used to pipeline multiple MAC operations, which allowed for more efficient use of hardware resources and potentially faster and more accurate predictions.

Array partitioning is another optimization technique used in the convolution functions of MobileNetV2. It involves breaking down arrays into smaller, more manageable pieces, which can be processed in parallel. By doing this, the implementation can take advantage of parallel processing capabilities and potentially reduce latency.

## F. On-board FPGA Implementation

Optimized design is exported as Vitis HLS IP to Vivado to generate bitstream for Pynq-Z2 board identified by its part number xc7z020clg400-1. The host code is written in python to verify the functionality of the IP synthesized to access the device for computation (i.e. PS  $\rightarrow$  Host, PL  $\rightarrow$  Device). Pynq-Z2 runs a Linux OS on top of its PS and allows you to write Python programs in Jupyter environment. Steps involved are as follows:

- Imported Python packages
- Overlay the bitstream on the PL of Pynq-Z2
- Created Pynq buffers for IO ports in Host
- Input feature map and weight parameters loaded from binary files
- Initialised control registers in PL with the pointers created for IO ports
- Ran the IP and verified output for functional correctness

## IV. EVALUATION

Internet of Things (IoT) applications requires intelligence on the edge but have extremely limited on-chip memory and compute capability. We need tiny vision models that fit within a few hundred kilobytes of memory footprint. Visual Wake Words [7] dataset from the TinyML benchmarks are used for running our experiments. The dataset is a binary classification model to identify whether a person is present or not, and provides a realistic benchmark for tiny vision model. Metrics like speedup will be compared across the runtimes seen on CPU for the same network.

## V. RESULTS

Mean Squared Error (MSE) was used as a metric for evaluating functional correctness. MSE is a common loss function used in machine learning to measure the average squared difference between the predicted and actual values, and a lower MSE value indicates better prediction accuracy. For the MobileNetv2 implementation, the objective was to achieve an MSE value of less than  $10^{-10}$ , which would indicate a high level of accuracy in the model's predictions. We achieved an MSE value in the order of  $10^{-11}$  for the HLS implementation. This is especially important in real-time applications, where high accuracy is necessary to ensure that decisions made by the model are reliable and consistent.

| Resource Utilization |      |     |        |       |
|----------------------|------|-----|--------|-------|
| Name                 | BRAM | DSP | FF     | LUT   |
| Available            | 280  | 220 | 106400 | 53200 |
| Utilized             | 116  | 98  | 16812  | 50698 |
| Utilization (%)      | 41   | 44  | 15     | 95    |

Besides LUT utilization (95%) the BRAM, FF, DSP utilization is low. LUT utilization is one other limitation for DSC Inference. Reusing the compute blocks requires loading parameters from different layers resulting in large number of multiplexers. Low resource utilization is credited to the tiled convolution and matrix-vector multiplication.

| Latency                       |           |         |
|-------------------------------|-----------|---------|
| Device                        | Time (ms) | Speedup |
| Unoptimized (w/o Pipelining)  | 1468      | 1x      |
| Unoptimized (with Pipelining) | 210       | 7x      |
| Optimized                     | 36        | 40x     |
| on-board FPGA                 | 92        | 16x     |

The difference in latency for optimized HLS code from the tool and on-board FPGA implementation is due to incorrect modelling of DRAM latency by the tool.

## VI. FUTURE WORK

The current implementation involves frequent DRAM accesses for reading parameters and storing intermediary results. In the bottleneck stride two layer, for example, there are six read accesses for parameters and intermediary input features, and three write accesses to store intermediary output features. While this approach is functional, there are many intermediate DRAM accesses that could potentially impact performance and limit achievable speedups. Therefore, optimizing DRAM access to reduce latency and improve performance is an important consideration for achieving higher speedups in the future.

Another area of focus is to improve the performance of the implementation to beat the time taken by the python implementation on Google Collaborator, which utilizes multi-

cores and GPUs. This would require further optimization of the HLS implementation. Furthermore, we could investigate the use of pruning and quantization techniques to reduce the size of the model and potentially improve performance.

## VII. CONCLUSION

The objective of this project was to accelerate Depthwise Separable Convolution (DSC) for inference in TinyML applications. To achieve this, bottleneck layers were implemented and optimized to a certain extent. Replicating these bottleneck layers would complete the MobileNet model, resulting in an efficient and accurate model for TinyML applications. With the use of various optimization techniques, such as tiling-based convolution, loop unrolling, and array partitioning, we achieved high levels of accuracy with reasonable speedup. Overall, the MobileNetv2 implementation shows great potential for accelerating DSC in TinyML applications.

## VIII. ACKNOWLEDGMENTS

We express our gratitude to Ashwin Bhat for his valuable discussions and guidance throughout the process. We thank Cong Hao and Akshay Kamath for very helpful discussions, and thank the anonymous reviewers for their feedback and insights during the process.

## REFERENCES

- [1] S. Yan, Z. Liu, Y. Wang, C. Zeng, Q. Liu, B. Cheng, and R. C. Cheung, "An fpga-based mobilenet accelerator considering network structure characteristics," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2021, pp. 17–23.
- [2] J. Knapheide, B. Stabernack, and M. Kuhnke, "A high throughput mobilenetv2 fpga implementation based on a flexible architecture for depthwise separable convolution," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2020, pp. 277–283.
- [3] D. Wu, Y. Zhang, X. Jia, L. Tian, T. Li, L. Sui, D. Xie, and Y. Shan, "A high-performance cnn processor based on fpga for mobilenets," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 136–143.
- [4] L. Bai, Y. Zhao, and X. Huang, "A cnn accelerator on fpga using depthwise separable convolution," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 10, pp. 1415–1419, 2018.
- [5] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.
- [6] B. Li, H. Wang, X. Zhang, J. Ren, L. Liu, H. Sun, and N. Zheng, "Dynamic dataflow scheduling and computation mapping techniques for efficient depthwise separable convolution acceleration," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 68, no. 8, pp. 3279–3292, 2021.
- [7] A. Chowdhery, P. Warden, J. Shlens, A. Howard, and R. Rhodes, "Visual wake words dataset," *arXiv preprint arXiv:1906.05721*, 2019.