Efficient Vision Transformer Inference using CPU+FPGA

Eric Dong, Hongyi Sun, Wenyun Wang, and Sebastian Pereira

May 6, 2023

Abstract

This paper presents a novel approach to efficiently perform inference using Vision Transformers (ViT) on a CPU and FPGA co-processing architecture. Vision Transformers have shown significant improvements in image recognition tasks but come with high computational complexity, making them expensive to train and deploy. To address this challenge, we implement parallelization using OpenMP for computation-level parallelism and utilize a Field Programmable Gate Array (FPGA) to optimize matrix multiplication operations within the Vision Transformer. Our approach focuses on accelerating the multi-head attention mechanism and patch and positional encoding of the transformer. We also integrate an FPGA for efficient matrix multiplication and analyze the results and performance of our proposed CPU+FPGA architecture. Experiments on the MNIST dataset demonstrate the effectiveness of our method and its potential to improve the performance of Vision Transformers in real-time computer vision applications.

1 Background and Significance

The Problem Vision transformers are large models that are typically packed with a lot of parameters. Because of this, for them to be fully taken advantage of they perform best when trained on huge sums of data. Lots of parameters and lots of training data lead to the training and deployment of vision transformers being expensive. The cost makes this a particularly appropriate problem to solve. With the widespread adoption of image recognition and a variety of other image-based artificial intelligence infrastructure, speeding up training and deployment of vision transformers is something that is of utmost importance.

Vision Transformers To fully understand the extent to which parallelization is possible it is essential to understand every aspect of vision transformers and how they work. Instead of using convolutional layers as is done in traditional CNNs, vision transformers first divide an image into fixed-size non-overlapping patches. These patches are then embedded into a high-dimensional space. After a positional encoding is added that encodes the location of each patch within the image, the data is ready to be fed through the actual transformer encoder which has multiple self-attention and feed-forward layers. After this, the image can be classified. The result is a model that learns to capture complex patterns and relationships between the patches, allowing for optimal image classification, as shown in figure 1 [1].

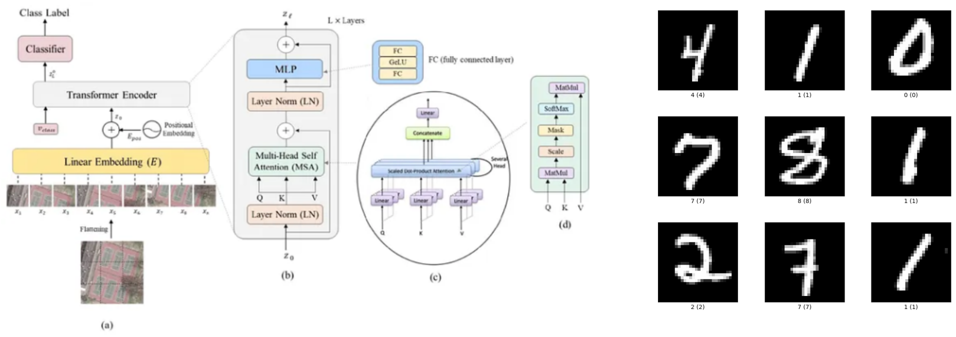


Figure 1: Vision Transformer Introduction. Left: vision transformer algorithm flow. Right: MNIST dataset example

Dataset

Due to the nature of the project and class being focused on parallel computing we wanted to work with a relatively simple dataset and try really focus on optimizing time and speed. For that reason, we chose the MNIST data set as it is easily accessible and widely used. The MNIST dataset is a large collection of 70,000 handwritten digits (0-9) split into 60,000 training and 10,000 testing images. It is a popular benchmark for machine learning algorithms and image recognition tasks because it's relatively simple yet also diverse, as shown in figure 1.

Research Methods

The research methods included a variety of different parallelization implementations. We used OpenMP for CPU-level parallelism. We also implemented our own Field Programmable Gate Array (FPGA) to optimize the system. The FPGA is a reprogrammable chip capable of emulating any hardware behavior. In this paper, we implemented a tiny TPU capable of accelerating matrix multiplication. GPUs may be fast and able to implement millions of calculations at once, but they are not dynamic like FPGAs. For this reason, FPGAs have been used in industry in fields from artificial intelligence to finance to create reconfigurable hardware targetting specialized tasks. For our sake, we programmed our FPGA to focus on the matrix multiplication aspects of a vision transformer. The results have been interesting and encouraging. The significance here once again cannot be understated. These models are huge and costly and there are clear benefits to speeding them up. This has been reflective of recent research trying to tackle similar problems.

