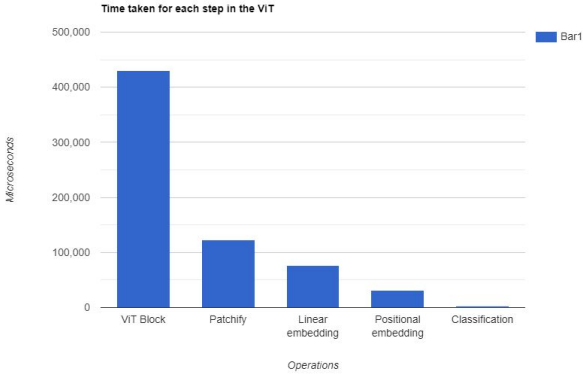
Efficient Vision Transformer Inference using CPU+FPGA

Problem Review: Motivation + Justification

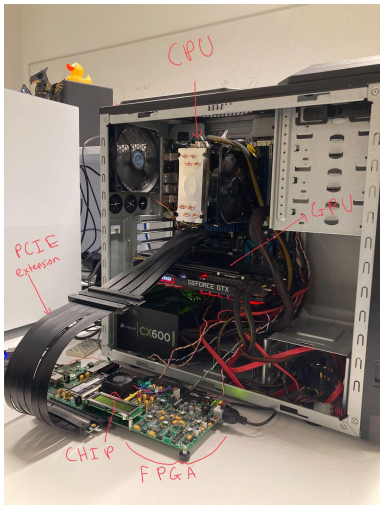


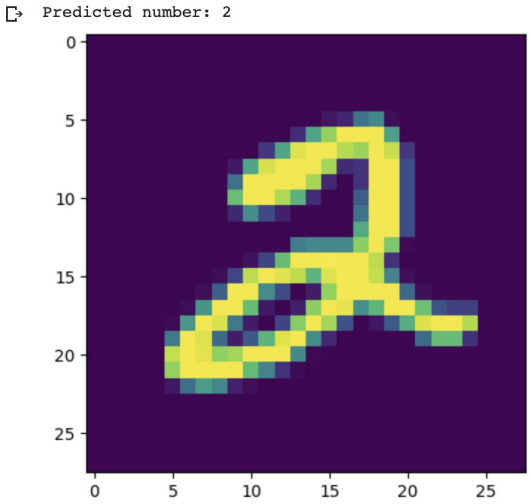
Parallelize vision transformer, offload GeMM to FPGA

Benchmark on different configurations of CPU, GPU, FPGA

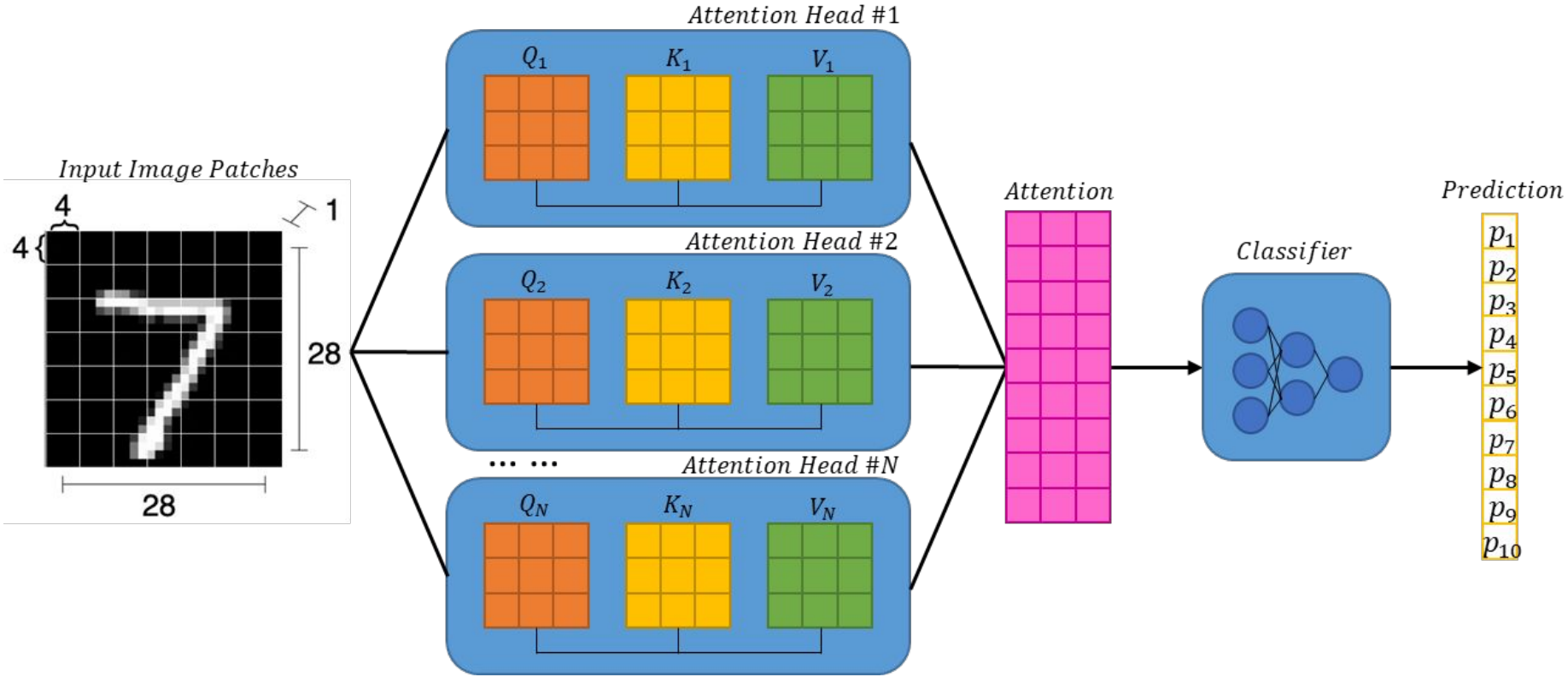
Why Vision transformers?

Why FPGA?

