Computer Science 133, Lab 5

Michael Wu
UID: 404751542

March 18th, 2019

1 Parallelization Strategy

In order to parallelize this code, I first began by writing a working version by comparing the sample code to the sequential code. The parallelized sections that were provided to us essentially allows us to multiply an entire 5×5 section of the weight array in parallel. It also pipelines each weight layer multiplication. Additionally, it provides code for pipelining and unrolled the loading of the input array. So I simply had to write code to load the bias value and store the output value using the ReLU function and max pooling. I wrapped the provided pipelined code inside larger loops to ensure that the kernel iterates through the entire convolution correctly. After this part I achieved a speed of approximately 6 GFlops.

In order to optimize my code, I investigated the performance summary and found that the majority of the computation time was spent in the loading and convolution steps. So I focused on these sections and I left the bias and output calculation sections to be sequential. I unrolled the convolution step by 16 times and partitioned the output and input buffers appropriately so that they could be accessed in parallel. I was able to reduce the latency of the convolution step by 16 times while maintaining an initiation interval of 1. Any additional unrolling and partitioning resulted in a synthesis time that was too long. I achieved slightly less than double the speed in my overall execution time after performing these actions.

Next I turned my attention to the loading step, since this was the slowest operation after I optimized the convolution. I found that I could only unroll the loading 4 times before the initiation interval began increasing. For example, when I unrolled 12 times my initiation interval became 3 and I achieved

the same speed. I believe this limitation is due to a memory bottleneck. Since the load operation reads from global memory, the available memory units limit my parallelism. This optimization gave me approximately 4 times the speed for the following final throughput.

$$\frac{256^2 \times 224^2 \times 5^2 \times 2 \times 250}{1109859329 \times 10^3} = 37 \text{ GFlops}$$

I obtained a sequential version speed of 11.5451 GFlops in the previous labs so this represents a 3.2 times speedup over the sequential version.

Lastly, I unsuccessfully attempted to use vector types and the dataflow pragma in order to further increase my performance. I had trouble with these methods and could not find a working solution.

2 Details of Optimizations

I declared my temporary buffers using the following code.

```
float output_buf[kImSize][kImSize]
__attribute__((xcl_array_partition(cyclic, 8, 1)))
__attribute__((xcl_array_partition(cyclic, 2, 2)))
;

float input_buf[kInImSize][kInImSize + kKernel - 1][kKernel]
__attribute__((xcl_array_partition(cyclic, 8, 1)))
__attribute__((xcl_array_partition(cyclic, 2, 2)))
__attribute__((xcl_array_partition(complete, 3)))
;
```

These were the minimum number of memory partitions necessary to allow for my unrolling to work. Then I unrolled the loading stage in the following fashion.

```
__attribute__((xcl_pipeline_loop))
for (int w = 0; w < kInImSize; w+=4) {
  for (int q = 0; q < kKernel; q++) { //make kKernel copy of input(j,h,w)
    input_buf[h][w     - q + kKernel - 1][q] = input(j, h, w    );
    input_buf[h][w + 1 - q + kKernel - 1][q] = input(j, h, w + 1);
    input_buf[h][w + 2 - q + kKernel - 1][q] = input(j, h, w + 2);
    input_buf[h][w + 3 - q + kKernel - 1][q] = input(j, h, w + 3);
  }
}
```

I unrolled the convolution stage in the following fashion.

```
for (int h = 0; h < kImSize; h+=8) {
  conv:
  __attribute__((xcl_pipeline_loop))
  for (int w = 0; w < kImSize; w+=2) { //pipelined loop
    float tmp0 = 0;
    // ...
    float tmp15 = 0;
    for (int p = 0; p < kKernel; p++) {  // unrolled loop
      for (int q = 0; q < kKernel; q++) {  //unrolled loop
        tmp0 += //will be synthesized into tree reduction
          weight_buf[p][q] * input_buf[h     + p][w + kKernel - 1][q];
        // ...
        tmp7 +=
          weight_buf[p][q] * input_buf[h + 7 + p][w + kKernel - 1][q];
        tmp8 +=
          weight_buf[p][q] * input_buf[h     + p][w + kKernel    ][q];
        // ...
        tmp15 +=
          weight_buf[p][q] * input_buf[h + 7 + p][w + kKernel    ][q];
      }
    }
    output_buf[h    ][w    ] += tmp0; //store reduction result
    // ...
    output_buf[h + 7][w    ] += tmp7;
    output_buf[h    ][w + 1] += tmp8;
    // ...
    output_buf[h + 7][w + 1] += tmp15;
  }
}
```

3 Differences from Previous Labs

My optimization strategies for this lab differs from the previous labs because in the previous labs the parallelism is achieved through the OpenCL framework an a fixed processor architecture. In this lab I do not use any additional work items and instead optimize my architecture for the computation. This means that I tried to parallelize the innermost loops by adding more memory units and arithmetic units, while in the previous labs I mainly tried to parallelize the outermost loops. I did do some unrolling in lab 3, but this was to fit my kernel to the CPU architecture. Otherwise I tried to keep the kernel small so it could execute on different outer loop iterations.