Literature Review

In a paper titled "Parallel and Fast Vision Transformer for Offline Handwritten Chinese Character Recognition" researchers tried to speed up vision transformers with parallelization. They experimented with many different variations of vision transformers such as a two-way parallel ViT, four-way parallel ViT, and seven-way parallel ViT, and compared the results. The two-way parallel model has shown a significant reduction in computational complexity compared to the ViT model. With 50.1 percent fewer parameters and 34.6 percent fewer FLOPs, this model has indirectly improved image recognition speed [2]. In addition to this study, a paper titled "Accelerating Training of Transformer-Based Language Models with Progressive Layer Dropping" worked on language transformer acceleration. Through progressive layer dropping (PLD), scheduled stochastic dropping of layers based on a parameter, they reduced the training time of the BERT language model by 24 percent on Wikipedia corpus data$\LaTeX$[3]. There have also been numerous posts on GitHub and Hugging Face describing different methods and actually implementing parallelized vision transformers. There are a variety of different methodologies but the wealth of robust projects that already exist in the space exemplifies its importance and value.

2 Scientific Goals and Objectives

Our overarching goal is to perform extensive experimentation and analysis to compare the performance of our proposed Vision Transformer architecture on CPU+FPGA against a normally operated Vision Transformer. In terms of scientific goals, we want to focus our efforts on the multi-head attention module and patch and positional encoding. Another goal is to successfully handle communication latencies due to the implementation of the FPGA. We hope to achieve an adequate speedup and will have a variety of different benchmarking comparisons such as using only a CPU and a hybrid between the CPU and FPGA for different parts of the problem. It was also an objective of our project to successfully program and apply the FPGA for GeMM to our architecture. The final setup is shown below.

We want to successfully parallelize a vision transformer to combat the variety of bottlenecks and data-intensive problems they face while doing prediction tasks. For instance, vision transformers complexity is $O(n^2d)$ which means complexity scales quadratically with input length. This is just one example of vision transformers being especially expensive and computationally taxing. Vision transformers are especially relevant in recent computer vision developments at the cutting edge such as autonomous driving, surveillance, and augmented reality. All of these application areas require and depend on a large number of inferences to be performed per second, which is difficult for traditional computer vision algorithms that use CPU or GPU-only architectures. The need for compute hours on an HPC for our vision transformer project is justified by the growing demand for real-time computer vision applications

3 Algorithms and Code Parallelization

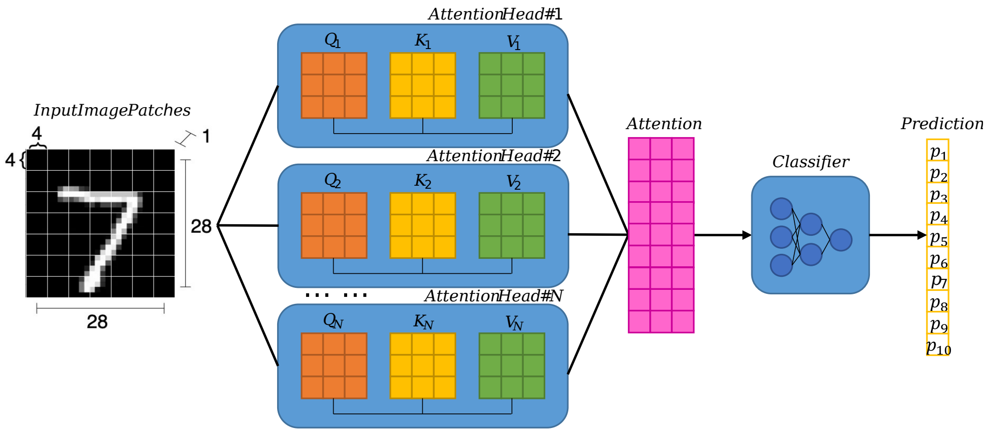


Figure 2: Overall view of the vision transformer.

General Algorithm

First of all, we work with the MNIST dataset, which consists of images of dimension $(1, 28, 28)$. Suppose we have a batch size of N, then each batch of input to the model will be $(N, 1, 28, 28)$.

