FPGA Accelerator for MLP-Mixer

Stefan Abi-Karam
Georgia Institute of Technology
stefanabikaram@gatech.edu

***Abstract*—Abstract**

I. Introduction

New deep learning architectures for vision tasks have emerged in the past decade since the rise of convolutional neural networks. These innovations include attention with the introduction of the vision transformer and MLP-Mixer and numerous MLP-Mixer-like architectures.

These new vision architectures are attractive for targeting FPGA hardware acceleration in the application of real-time inference. Some of these applications include autonomous driving and autonomous industrial robotics. In these cases, customized models can be deployed on FPGAs an updated over time as more vision data is collected and models are fine-tuned over time.

The MLP-Mixer model is an attractive target since most operations can be formulated as linear operations using GEMM in conjunction with element-wise operations. This allows the hardware need much simpler to implement or to be mapped to existing simpler systolic-based accelerators.

II. Related Work

The original MLP-Mixer architecture [1] proposed a similar structure to the Vision Transformer [2] with multi-headed attention replaced with an Spatial MLP as the token mixer. Since then a whole class of "metaformers" or "MLP-Mixer-like" architectures have been introduced in the literature which follow the mixing block structure of ViT and MLP-Mixer. The original Metaformer work [3] proposed the use of a pooling for token mixer. However there are many other variations on this theme which are comprehensively covered as best as possible in [4]. It is also important to note the work of [5] which revisits CNNs in the age of transformers and provides a though comparison of a rearchitected improved ResNet architecture and STOA vision transformers.

Currently, [6] is the only work exploring the acceleration of these new architectures focusing on the co-design of vision transformers for deployment on an FPGA. Together with the use of quantization and co-design they can archive significantly better FPS/W compared to a GPU or CPU with their best implementation achieving 31.6 FPS.

The HyperMixer [7] is another interesting work in the area of optimized MLP-Mixer-like inference in the name of "Green AI". HyperMixer aims to incorporate similar inductive biases as Transformers by introducing a hypernetwrok to predict the weights of the token mixing MLP in the mixer block.

III. MLP-Mixer Accelerator

A. High-Level Accelerator Architecture

The overall hardware architecture follows a similar structure to the MLP-Mixer architecture as shown in Figure 1. The hardware kernel performs the following operations sequentially:

1) Load model parameters if flag is passed into the kernel
2) Load input image
3) Compute patch embeddings
4) For N blocks
   - Compute mixing block
5) Compute global average polling
6) Compute linear classification head

The kernel contains internal memory buffers implemented as BRAM to store the input image as well as model parameters locally within the kernel. On each kernel call, these inputs are loaded from of-chip, such as high-bandwidth memory (HBM), and directly copied into the local BRAM.

The intermediate outputs from each computation are also stored on local memory buffers implemented as BRAM. However for the mixer blocks, we use a double buffer to reduce the amount of local memory needed. The mixer block function is called with the input an output buffers swapped for even and odd values of the current block N.

B. Patch Embedding

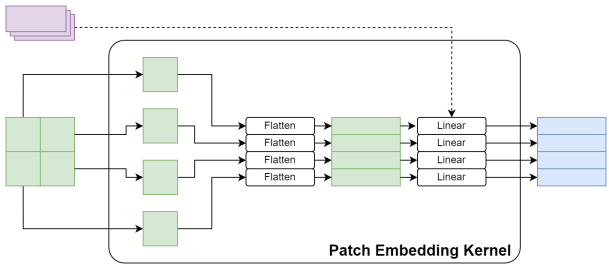

Fig. 2. Patch embedding architecture

The patch embedding kernel flattens patches from the input buffer using some indexing arithmetic and applies a linear transformation to embed each patch into the patch embedding array. The linear layer is implemented using a block parallel linear layer also used thought the accelerator.