4 Resource Usage

The FPGA resource usage is shown in the following table.

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	0	0	10332
FIFO	-	-	-	-
Instance	30	2032	219835	159833
Memory	768	-	0	0
Multiplexer	-	-	-	3971
Register	16	-	50124	15653
Total	814	2033	269959	189789
Available	4320	6840	2364480	1182240
Utilization(%)	18	29	11	16

The DSP48E resource has been used the most in terms of percentage. This makes sense since these components are used for the multiplication and addition of floating point values.

To calculate the utilization, I would take a look at the array sizes and find that the input buffer has size $228 \times 232 \times 5 = 264480$, the output buffer has size $224 \times 224 = 50176$, and the weight matrix has size $5 \times 5 = 25$. Since the size of a float is 4 bytes, this means that the kernel uses the following amount of memory.

$$4 \times (264480 + 52176 + 25) = 1266724 \text{ Bytes}$$

Theoretically the minimum number of 18Kb BRAM blocks that I would need is given by the following calculation.

$$\frac{1266724}{18 \times 1024} = 69$$

When looking at my loop I see that there are 16 floating point multiplications in the inner loop. After unrolling the p and q loops, this results in a total of $16 \times 25 = 400$ floating point multiplications being done in parallel. This would be the minimum number of DSP units necessary. In reality the synthesis uses more of both the memory and DSP units in order to allow for shorter critical paths and parallel accesses in order to reduce latency. That is why the shown resource utilization is much higher than the minimum.

5 Merlin Parallelization Strategy

My parallelization strategy followed the same structure as my SDAccel code. I used temporary buffers, pipelining, and flattening in the same places as my

SDAccel implementation. I also used a pragma to indicate a false dependency in the convolution buffer so that the convolution step could be parallelized. This led to a reasonable speed. My calculation for the throughput is shown below.

$$\frac{256^2 \times 224^2 \times 5^2 \times 2 \times 250}{4723995781 \times 10^3} = 8.7 \text{ GFlops}$$

Since the sequential throughput was 11.5451 GFlops, this represents 0.75 times the performance of the sequential version. It is worse than the SDAccel code. However, this is still orders of magnitude better than the baseline Merlin code. I chose to parallelize the main convolution loop only 8 times instead of 16 times like in my SDAccel implementation in the interest of time. Synthesizing my code took much longer with Merlin. I found that most of my latency was again due to reading memory.

6 Merlin Resource Usage

The FPGA resource usage is shown in the following table.

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	7	-	-
Expression	-	-	0	37102
FIFO	-	-	-	-
Instance	30	1041	118065	87934
Memory	1076	-	0	0
Multiplexer	-	-	-	10430
Register	0	-	72290	22311
Total	1106	1048	190355	157777
Available	4320	6840	2364480	1182240
Utilization(%)	25	15	8	13

The BRAM_18K resource has been used most in terms of percentage.

7 Merlin Optimization Analysis

Merlin seems to have flattened and pipelined the same loops as SDAccel. The SDAccel HLS report for this flow looks like my unoptimized version of the SDAccel code. The Merlin report indicates that certain sections are 4x,

5x, and 8x parallelized, which would speed up my performance by having parallel processing paths for the data to flow through. The pipelining improves performance by allowing the next iteration of a loop to begin while the previous iteration is still being processed so that the data flows through the processor without delay. Flattening improves performance by unrolling a loop entirely so that the whole loop is processed in parallel.

8 Comparison to SDAccel

In my implementations, I was able to achieve better performance with SDAccel. This was because making an efficient Merlin implementation required absurd amounts of synthesis time. Merlin allows for easier coding and less involved optimization strategies since it analyzes code for you, but this increases the computational power required to run it. SDAccel requires more explicit optimizing by the programmer, but it has a faster synthesis time. If I had enough time to run Merlin and SDAccel for long periods of time, perhaps my results would be different. In that case Merlin might be able to optimize better than I could by hand. This matched my expectations, as computing tools that operate at a higher level of abstraction typically perform slower than tools that operate at a lower level of abstraction.

9 Challenges

Some challenges I faced were the long synthesis times and understanding the structure of the FPGA memory. I was stuck with optimizing the load step for a long time since I thought that the local memory was lowering the initiation interval whenever I unrolled more than 4 times. Sometimes I would try to synthesize a loop that would be too large to fit on the FPGA and the synthesis would hang. I was tempted to run a 24x instance to develop faster, but I did not want to run out of AWS money. In the end my performance was limited by synthesis time and computing resources, as I did not have enough time or money to generate a very fast FGPA.