One forward pass of our implementation for the vision transformer is as follows: 1. We start by patchifying the input image, where we patch the image into patches of $(4, 4)$, which means the output will be $(N, 49, 4, 4)$. 2. Then we linearly embed each patch, essentially passing these patches through a linear layer that embeds the $(4, 4)$ patch into a vector of size d, where d is the hidden dimension hyperparameter set by the user. 3. We then positionally embed each patch with there relative position, each patch is given another position number

$$p_{i,j} = \begin{cases} \sin \frac{i}{(\frac{j}{d})^{10000}} & j\%2 = 0 \\ \cos \frac{i}{(\frac{j-1}{d})^{10000}} & j\%2 \neq 0 \end{cases}$$

4. After all the embedding steps, we input the patch of size $N, 50, d$ (50 because we add 1 positional embedding) to a ViT block, which consists of a self-attention mechanism and outputs a single attention matrix across all attention heads. 5. Lastly the attention matrix goes through the last few linear classification layers and outputs a vector of size 10 which is the number of classes the MNIST dataset has.
Self-Attention Suppose we have n_{head} heads, each head will be responsible for $d_{head} = \frac{d}{n_{head}}$ elements from the embedded patches. We run inference on 3 linear layers to get 3 tensors of size $(N, 50, d_{head})$ we will get 3 tensors which we will denote as Query(Q), Key(K), and Value(V). We then calculate the attention matrix $A_i = \texttt{Softmax}(\frac{Q_i K_i^T}{\sqrt{d_{head}}})V_i$ for each input i in a batch of size N. Each of these operations is called a **scaled-dot product**.

CPU only parallelism

For the CPU-only implementation without the FPGA, we employ OpenMP for shared memory parallelism.

Parallelized muti-headed attention

The first step is to parallelize the multi-headed attention mechanism. This is accomplished by assigning each thread to a smaller, independent attention head 3, each of which operates on the same set of input data, but learns and calculates the attention differently. Each thread is responsible for its own Q, K, V inference and the corresponding scaled-dot product calculation. In the end, the final product of the scaled-dot product operation is written into the final big attention matrix and is stacked together after all attention heads complete their calculation.

Parallelized Q, K, V matrix inference

Inside each head, we can also parallelize the Q, K, V inference. Since running inference on different threads has no dependencies, we can assign three threads to perform inference of these three matrices in parallel (one thread for each of the Q, K, V). Moreover, since each of these matrices is of the same size (N, l, d_{head}), the inference took approximately the same amount of time and will not cause load imbalance on any thread. After the inference of all Q, K, V matrices, we join those threads and start performing scaled-dot products using those matrices.

Vectorized parallel fused-transpose matrix multiplication

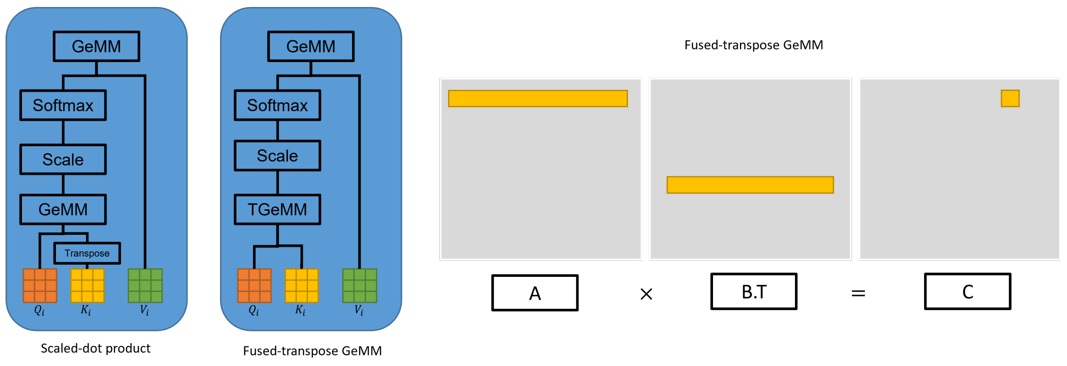


Figure 3: Fused-transpose matmul algorithm

Instead of doing Transpose K then QK^T, we can specifically design a fused transpose-matrix multiplication method. By doing this, first, we avoid the need to explicitly do a matrix transpose on K. Secondly, in the specifically designed matrix multiplication method, since Q and K are stored in memory in row-major order (by default in PyTorch), and we grab Q and K row by row, this access pattern exhibits better spatial locality than doing a naive matrix multiplication, especially when caches are involved. Better yet, Q and K are of dimension (l, d_{head}) where l = sequence length and d_{head} is the hidden dimension divided by the number of attention heads. Therefore each row is of length d_{head} and d_{head} and is usually a factor of 8 (the hidden dimension in the vision transformer is usually 256 or 512). We are using single-precision floating point numbers. Therefore, given that our CPU supports AVX2 (256-bit wide registers), we can employ Intel SIMD intrinsicsl. We load 8 numbers from each matrix into each register and perform a multiplication between them and add them to the resulting accumulated sum. Since we have an equal number of add and multiplication, we can also employ fused-multiply-add to be even more efficient.