C. Mixer Block

The mixer block computes the token-mixing and channel-mixing operations with the addition of layer normalization,

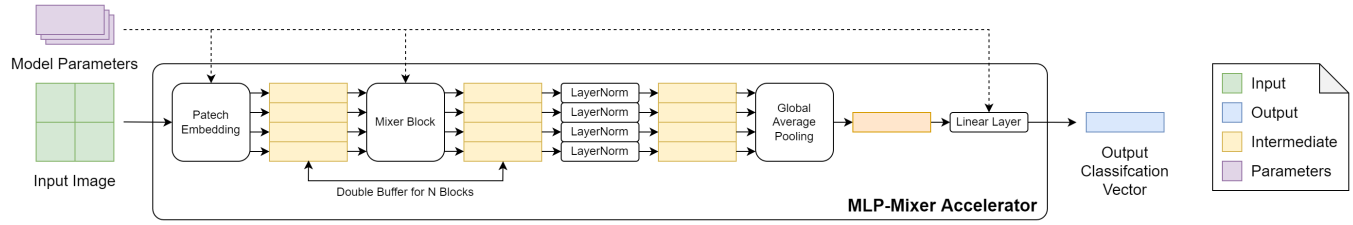

Fig. 1. Overall accelerator architecture

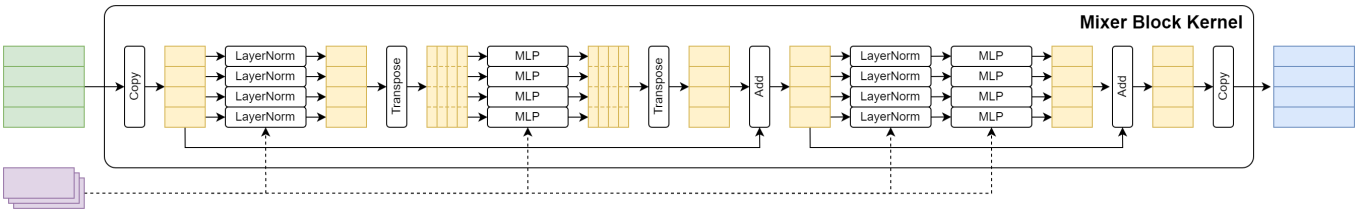

Fig. 3. Mixer block architecture

skip connections, and transpose operations. The MLP and layer normalization operations are discussed in more detail in other sections. The whole mixer block kernel region has a dataflow pragma applied to optimize data movement such as with the skip connections, transposition, and coping of data between internal buffers. The whole mixer block function is also implemented as a template functions to allow for different sizes of embeddings as well as different parallel factors for the layer norm and MLP kernels.

D. MLP

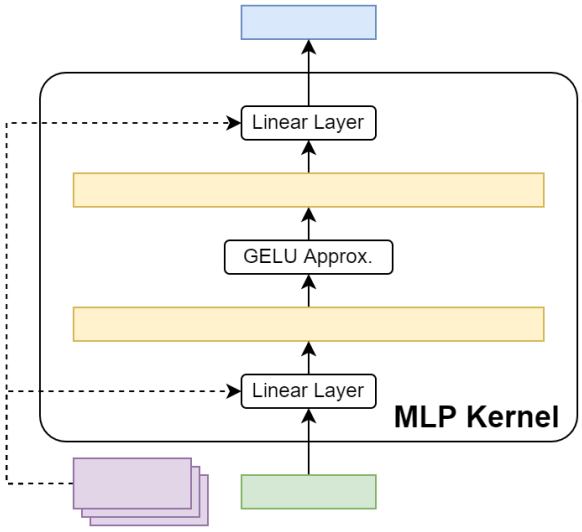

Fig. 4. MLP architecture

The MLP kernel follows a 2-layer implementation from the paper. The hidden dimension expands the input into a bigger size and contacts back down to the input size. Linear layers are a block parallel implementation with two parallelization factors for the input and output block partition size.

E. Layer Normalization

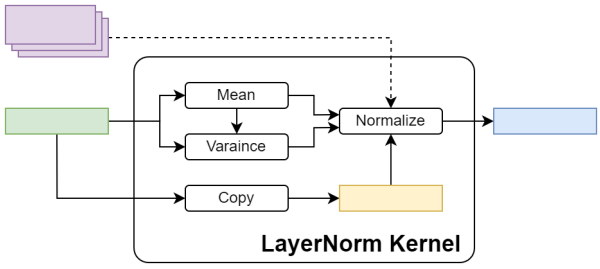

Fig. 5. Layer normalization architecture

The layer normalization kernel follows a 3-pass calculation, one for the mean, one for the variance, and one for the normalization. This is implemented using block parallel approach like the linear layer with a single parallelization factor for the vector block partition.

