Parallel Design Update: Efficient Vision Transformer Inference using CPU+FPGA

Group 15

Model Output:

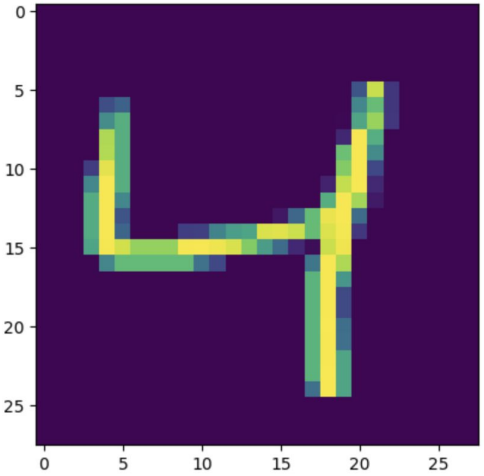
General Description of Architecture:

Vision transformer class, initialize, feed forward, and predict functions within

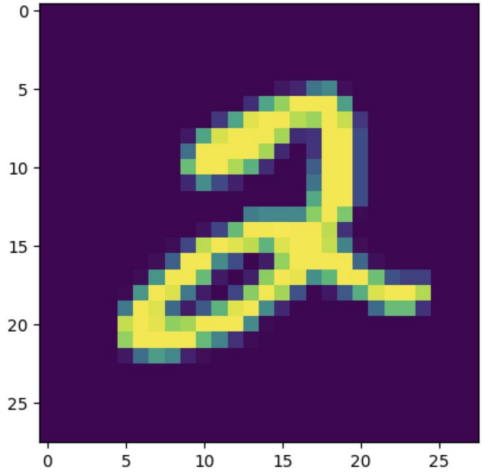
Accuracy, nuances of vision transformer

```
int64_t c = 1;
int64_t h = 28;
int64_t w = 28;
int64_t n_blocks = 8;
int64_t n_patches = 7;
int64_t hidden_d = 8;
int64_t n_heads = 2;
int64_t out_d = 10;
ViT model = ViT(c,h,w,n_patches, n_blocks, hidden_d, n_heads, out_d);
```