Algorithm 1 Transpose matrix multiplication

```
procedure TRANSPOSE_MM($A$, $B$, $C$, $m$, $n$)
    $sum, a, b \leftarrow$ __m256 set to zero
    for $i$ from 0 to $m$ do
        for $j$ from 0 to $m$ do
            $sum \leftarrow$ __m256 set to zero
            for $l$ from 0 to $n$ by 8 do
                $a \leftarrow$ _MM256_LOAD_PS(&A[i * n + l])
                $b \leftarrow$ _MM256_LOAD_PS(&B[j * n + l])
                $sum \leftarrow$ _MM256_FMADD_PS($a, b, sum$)
            end for
            $res \leftarrow$ array of 8 floats
            _MM256_STOREU_PS(res, sum)
            $C[i * m + j] \leftarrow res[0] + ... + res[7]$
        end for
    end for
end procedure
```

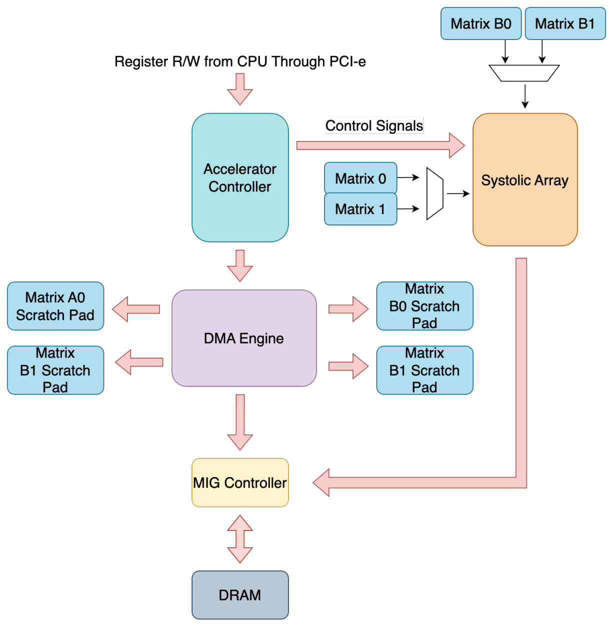


Figure 4: Left: Transpose Matrix Multiplication Algorithm (For-loops are parallelized using OpenMP), Right: FPGA Accelerator Architecture

FPGA HW/SW Codesign

In this project, we integrated a VC707 Xilinx FPGA to offload the matrix multiplication tasks. Our FPGA design consists of a matrix-matrix multiplier accelerator which could be programmed via a PCIe connection to the host CPU. The FPGA is equipped with a PCIe 2.0 x8 connection along with a single 8GB DDR3 DIMM operating at 400 MHz. Furthermore, the accelerator runs at a peak clock speed of 250MHz.

The actual accelerator consists of a controller, DMA engine, systolic array, as well as 4 scratchpads each capable of storing a skewed matrix. The right subfigure of Figure 4 shows the core accelerator architecture. By having 4 scratchpads, we can double buffer the input matrices by using the DMA engine while the FPGA performs matrix multiplication.

Similar to GPU programming, in order to program the FPGA, the user would first need to send the matrices via DMA to the FPGA DRAM. Next, the user would need to issue the GeMM instruction via writing to one of the FPGA's registers. Finally, once the matrix multiplication is complete, the resulting matrix may be read back to the host memory via one last PCIe DMA transfer.

FPGA and CPU parallelism

Since the FPGA can only deal with 1 matrix-matrix multiplication(MMM) instruction and 1 load/read instruction at the same time, we have to perform sequential operations on the CPU side in terms of multi-headed attention.

Matrix Preprocessing

There are several state-of-the-art hardware matrix multiplier architectures such as outer-product engines used in Apple's AMX accelerator, the Coppersmith-Winograd algorithm implemented in Nvidia's NVLDA DNN accelerator, and systolic arrays in Google's TPU family. In this project, we choose to implement the systolic array as our backend matrix multiplier. There are also different systolic array dataflows such as output stationary (OS), weight stationary (WS), and row stationary (RS). In our project, we implemented a 32×32 and 16×16 OS systolic array equipped with Float16 multiply and accumulate units. (A quick OS systolic array demo may be found here) In comparison, Apple's AMX accelerator is a 32×32 output stationary outer product engine and the Google TPUv4 has 4 128×128 systolic arrays. In order to fit the 32×32 systolic array in the FPGA, we reduced the accumulator range aggressively while ensuring that no overflow occurs.