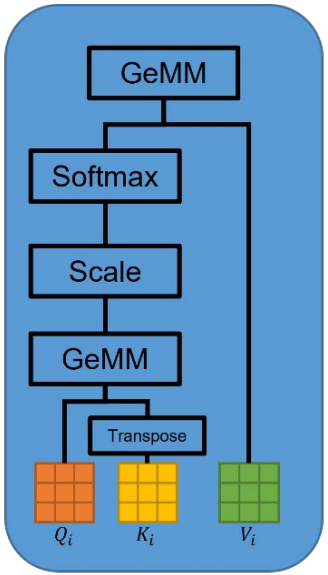




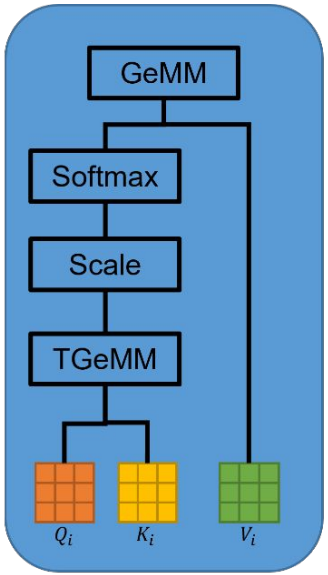
Parallelized Multi-headed Attention



Scaled Dot Product

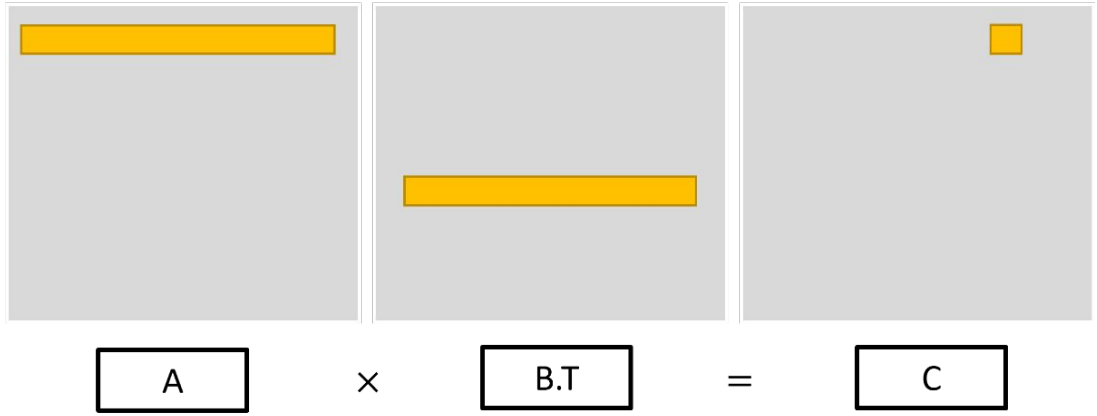


Scaled-dot product



Fused-transpose GeMM

Fused-transpose GeMM

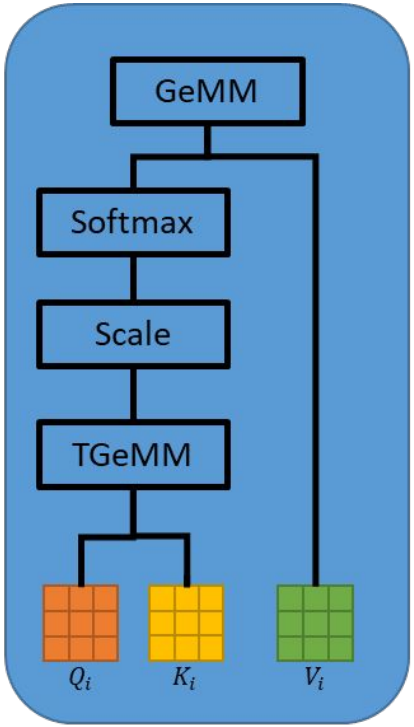


Transpose Matrix Multiplication

Algorithm 1 Transpose matrix multiplication

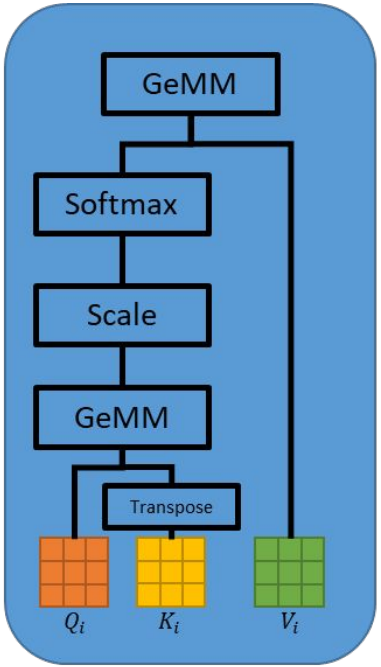
```
1: procedure TRANSPOSE_MM(A, B, C, m, n)
2:     sum, a, b ← __m256 set to zero
3:     for i from 0 to m do
4:         for j from 0 to m do
5:             sum ← __m256 set to zero
6:             for l from 0 to n by 8 do
7:                 a ← _MM256_LOAD_PS(&A[i * n + l])
8:                 b ← _MM256_LOAD_PS(&B[j * n + l])
9:                 sum ← _MM256_FMADD_PS(a, b, sum)
10:            end for
11:            res ← array of 8 floats
12:            _MM256_STOREU_PS(res, sum)
13:            C[i * m + j] ← res[0] + ... + res[7]
14:        end for
15:    end for
16: end procedure
```