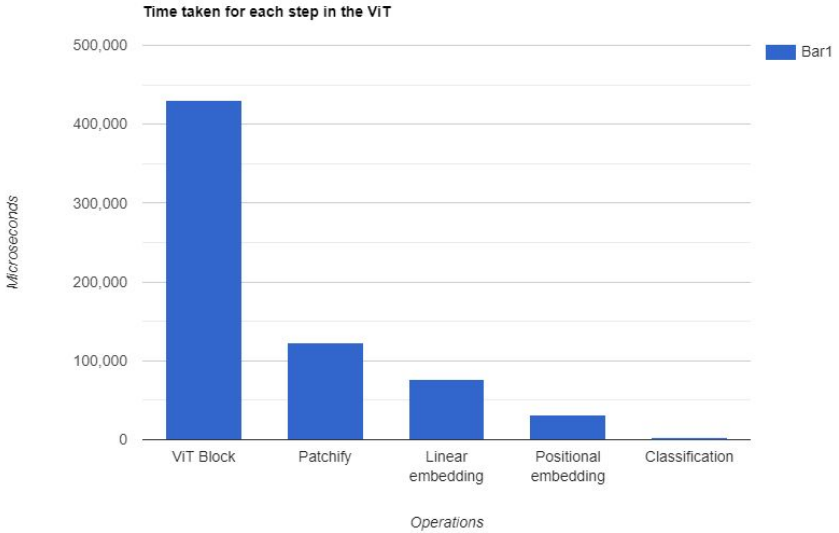
Predicted class: 4

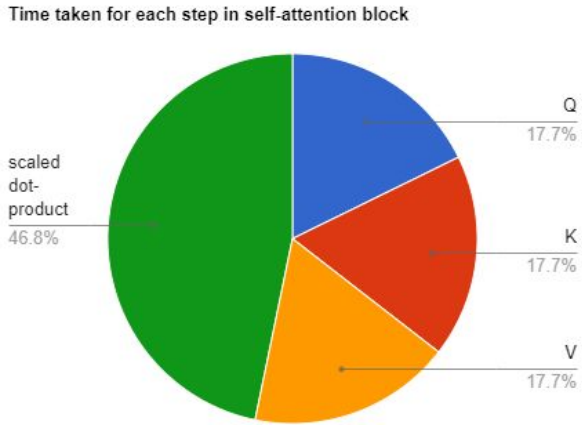


Predicted number: 2



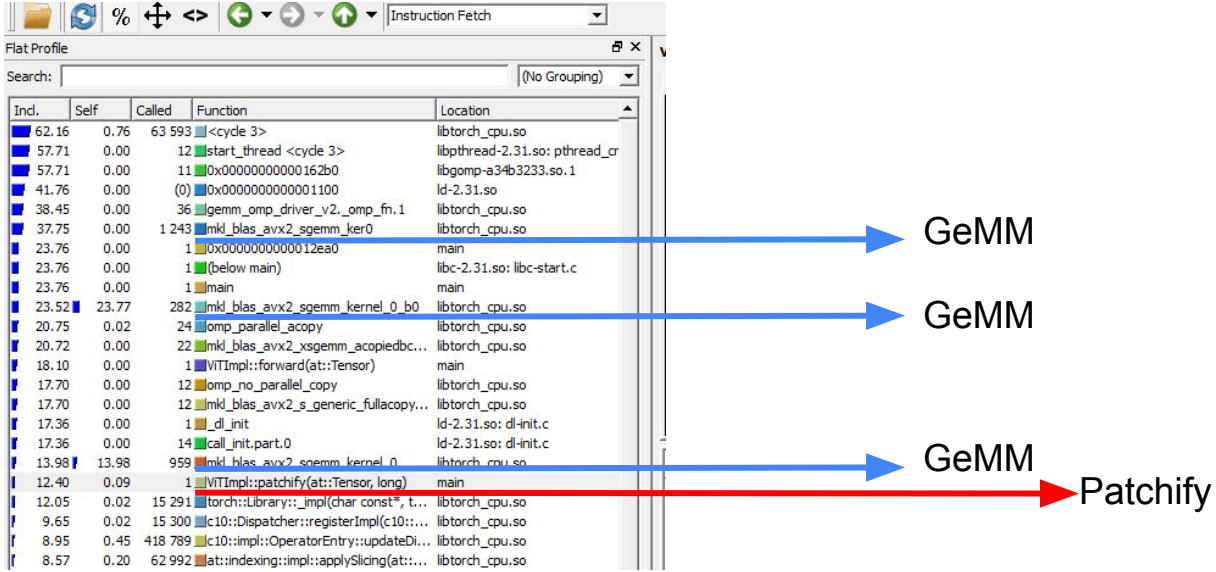
Profiling:





- ➔ Most of the time taken are on the attention blocks, and the patchify step.
- ➔ In each self-attention block, the scaled dot-product takes roughly the same amount of time as the Q,K,V inference combined.

Profiling:

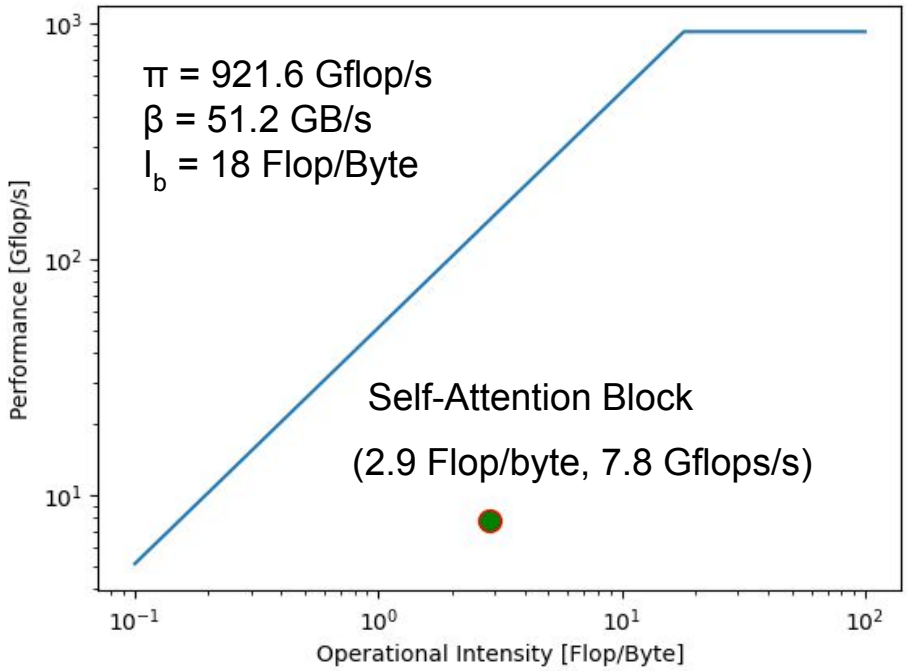


➔ This agrees with our previous observations. Computations that takes the most time are mostly GeMM.

➔ Patchify, without any floating point operation, still takes quite a long time.