Skewing

In order to perform matrix multiplication on the systolic array, the input matrices need to be skewed for synchronization as shown in Figure 5. As a result of time and FPGA resource constraints, we decided to perform skewing in software which results in a marginal overhead.

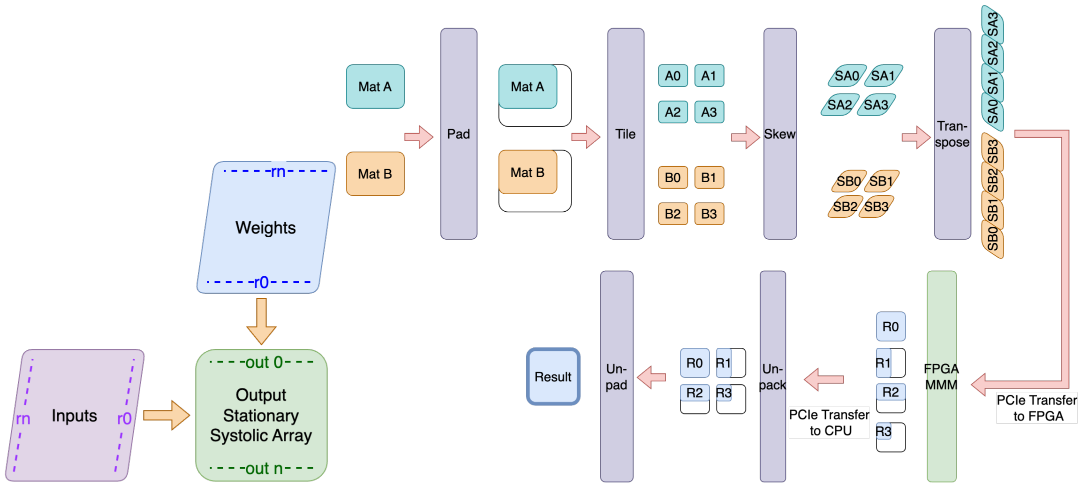


Figure 5: Left: Systolic Array with Skewed Inputs. Right: Software Flow using Systolic Array

Tiling and Padding

Because the systolic array in the FPGA is limited to executing either 16×16 or 32×32 matrix multiplication at a time, we employ a tiled matrix multiplication (MMM) algorithm to compute the accurate outcome. To accommodate the systolic array configuration, as shown in the right subplot of Figure 5 we begin by padding the input matrices to the closest multiple of either 16×16 or 32×32. Once the result is read back from the FPGA DRAM, we would need to first unpack and un-pad the data to obtain the correct result.

General FPGA APIs

The main APIs we use to communicate are `write_from_buffer()`, `read_to_buffer()`, which are provided by Xilinx. We can also write to the FPGA registers to issue instructions like matrix multiplication by calling the same function.

Communications and Computations

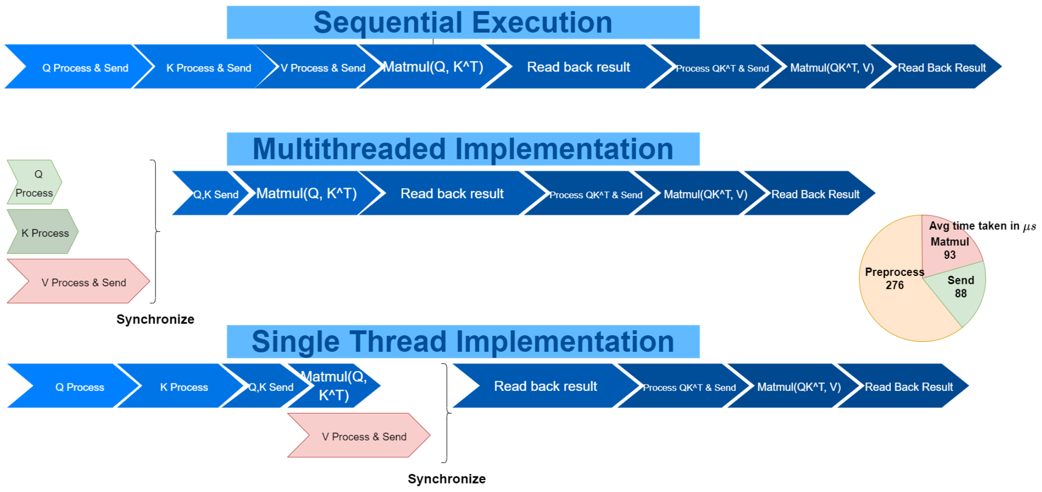


Figure 6: Hiding communication latencies using computations. Pie chart on the right shows approximately how long each step takes comparing to each other. Note that we are doing FPGA+CPU on an old PC with Intel third gen i7-3820, on a better PC the preprocessing step will likely be faster. Also note, this measurements are collected with Q, K, V of size $(256, 64)$, larger matrices will likely result in higher read/write time and higher multiplication time.