Parallelized using OpenMP

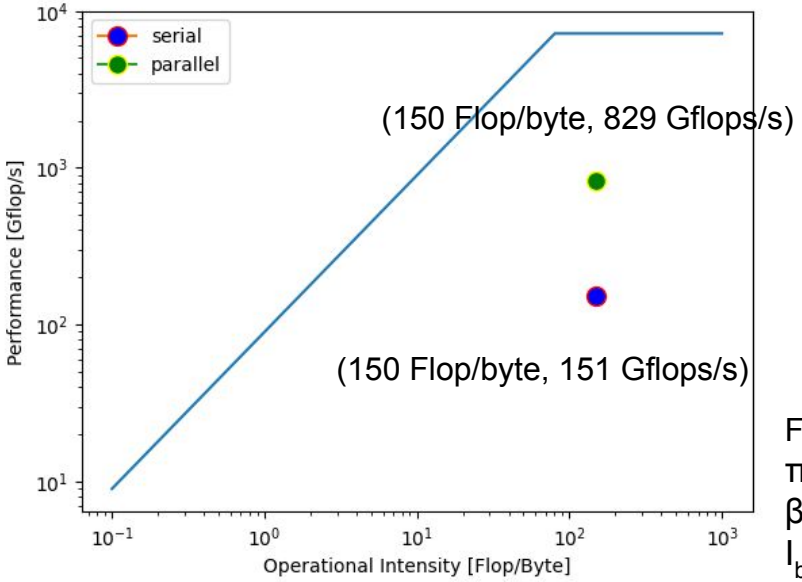


Fused-transpose GeMM

Self-Attention Mechanism Roofline Analysis



Scaled-dot product

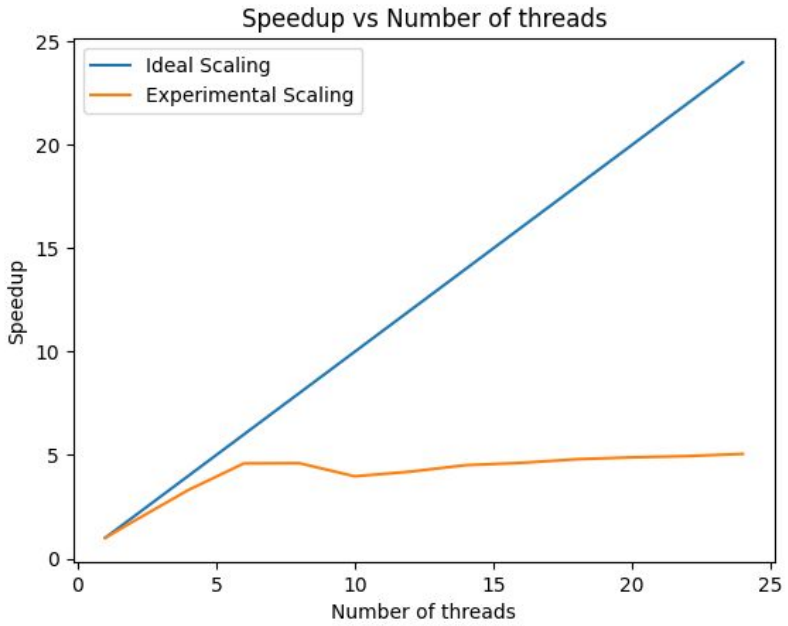


Architecture: Intel® Core™ i9-13900K Processor (24 cores)

- 4.7 GHz,
- Support AVX-512
- Support FMA3 instructions (4Flop/cycle)
- ML models typically use single precision

For batch size=2048, hidden layer=512:
π = 7173.12 Gflop/s
β = 89.6 GB/s
I_b = 80.05 Flop/Byte

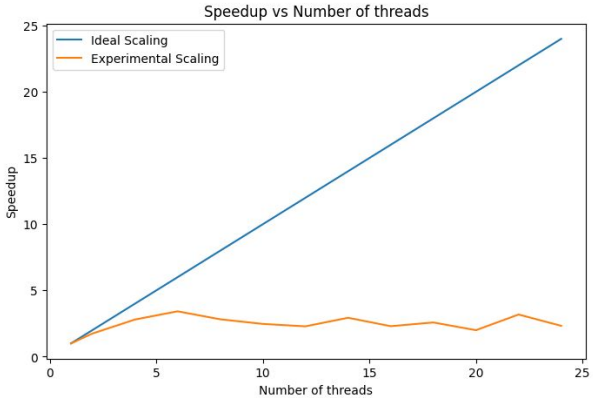
Performance and benchmarking: CPU



Potential reason: We only have 8 performance cores with higher clock rate

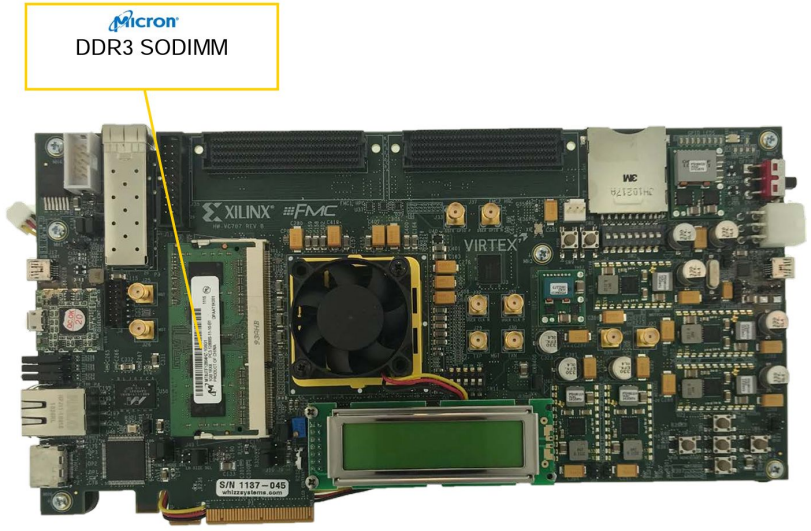
Linear embedding using Pytorch show similar behavior:

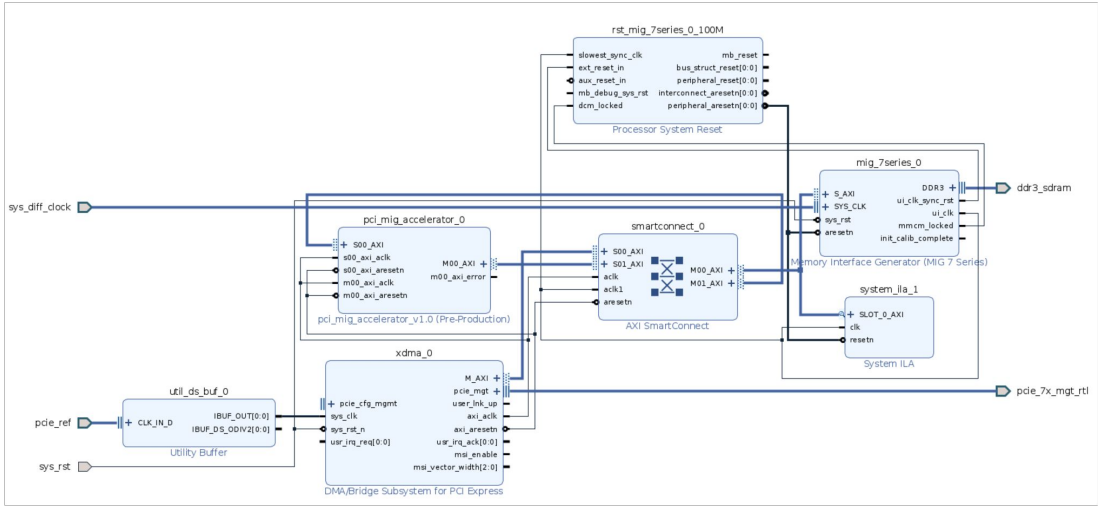
```
torch::Tensor tokens = linear_mapper->forward(patches);
```



FPGA implementation and setup

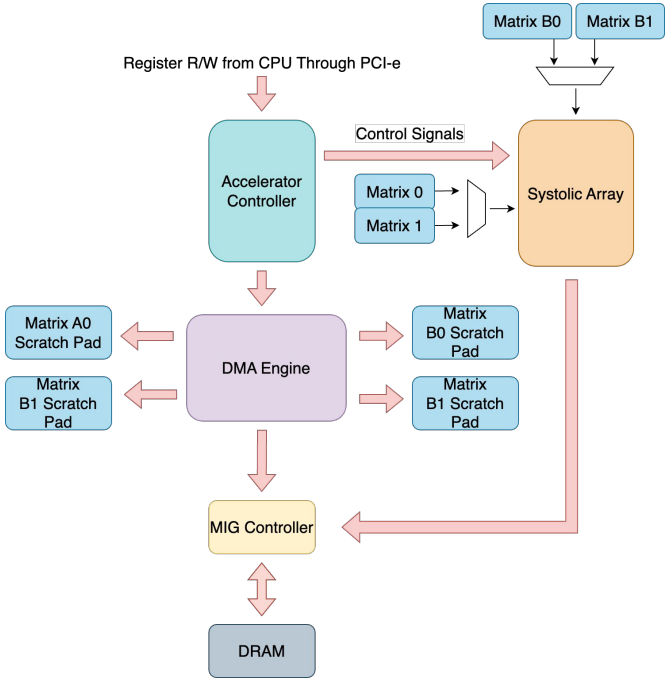
- PCI-E DMA to on board DDR3 DRAM
  - Similar to GPU/TPU architecture
- PCI-E 2.0 and DDR3 @ 4 MHz
  - Memory bound
- FPGA @ 250MHz
- 1.5 hour compilation time
  - Hard to debug, especially staring at waveforms
- Timing requirements
  - To guarantee 250 MHz clock





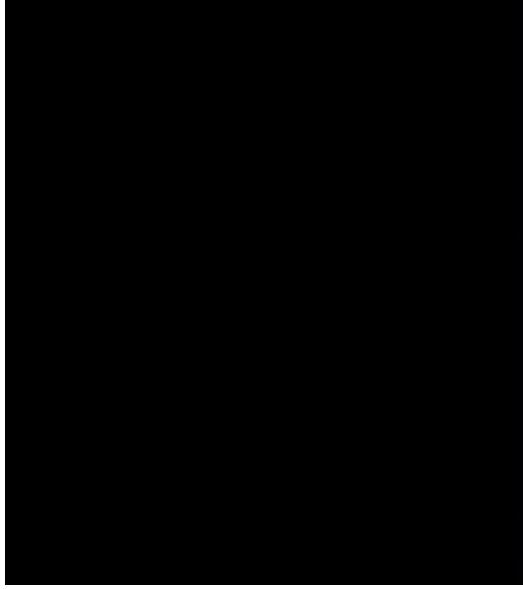
FPGA implementation and setup

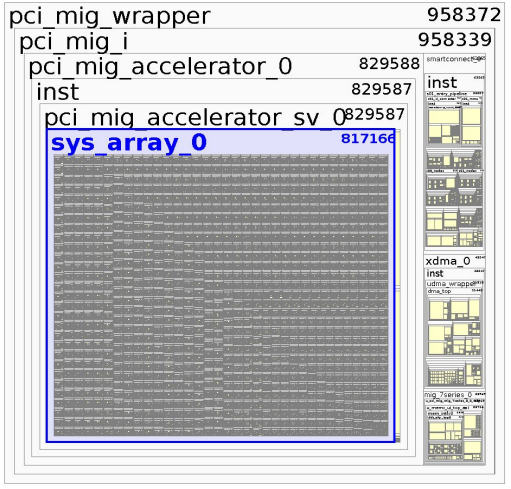
- Accelerator Controller
  - Execution logic
- Scratchpads
  - Double buffered
- DMA Engine
  - Hides memory overhead
- Systolic Array



FPGA - Systolic Array

- Matrix multiplier
- Capable of performing n^2 MACs per cycle
- Output stationary dataflow