F. Block-Parallel Linear Layer

The following is the C++ HLS code for the block-parallel linear layer.

```
template <const int in_size, const int out_size,
↪  const int BLOCK_SIZE_IN_ = 1, const int
↪  BLOCK_SIZE_OUT_ = 1>
void linear(F_TYPE input[in_size], F_TYPE
↪  output[out_size], F_TYPE
↪  weight[out_size][in_size], F_TYPE
↪  bias[out_size]) {

  #pragma HLS INLINE off

  const int BLOCK_SIZE_OUT = BLOCK_SIZE_OUT_;
```

```
  const int BLOCK_SIZE_IN = BLOCK_SIZE_IN_;

  #pragma HLS array_reshape variable = input cyclic
  ↪  factor = BLOCK_SIZE_IN dim = 1
  #pragma HLS array_reshape variable = output
  ↪  cyclic factor = BLOCK_SIZE_OUT dim = 1

  #pragma HLS array_reshape variable = weight
  ↪  cyclic factor = BLOCK_SIZE_OUT dim = 1
  #pragma HLS array_reshape variable = weight
  ↪  cyclic factor = BLOCK_SIZE_IN dim = 2

  #pragma HLS array_reshape variable = bias cyclic
  ↪  factor = BLOCK_SIZE_OUT dim = 1

  F_TYPE temp_sum[BLOCK_SIZE_OUT];

  #pragma HLS ARRAY_PARTITION variable = temp_sum
  ↪  complete

  for (int i = 0; i < out_size; i +=
  ↪  BLOCK_SIZE_OUT) {
    for (int j = 0; j < in_size; j +=
    ↪  BLOCK_SIZE_IN) {
      #pragma HLS PIPELINE

      for (int k = 0; k < BLOCK_SIZE_OUT; k++) {
        temp_sum[k] = 0;
      }

      for (int k = 0; k < BLOCK_SIZE_OUT; k++) {
        SUM_INNER: for (int l = 0; l <
        ↪  BLOCK_SIZE_IN; l++) {
          temp_sum[k] += weight[i + k][j + l] *
          ↪  input[j + l];
        }
      }

      for (int k = 0; k < BLOCK_SIZE_OUT; k++) {
        if (j == 0) {
          output[i + k] = bias[i + k];
        }

        output[i + k] += temp_sum[k];
      }
    }
  }
}
```

This implementation partitions the input array based on the input partition size, the output array based on the output partition size, the weight array based on the input and output partition size, and the bias array based on the output size. This allows for parallel multiply-accumulate computations of partial outputs. The level of parallelization is directly controlled by the BLOCK_SIZE_IN and BLOCK_SIZE_OUT template parameters.

G. Block-Parallel Layer Normalization

The following is the C++ HLS code for the block-parallel layer normalization layer.

```
template <int SIZE, const int BLOCK_SIZE_=1>
void layer_norm(F_TYPE x_in[SIZE], F_TYPE
↪  x_out[SIZE], W_TYPE gamma[SIZE], W_TYPE
↪  beta[SIZE]) {

  #pragma HLS INLINE off

  const int BLOCK_SIZE = BLOCK_SIZE_;
  #pragma HLS ARRAY_PARTITION variable = x_in
  ↪  cyclic factor = BLOCK_SIZE
  #pragma HLS ARRAY_PARTITION variable = x_out
  ↪  cyclic factor = BLOCK_SIZE
  #pragma HLS ARRAY_PARTITION variable = gamma
  ↪  cyclic factor = BLOCK_SIZE
  #pragma HLS ARRAY_PARTITION variable = beta
  ↪  cyclic factor = BLOCK_SIZE

  const F_TYPE eps = 1e-5;

  F_TYPE x_sum = 0;
  for (int i = 0; i < SIZE; i += BLOCK_SIZE) {
    #pragma HLS PIPELINE
    for (int j = 0; j < BLOCK_SIZE; j++) {
      x_sum += x_in[i + j];
    }
  }
  F_TYPE mean = x_sum / SIZE;

  F_TYPE var_sum = 0;
  for (int i = 0; i < SIZE; i += BLOCK_SIZE) {
    #pragma HLS PIPELINE
    for (int j = 0; j < BLOCK_SIZE; j++) {
      F_TYPE x_in_buf = x_in[i + j];
      F_TYPE x_shifted = x_in_buf - mean;
      F_TYPE x_shifted_sq = x_shifted * x_shifted;
      var_sum += x_shifted_sq;
    }
  }
  F_TYPE var = var_sum / SIZE;

  for (int i = 0; i < SIZE; i += BLOCK_SIZE) {
    #pragma HLS PIPELINE
    for (int j = 0; j < BLOCK_SIZE; j++) {
      F_TYPE x_in_buf = x_in[i + j];
      F_TYPE x_out_buf;
      x_out_buf = (x_in_buf - mean) / sqrt(var
      ↪  +eps);
      x_out_buf = (x_out_buf * gamma[i + j]) +
      ↪  beta[i + j];
      x_out[i + j] = x_out_buf;
    }
  }
}
```