In a naive implementation, we would calculate Q, K, V in a sequential manner, we will first send over Q and K transpose and calculate the matrix product QK^T. Then we will read that back and perform a softmax and scaling to get $A = \texttt{softmax}(QK^T/\sqrt{d_{head}})$. Then we send both the resulting matrix A and V to the FPGA memory and perform another matrix multiplication. We've sent 4 matrices and received 2 matrices for each scaled-dot product, and at those times the CPU is not doing anything, which is not utilizing it really well.
An optimized way will be to do something during the time of communication, for example, to compute the product of Q, K, since the preprocessing of Q, K, and V are not balanced: For Q, K we need to do one more elementwise division on either Q or K (which we can do before we have the result of their product) and for K we need an extra transpose operation. Therefore this extra computation time can be used to hide the latency of sending over V. However, since the read/write function does not have a non-blocking version, this method would require us to spawn multiple threads.
We also have a single-thread implementation, which relies on the fact that matrix multiplication instruction can be non-blocking. During the first GeMM between Q, K, we can use the CPU to preprocess V and send it over to the FPGA (it can manage store, and compute simultaneously). By doing this we can hide a little bit of communication overhead, especially when the matrices get larger.

Validation, Verification

Since we are doing inference acceleration, the training of the model is not our priority. Although we have speedup the inference, the training procedure on CPU is still very slow. To save some hassle, we've trained for 4 epochs on a small-scaled ViT and achieved 90% test accuracy. We've saved the trained weights for further inference analysis. Moreover, we've rewritten some matrix multiplication algorithms either on CPU or on FPGA. Therefore PyTorch cannot track and accumulate gradients on the involved tensors, making it impossible to perform backward pass without explicitly writing backward functions. We validate our method using 2 ways: 1. We validate against our other experiments with different settings. For example, we validate our results from FPGA with the results we got using pytorch's `matmul()` function. 2. We verify the numerical consistency of our method by running multiple experiments using the same model and input in different settings. For example, in our CPU-only implementation, we run with the same model on the same data with multiple threads, and each of these runs should provide the same result.

4 Performance Benchmarks and Scaling Analysis

CPU Roofline

For roofline analysis, we are working with Intel® Core™ i9-13900KS Processor (36M Cache, up to 6.00 GHz). It has an average base clock rate (averaged among performance cores and efficient cores)of 4.67 GHz, with 24 cores in total, and supports FMA3 with AVX-512. We are working with 32-bit single precision, so we have SIMD lanes of 512/32 = 16. So the nominal peak arithmetic performance is $\pi = 4.67 * 10^9 * 24 * 16 * 4 = 7173.12$ Gflop/s. Our peak memory bandwidth is $\beta = 89.6$ GB/s. So we have our ridge point at $\frac{\pi}{\beta} = 80$ flop/byte. The roofline plot is shown in Figure 7.

By benchmarking the performance time, we have noticed that the Multi-Head Attention (MSA) step within the ViT block is the most time-consuming. Then to calculate where this algorithm is on the roofline model, we need to compute the operational intensity first. The number of operations per head is listed in the table. The number of memory access per head is listed in table 1. When calculating the number of memory accesses, we assume that q,v,k, and attention matrix can be stored in the cache once they get calculated or updated, which means for the second and third steps in table 1, we do not need to read them in as read access anymore. In our code: N is the batch size and is set to 2048; d is the hidden dimension and is set to 512.

Operation	**Number of Operations Per Head**	**Number of Memory Accesses Per Head**
Calculating q,v,k	$3 \times (2Nd^2)$	$3 \times (2Nd + d^2 + d)$
Softmax(q@k.T)	$3N^2 + N^2 \times 2d$	$2N * d + N^2$
Attention@v	$2N^2d - Nd$	$2Nd + N^2$
Total	$3 \times (2Nd^2) + N^2 \times 2d + 3N^2 + 2N^2d - Nd$	$3 \times (2Nd + d^2 + d) + 2Nd + N^2 + 2Nd + N^2$

Table 1: Comparison of operations and memory accesses per head.