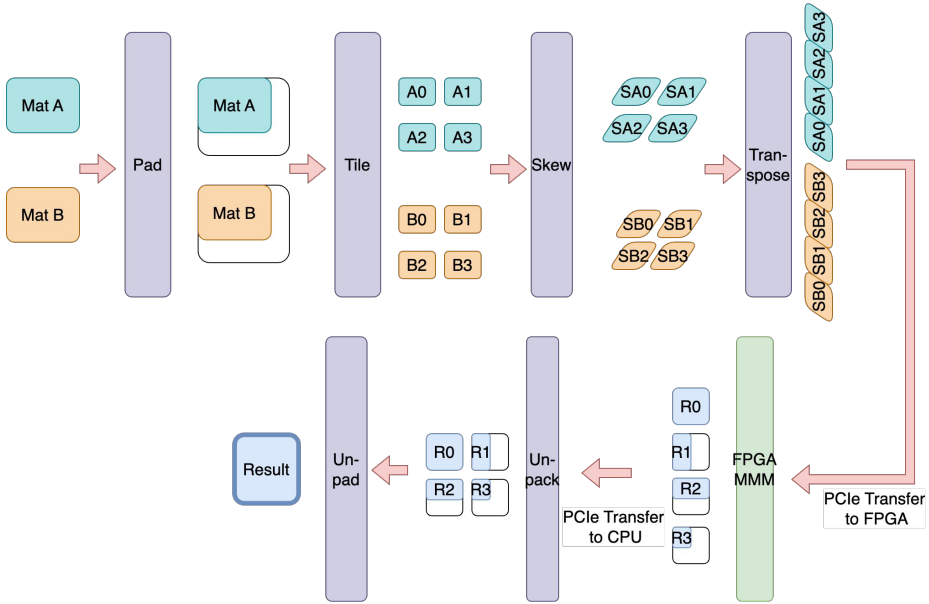




FPGA + CPU

To utilize FPGA on our vision transformer, a lot of work has to be done.