This implementation uses a three pass approach. The first loop computes the mean, the second loop computes the variance which depends on the mean, and the third loop computes the normalization which depends on the mean, variance, and model parameters. The input, output, and model parameters are all partitioned by the same factor. This allows for blocks of the input to be used to compute partial sums used for computing the mean and variance and for blocks of the output to be computed at once. The level of parallelization is directly controlled by the BLOCK_SIZE template parameter.

H. Gaussian Error Linear Units (GELU) Activation

The GELU activation [8] is precisely defined as the following.

$$\mathrm{GELU}(x) = x \cdot \frac{1}{2}[1 + \mathrm{erf}(x/\sqrt{2})] \tag{1}$$

However it can be approximated using either of the following approximations:

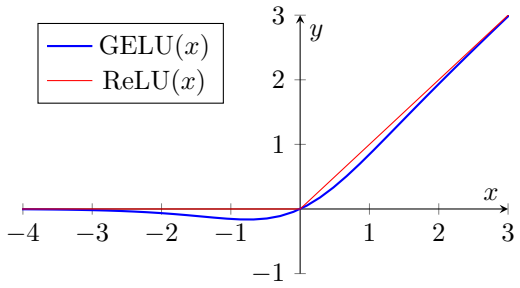

Fig. 6. GELU Activation

$$0.5x\left(1+\tanh\left[\sqrt{2/\pi}\left(x+0.044715x^3\right)\right]\right) \quad (2)$$

or

$$x\sigma(1.702x) \quad (3)$$

For inference, equation 3 has the lowest latency and lowest DSP usage compared to equation 2 and equation 1. However it incurs the greater computational error compared to the the equation 2. This error is minimal in affecting the predicted output when performing fixed-point arithmetic. Therefore, in our final implementation, the approximation in equation 3 is used.

IV. Implementation

We implemented the S/32 variant of the MLP-Mixer architecture proposed in the original MLP-Mixer work [1]. The model S/32 configuration is shown in Table I.

TABLE I
MODEL CONFIGURATION FOR THE S/32 VARIANT OF THE MLP-MIXER ARCHITECTURE

Specification	S/32
Number of layers	8
Patch resolution $P \times P$	32×32
Hidden size C	512
Sequence length S	49
MLP dimension D_C	2048
MLP dimension D_S	256
Parameters (M)	19

We targeted the Xilinx Alveo U50 FPGA accletaor card using High-Level Synthesis (HLS) by Vitis HLS and Vivado. The available resources of U50 is shown in Table III, and the FPGA logic targets 300 MHz clock frequency.

For our implementation, we used fixed point arithmetic with 16 bits of fractional precision, 15 bits of integer precision, and 1 bit for the sign for a total 32-bit fixed-point datatype.

We also employ the following parallelization factors in our model implementation.