Roofline Analysis:

x86_64 architecture



CPU: Ryzen R9 5900x

3.7Ghz, up to 4.8Ghz

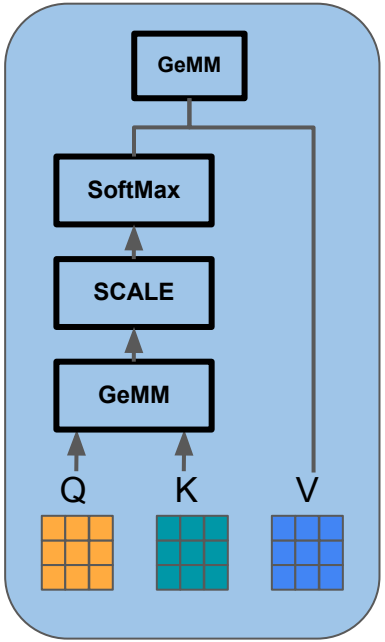
12 Cores

Support AVX2 (256-bit SIMD vector width)

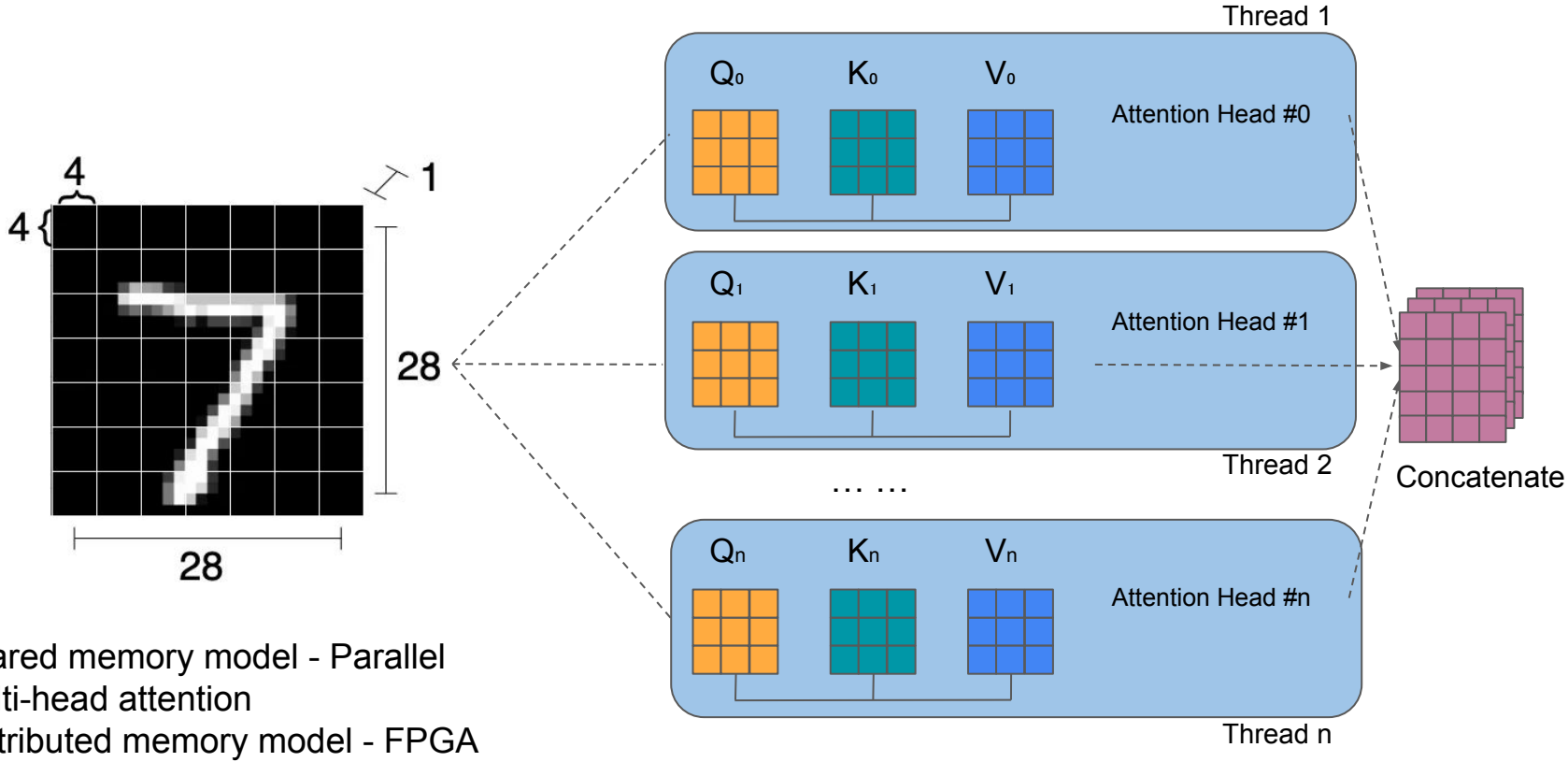
Support FMA3 instructions (4Flop/cycle)