1. Padding
2. Tiling
3. Skewing
4. Transpose
5. Untiling
6. Unpad

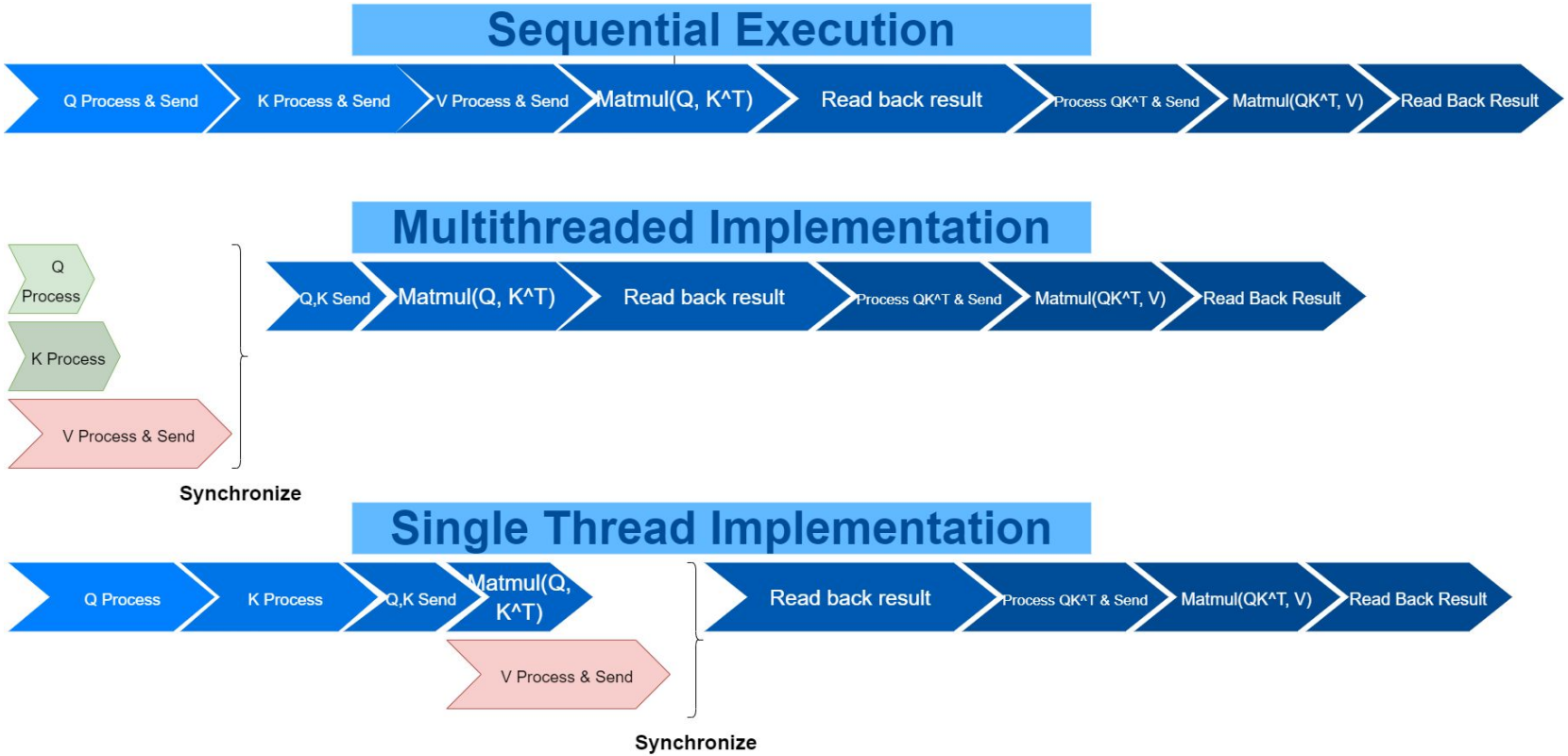


FPGA + CPU

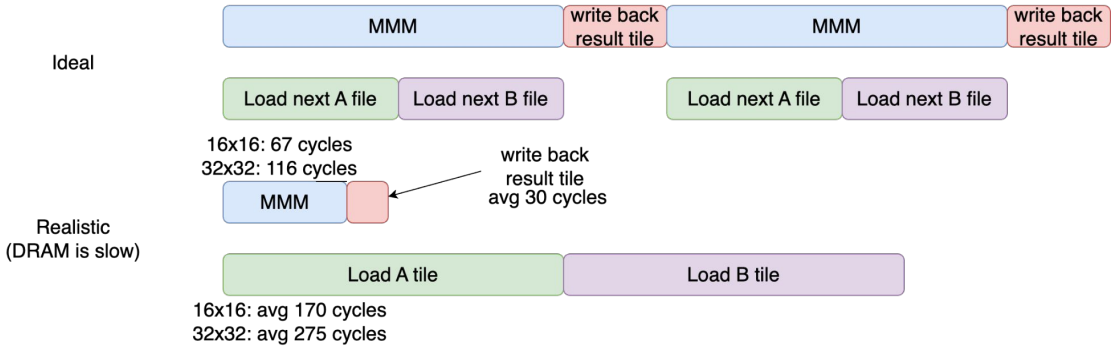
Furthermore, countless of memory allocations/access needs to be carefully done.

```
# Open FDs for read and write
fpga_w_fd = open(WRITE_DEV_NAME, O_RDWR);
fpga_r_fd = open(READ_DEV_NAME, O_RDWR);

# Writing to instruction register
fpga_reg_write(mat_d_addr, R_MATRIX_B_ADDR);
fpga_reg_write(mat_c_addr, R_MATRIX_C_ADDR);
fpga_reg_write(fpga_num_mm, R_MATRIX_RD_CNT);
fpga_reg_write(mat_m_size, R_MATRIX_M_SIZE);

# Write and read from FPGA
write_from_buffer(WRITE_DEV_NAME, fpga_w_fd, (char
*)temp_att.data_ptr<at::Half>(), mat_ab_offset,
mat_a_addr);

read_to_buffer(READ_DEV_NAME, fpga_r_fd, (char
*)temp_holder, 16*OUT_MATRIX_BUF_SIZE, mat_c_addr);
```

FPGA + CPU


Sequential Execution
Q Process & Send
K Process & Send
V Process & Send
Matmul(Q, K^T)
Read back result
Process QK^T & Send
Matmul(QK^T, V)
Read Back Result
Multithreaded Implementation
Q Process
K Process
V Process & Send
Synchronize
Q,K Send
Matmul(Q, K^T)
Read back result
Process QK^T & Send
Matmul(QK^T, V)
Read Back Result
Single Thread Implementation
Q Process
K Process
Q,K Send
Matmul(Q, K^T)
V Process & Send
Synchronize
Read back result
Process QK^T & Send
Matmul(QK^T, V)
Read Back Result

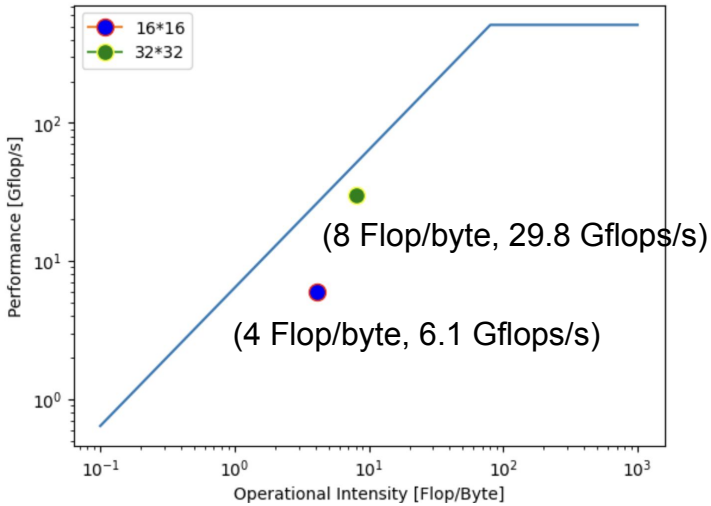
Performance and benchmarking: FPGA

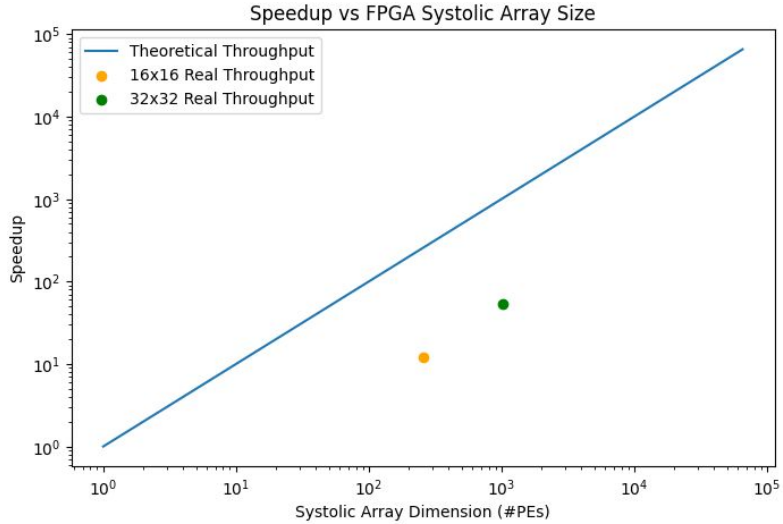


Performance and benchmarking: FPGA

Clock rate:
0.4 GHz

1 channel





π = 512 Gflop/s
β = 6.4 GB/s

Future Work and Project Insights

In the future we hope to

- Program the FPGA to work with GPU. more advanced
- Multiple FPGAs so that they equal thread #
- Optimize accelerator architecture (tiling, dataflow and caching)
- Tapeout a chip specifically for GeMM

Project Insights

- limitless possibilities of FPGA
- Customizable hardware where speed is important

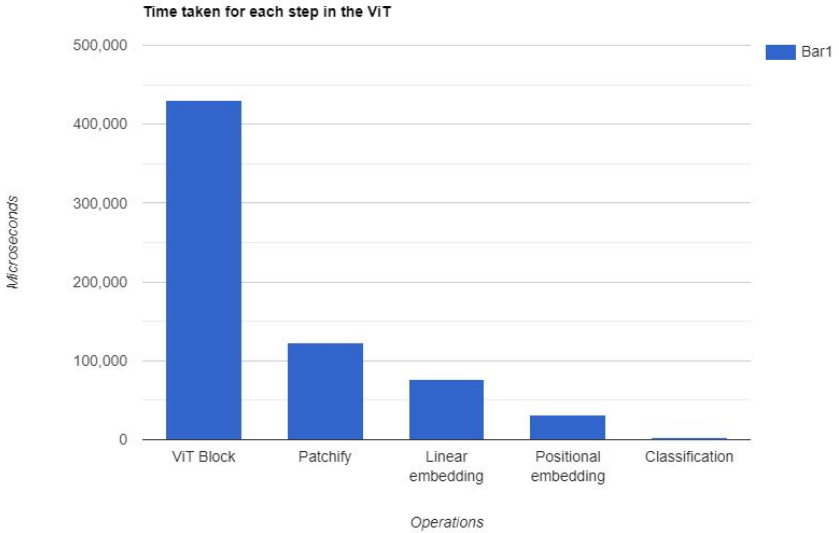
Talk about our futures and how this project will affect them