- Patch Embedding Linear
  - BLOCK_SIZE_in = 8
  - BLOCK_SIZE_out = 8
- Pre-Head Layer Norm
  - BLOCK_SIZE = 8
- Classification Head Linear
  - BLOCK_SIZE_in = 8
  - BLOCK_SIZE_out = 8
- Mixer Block Layer Norm 1
  - BLOCK_SIZE = 32
- Mixer Block MLP 1
  - BLOCK_SIZE_in = 7
  - BLOCK_SIZE_out = 32
- Mixer Block Layer Norm 2
  - BLOCK_SIZE = 32
- Mixer Block MLP 2
  - BLOCK_SIZE_in = 32
  - BLOCK_SIZE_out = 32

V. Results

After HLS synthesis we achieved a total latency of 23.09 ms. which corresponds to an 43.30 FPS. This achieved FPS is more than enough for autonomous driving at highways speeds.

TABLE II
IMPLEMENTED MODEL PERFORMANCE

Implementation	Latency (ms.)	FPS
CPU	17.33	57.70
GPU	6.29	158.98
FPGA	23.09	43.30

We also show the resource utilization of our model in Table III. We exceed the resource usage for DSP and LUTs. However, this can be easily fixed by tweaking the parallelization factors.

TABLE III
RESOURCE UTILIZATION ON XILINX ALVEO U50 FPGA. THE CLOCK FREQUENCY IS 300 MHZ.

Model	DSP	LUT	FF	BRAM
Available	5952	872K	1,743K	2,688
S/32	8,031	950k	1,577k	1,794

VI. Conclusion

We have demonstrated the first FPGA hardware accelerator for the MLP-Mixer architecture and provided inference performance comparisons with CPU and GPU. Even though the latency is slower than CPU or GPU inference, we predict that the our model performs better in terms of energy when looking FPS/Watt. We hope to expand out work to explore and support other MLP-Mixer-like architectures through modular HLS framework. We also hope to explore hardware design space exploration, hardware-software co-design, and quantization to further increase the performance of our model.

REFERENCES

[1] I. Tolstikhin, N. Houlsby, A. Kolesnikov, L. Beyer, X. Zhai, T. Unterthiner, J. Yung, A. Steiner, D. Keysers, J. Uszkoreit, M. Lucic, and A. Dosovitskiy, "MLP-Mixer: An all-MLP Architecture for Vision," *arXiv:2105.01601 [cs]*, Jun. 2021, arXiv: 2105.01601. [Online]. Available: http://arxiv.org/abs/2105.01601

[2] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," *arXiv:2010.11929 [cs]*, Jun. 2021, arXiv: 2010.11929. [Online]. Available: http://arxiv.org/abs/2010.11929

[3] W. Yu, M. Luo, P. Zhou, C. Si, Y. Zhou, X. Wang, J. Feng, and S. Yan, "MetaFormer is Actually What You Need for Vision," *arXiv:2111.11418 [cs]*, Nov. 2021, arXiv: 2111.11418. [Online]. Available: http://arxiv.org/abs/2111.11418

[4] R. Liu, Y. Li, L. Tao, D. Liang, S.-M. Hu, and H.-T. Zheng, "Are we ready for a new paradigm shift? A Survey on Visual Deep MLP," *arXiv:2111.04060 [cs]*, Mar. 2022, arXiv: 2111.04060. [Online]. Available: http://arxiv.org/abs/2111.04060

[5] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A ConvNet for the 2020s," *arXiv:2201.03545 [cs]*, Mar. 2022, arXiv: 2201.03545. [Online]. Available: http://arxiv.org/abs/2201.03545

[6] M. Sun, H. Ma, G. Kang, Y. Jiang, T. Chen, X. Ma, Z. Wang, and Y. Wang, "VAQF: Fully Automatic Software-Hardware Co-Design Framework for Low-Bit Vision Transformer," *arXiv:2201.06618 [cs]*, Feb. 2022, arXiv: 2201.06618. [Online]. Available: http://arxiv.org/abs/2201.06618

[7] F. Mai, A. Pannatier, F. Fehr, H. Chen, F. Marelli, F. Fleuret, and J. Henderson, "HyperMixer: An MLP-based Green AI Alternative to Transformers," *arXiv:2203.03691 [cs]*, Mar. 2022, arXiv: 2203.03691. [Online]. Available: http://arxiv.org/abs/2203.03691

[8] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," 2016. [Online]. Available: https://arxiv.org/abs/1606.08415