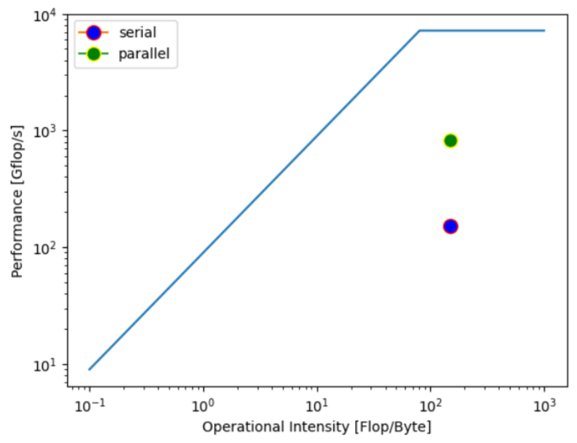
Our roofline is drawn in figure 7.

FPGA roofline

FPGA roofline is calculated in the following way: $\beta = 2 * 0.4 * 1 * 8 = 6.4$ GB/s, where we have $f_{\mathrm{DDR}} =$ 0.4MHz, maximum memory channel is 1; $\pi = 0.25$ GHz$*2$ Flop/cycle$32*32$ channels $= 512$ GFlop/s. We have 16*16 and 32*32 implementations. For 16*16, the operational intensity is $\frac{16*16*16*2}{31*16*2*2} = 4$Flop/Byte and the performance is $\frac{16*16*16*2}{340 \text{ cycles}}0.25\text{GHz} = 6\text{GFlop/s}$. For 32*32 implementation, the operational intensity is $\frac{32*32*32*2}{63*32*2*2} = 8$Flop/Byte and the performance is $\frac{32*32*32*2}{550 \text{ cycles}}0.25\text{GHz} = 29.8\text{GFlop/s}$. The roofline analysis plot is shown in Figure 7.

CPU Scaling Analysis

For the CPU-only implementation without FPGA, we mainly use openMP for thread-level parallelization. Therefore, we are using strong scaling analysis for this section. The experiments are run on Intel® Core™ i9-13900KS Processor (36M Cache, up to 6.00 GHz) from Kung's lab because this allows us to compare



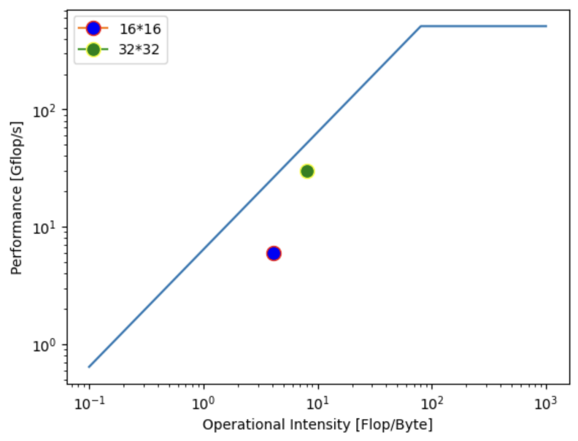
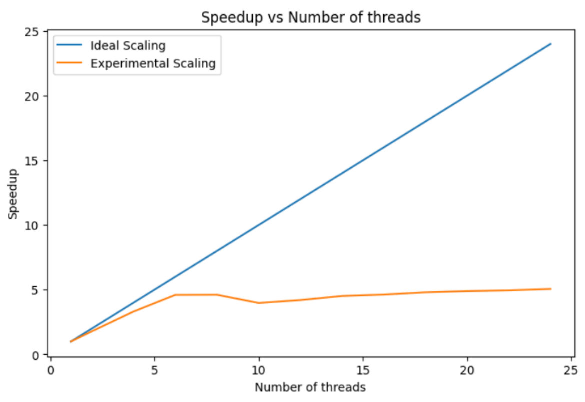


Figure 7: Roofline Model. Left: CPU Only Implementation; Right: FPGA Section

the performance of the code with and without FPGA on the same machine. We have investigated the performance time of using 1 thread (sequential), up to 24 threads.