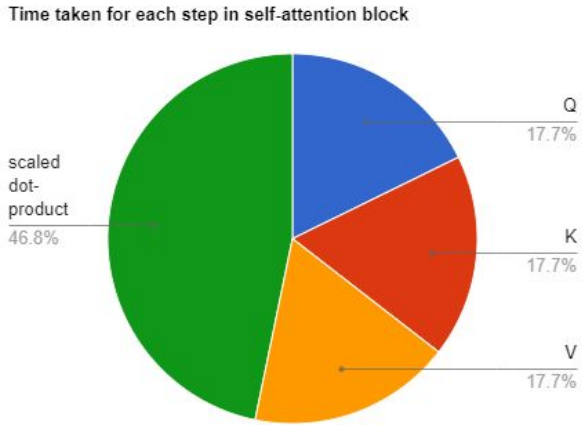
References:

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

Thank you!

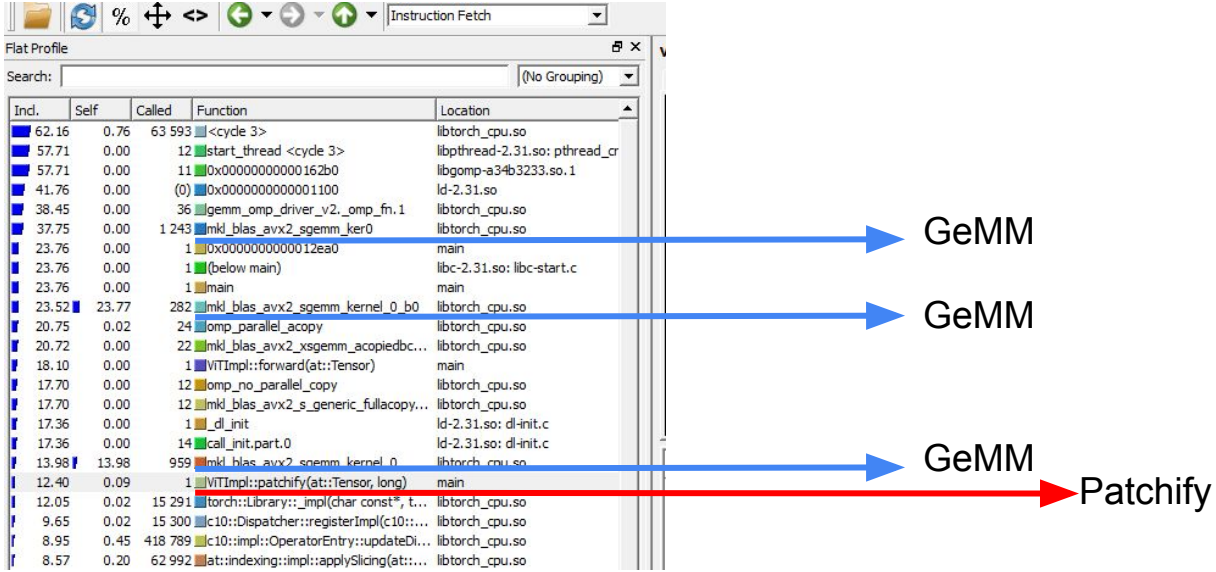
Profiling:





- ➔ 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:

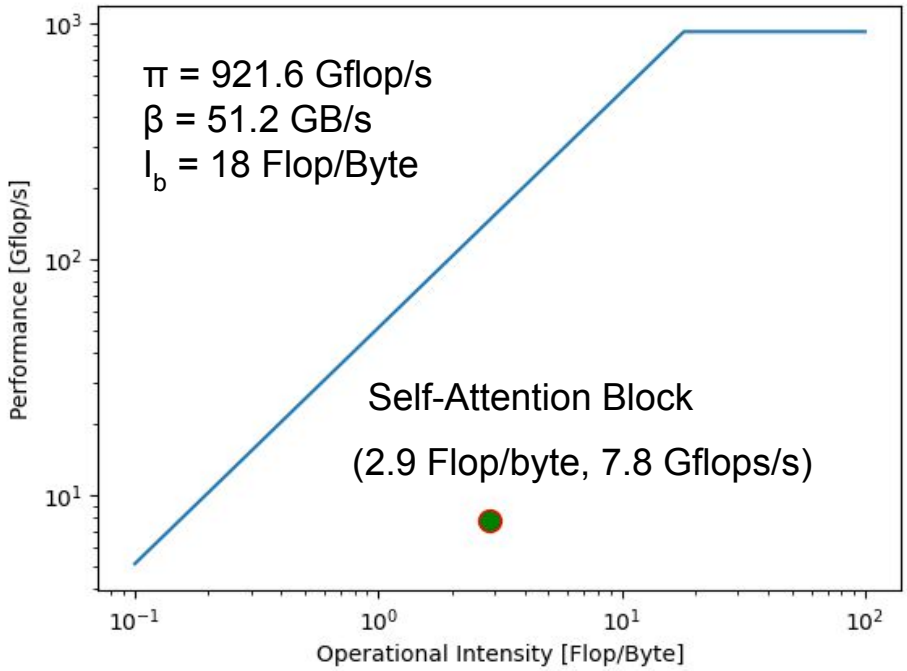


➔ This agrees with our previous observations. Computations that takes the most time are mostly GeMM in the self-attention module.