ML models typically use single precision

Self-Attention Block

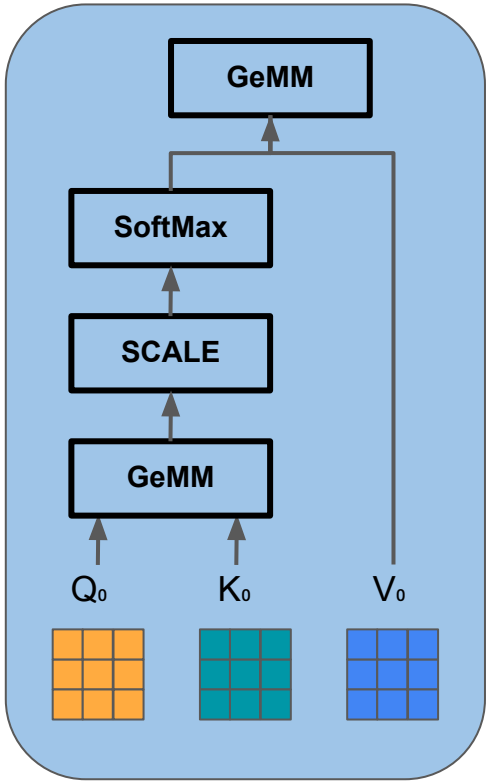


Multi-head attention:



Shared memory model - Parallel multi-head attention
Distributed memory model - FPGA

Self-attention mechanism:



- If we offload computations to the FPGA, we also need to be aware of the communication overhead.
- We can use computation to hide communication latencies.
- When we finish calculating Q on CPU, we send Q to the FPGA while calculating K on CPU. When we finish the computation of K, we send K to the FPGA and perform a series of operations while we calculate V on the CPU and send it to FPGA.

Benchmark **3** implementations:

1. CPU only
2. Hybrid (CPU network inference + FPGA scaled-dot product)
3. Mostly FPGA (Except some tensor slicing/indexing)

FPGA Computation Steps

- Currently the plan for the FPGA implementation only accelerates GEMM computation
- FPGA GEMM logic:
- pcie_dma(&a, &b, fpga_ptr_a, fpga_ptr_b, HOST2DEV)
- set_fpga_reg(fpga_ptr_a, fpga_ptr_b, fpga_ptr_c, &M, &K, &N)
- issue_fpga_instr(I_FPGA_GEMM)
- while(~read_fpga_reg(fpga_status))
- pcie_dma(fpga_ptr_c, &C, DEV2HOST)

FPGA Implementation Logic

- Single thread
  - Only parallelism is on the FPGA
  - GEMM computation is parallelized
- Multi-thread
  - Need locks for FPGA acceleration
  - Threads have the option to compute GEMM on CPU using vector instructions
  - Pipeline FPGA design to hide the weight times
  - FPGA is bottlenecked by the onboard DRAM
  - FPGA can also be bottlenecked by PCIe transfer speeds

	PCIe 2.0 x8 (theoretical)	DDR3 @ 800MHz (theoretical)	FPGA (experimental) (2GB transfers) R/W
Bandwidth	5GB/s	6.4GB/s	3.5/2.4 GB/s

Patchify:

```cpp
auto patch = image.index({
    torch::indexing::Slice(),
    torch::indexing::Slice(i * patch_size, (i + 1) * patch_size),
    torch::indexing::Slice(j * patch_size, (j + 1) * patch_size)
});

// Flatten the patch and add it to the output tensor
patches.index_put_({idx,i * n_patches + j},  patch.flatten());
```

Create 2 copies for each patch.
5 Read + Write per element per patch.

~120000 ms

```cpp
patches.index_put_({idx,i * n_patches + j},  image.narrow(1, i * patch_size, patch_size).narrow(2, j * patch_size, patch_size).flatten());
```

Use tensor::narrow to create a view of the patch.
3 Read + Write per element per patch.

~90000 ms, 1.3x speedup

Patchify:

Task-level Parallelism:

- The nested for loops can be unrolled into a single for loop.
- Using *#parallel omp for* to divide the task of copying, reshaping and writing each patch to each thread and perform them in parallel.

```
#parallel omp for collapse(2)
for i from 0 to npatch:
    for j from 0 to npatch:
        patch = image.narrow(,,,).flatten()
```

- Benchmark against the PyTorch implementation.

Pytorch Implementation:

- Using vectorization on the image tensor:

```
image = image.reshape(c, npatch, patch_size, npatch, patch_size)
image = image.permute(0,1,3,2,4)
image = image.reshape(npatch**2, -1)
```

- Can be more efficient than the for loop with OpenMP parallelization, as vectorized operations can be highly optimized and can take advantage of SIMD instructions on modern CPUs.
- However, performance will still depends on the size and number of patches.

References:

https://medium.com/mlearning-ai/vision-transformers-from-scratch-pytorch-a-step-by-step-guide-96c3313c2e0c

Thank you!