We can get close to ideal scaling for the first 8 threads. The performance with higher than 8 threads seems to increase very slowly. It could be caused by the fact that our architecture has 8 performance cores and 16 efficient cores, and performance cores have a much higher clock rate of 5.40 GHz compared to efficient cores with 4.30 GHz. In addition, having more threads would add more overhead to the computing progress. The strong scaling plot is shown in figure 8. We also noticed that the same scaling behavior also applies to the linear embedding, where we only have one line of PyTorch linear layer and itself also uses multi-thread parallelization behind. So it further confirms our hypothesis about the observed scaling behavior.



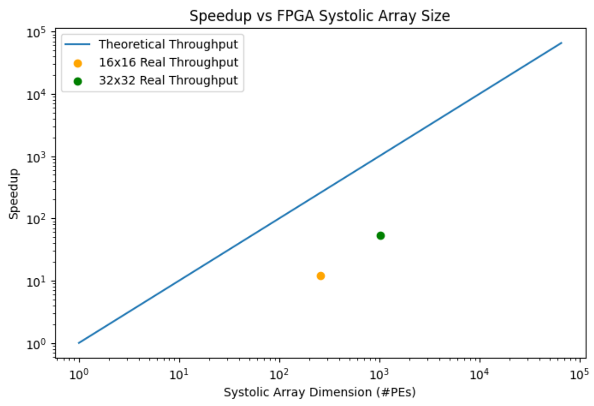


Figure 8: Strong Scaling Analysis. Left: CPU-Only Implementation; Right: FPGA Section

The job size we select for running the analysis is batch size=2048, number of blocks=2, number of heads=64, and hidden layer size=512. With our parallelization, we are able to infer the MNIST set with these parameters in $5 * 10^{-4}$ hours. We can further scale up by increasing the batch size if more resources are given. We typically only need 1 node for CPU-only implementation and the memory we need per node for the current job size is 5.16 GB.

FPGA Scaling Analysis

On the FPGA, we are able to theoretically scale 2 major contributing factors, the size of the systolic array as well as the clock frequency. The size of the systolic array is dependent on the number of hardware resources available on the FPGA (namely LUT and DSP units). In our case, the maximum size of the systolic array that we could fit in our VC707 FPGA was 32×32. In this case, we completely maxed out the number of available DSPs and also consumed over 82% of the available LUTs. Through running benchmarks, we see from the right subplot of Figure 8, that our FPGA implementation scales well with the size of the systolic array.

Scaling the clock frequency of the FPGA will lead to linear performance scaling. In our case, the FPGA's max clock frequency is 250MHz which is around $20\times$ slower than commercial ASICs (application-specific integrated circuits). This is understandable because FPGAs are used mainly for prototyping ASICs. We can linearly scale up our throughput with the clock frequency to estimate the theoretical throughput if our design were to be fabricated into an ASIC.

5 Resource Justification

When it comes to resource allocation our requests may be a bit different due to the nuances of our project and the FPGA implementation. To start in terms of node allocation, we only need one node since we are running on a CPU. In terms of node hours, this is totally dependent on the number of images we train on. For the CPU implementation, we are running a 2048 batch size of 28 pixel * 28-pixel images, which gives a size about 4 MB. We are only using 1 node right now and it runs for $5 * 10^{-4}$ hours. So our node hour is $5 * 10^{-4}$ hours. If we want to process a large amount of data, for example, 100 GB, that means 25,600 iterations, then we would be 12.8 node hours.

For FPGA implementation, the total inference time taken for inferencing a batch size of 64 images is $2.7 * 10^{-4}$ hours. This performance can be greatly improved if we are given more and newer FPGAs, ideally the same amount as the number of threads. and FPGAs with faster clock rates.

A conclusion of our total resource request is shown in the table below.

	CPU Implementation	FPGA (Based on one FPGA data)
Inference size	100 GB	100 GB
Iterations per simulation	25,600	819200
node hours per iteration	$5 * 10^{-4}$ hour	$2.7 * 10^{-4}$
Total node hours	12.8 hour	221 hour

Table 2: Justification of the resource request

References

[1] mLearning.ai. Vision transformers from scratch (pytorch): A step by step guide. https://medium.com/mlearning-ai/vision-transformers-from-scratch-pytorch-a-step-by-step-guide-96c3313c2e0c, 2022. Accessed: 2023-05-04.

[2] Yongping Dan, Zongnan Zhu, Weishou Jin, Zhuo Li, et al. Pf-vit: Parallel and fast vision transformer for offline handwritten chinese character recognition. *Computational Intelligence and Neuroscience*, 2022, 2022.

[3] Minjia Zhang and Yuxiong He. Accelerating training of transformer-based language models with progressive layer dropping. In *Advances in Neural Information Processing Systems*, 2020.