➔ 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

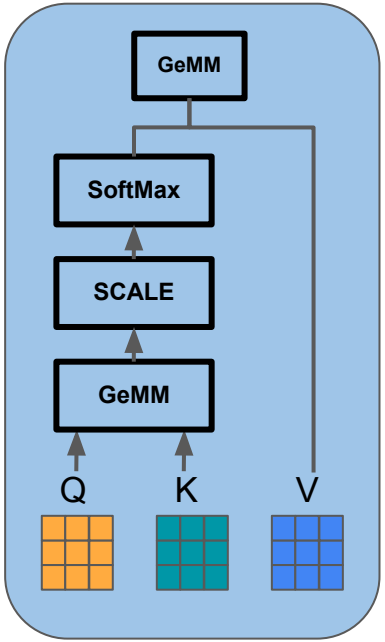
12 Cores

Support AVX2 (256-bit SIMD vector width)

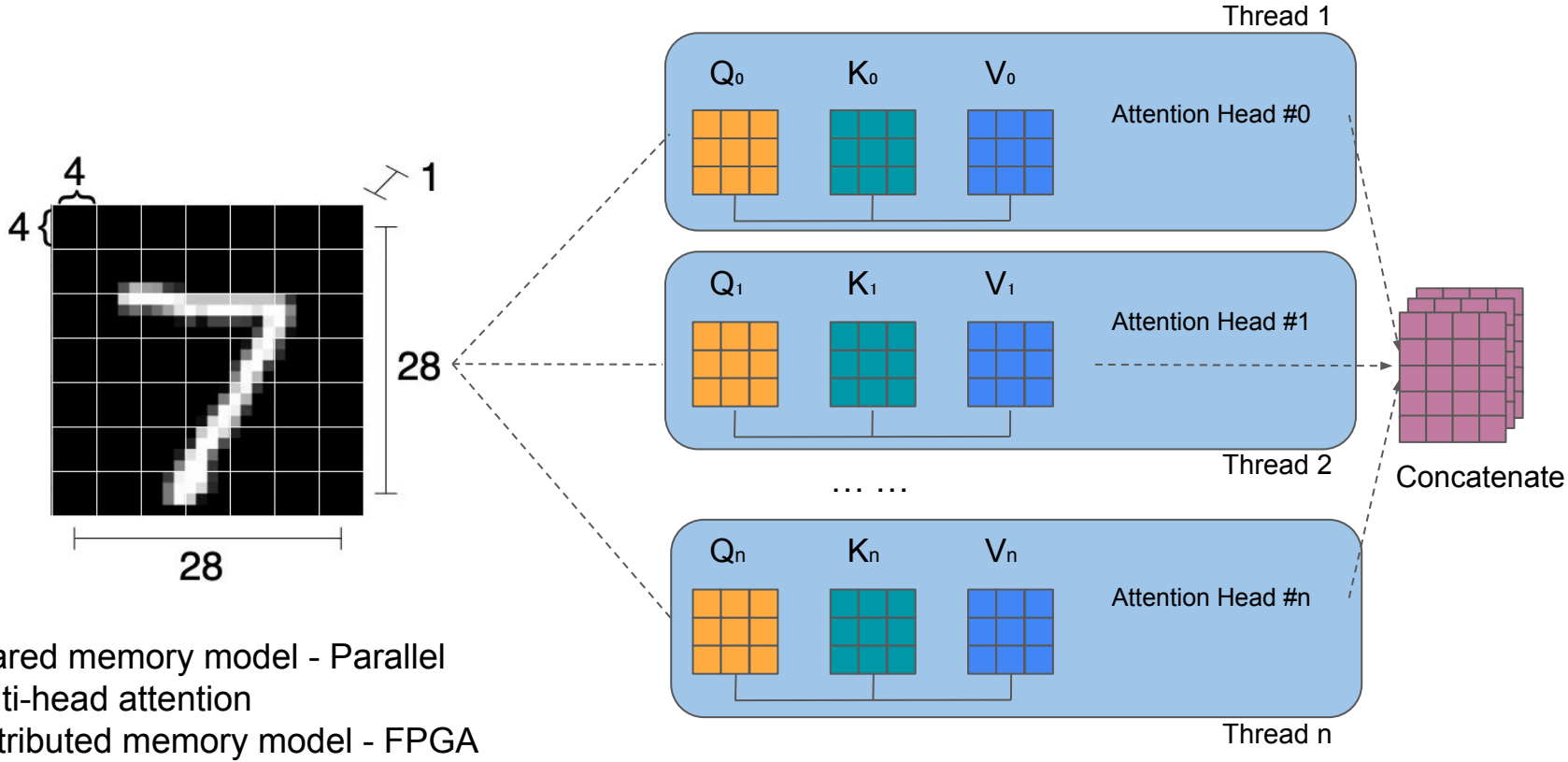
Support FMA3 instructions (4Flop/cycle)

ML models typically use single precision

Self-Attention Block

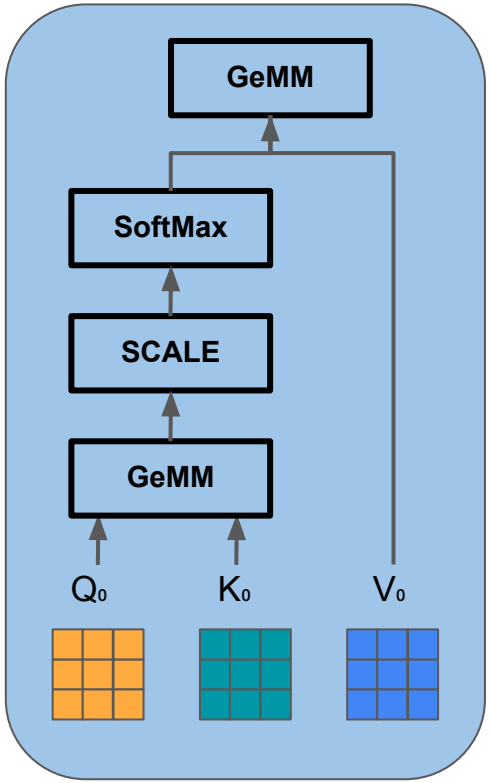


Multi-head attention:



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!