

Parallelization and optimization strategies:

1. P loop: I applied “#pragma HLS unroll factor=5” to fully unroll the innermost loop since computing the product of the corresponding weight and input element are independent. However, there exist a dependency of first loading the original value then updating. I believe this is addressed in the optimization using tree reduction.
2. Q loop: Same as P loop, I applied “#pragma HLS unroll factor=5” to fully unroll the loop since no dependency exist except the tree reduction to update.
3. W loop: I applied “#pragma HLS pipeline II=1” to pipeline the w loop such that consecutive output elements update can be pipelined.

Strategy difference from lab3:

1. The major strategy difference is the use of pipelining here. Specifically, when writing CUDA code, similar level of parallelism is achieved through different threads. However, CUDA implements a warp level pipelining while the FPGA allows a more fine-grained loop iteration level pipelining.
2. In lab3, Grid dimension is designed such that I have 256 output dimensions scattered through the z dimension of grid. Then for each individual grid block I would have $7 * 7$ blocks of $16 * 16$ threads each for one output element. In this case, warps of same block are assigned to different SM by warp scheduler and pipelining is implemented through executing ready warps when current warp is stalled. There is a degree of uncertainty here that user can't control in terms of how pipelining is achieved.
3. In lab4, for a given output tile, we traverse through its width elements in order when implementing pipelining, giving us the exact control on how pipelining can be achieved.
4. On top of the fine-grained control of the warps, the FPGA also imposes the resource limitation on the number of DSPs/BRAMs to unroll in each pipelined stage. I tried to pipeline at h and w loop respectively, the resource utilization for h loop seems to exceeds the resource available, so I eventually decided to pipeline at w loop.

FPGA resource usages:

DSP was used most extensively in terms of percentage.

- BRAM: 49%. Total: 2152
- DSP: 58% Total: 4004
- FF: 23% Total: 544957
- FUT: 28% Total: 336496
- URAM: 0% Total 0

Incremental evaluation:

1. Baseline

$$\text{throughput} = 41104179200/7323752913 = 5.612$$

1. unrolling p, q, default partition, this allows parallel execution of convolution computation, though data dependency still exist and memory conflict for input still exist.

$$41104179200/6632937938 = 6.197$$

2. unrolling p, q, partition p, q = 5. This new array partition avoided memory contention for input element for each of the pq iteration.

$$41104179200/3344603602 = 12.289$$

3. unrolling p, q, partition p, q complete. This allows more ports concurrently reading and writing weights, though it didn't improve performance.

$$41104179200/3344603602 = 12.289$$

4. unrolling p, q, partition p, q complete, output dimension 3 = 1. This step is trying to shift optimization focus from across different width element to true parallelization within in the convolution filter for one element.

$$41104179200/3355007442 = 12.251$$

5. unrolling p, q, partition p, q complete, output dimension 3 = 1, input dimension 2/3 = 5

This step is trying to make concurrent access for each of the input element in convolution filter truly parallel

$$41104179200/3355011538 = 12.251$$

This step takes significantly longer. (129 S compares to usual 60 S)

6. unrolling p, q, partition p, q complete, output dimension 3 = 1, input dimension 2/3 = 5, unrolling w loop by factor 2. This step is trying to parallelize computation of convolution filter across neighboring elements. It didn't help because there is still intrinsic memory access conflict for input elements.

$$41104179200/3355015634 = 12.251$$

Elapsed time = 131

7. unrolling p, q, partition p, q complete, output dimension 3 = 1, input dimension 2/3 = 5, unrolling w loop by factor 4. This was trying to figure out why unrolling w didn't increase performance

41104179200/3355040210 = 12.251

Elapsed time = 336

8. unrolling p, q, partition p, q complete, output dimension 3 = 1, input dimension 2/3 = 5, unrolling w loop by factor 4. Permuting H and W loop. This was trying to avoid memory conflict in computing convolution filter since different row access partially different input elements. Didn't turn out well

41104179200/3355040210 = 12.251

Elapsed time = 189.33

9. unrolling p, q, partition p, q complete, output dimension 3 = 1, input dimension 2/3 = 5, unrolling w loop by factor 5. Permuting H and W loop, same as above.

41104179200/3355040210 = 12.251

Elapsed time = 184.33

10. unrolling p, q, partition p, q complete, output dimension 3 = 1, input dimension 2/3 = 5, unrolling w loop by factor 5. Permuting H and W loop, used accumulator for output[i1][h][w]. This approach is trying avoid the data dependency when computing one single output element.

41104179200/3355011538 = 12.2515761077

11. unrolling p, q, partition p, q complete, output dimension 3 = 1, input dimension 2/3 = 5, used accumulator for output[i1][h][w], pipelined w, made h,w iteration's needed input register. This approach though didn't fix the latency problem, it laid foundation for successfully transferring needed input element into local register with unlimited read/write ports to avoid memory conflict.

Notable time difference: 46.02 S

41104179200/4734708178 = 8.681

12. unrolling p, q, partition p, q complete, output dimension 3 = 1, input dimension 2/3 = 5, used accumualtor for output[i1][h][w], pipelined w, made h,w iteration's needed input register, permuted the i1 loop

Elapsed time: 259.51 S

$$41104179200/3447450066 = 11.92$$

13. unrolling p, q, partition p, q complete, output dimension 1 = 16, input dimension 2/3 = 5, used accumualtor for output[i1][h][w], pipelined w, made h,w iteration's needed input register, permuted the i1 loop. This approach drastically increased performance since now I am able to fully parallelize the output element update across parallel output dimension since memory conflict is no longer the issue

Elapsed time: 88.98 S

$$41104179200/349956562 = 117.46$$

14. unrolling p, q, partition p, q complete, output dimension 1 = 16, input dimension 2/3 = 5, used accumualtor for output[i1][h][w], pipelined w, made h,w iteration's needed input register, permuted the i1 loop, unrolled w loop by 4. Further parallelization

Elapsed time: 160.21 S

$$41104179200/256371154 = 160.33$$

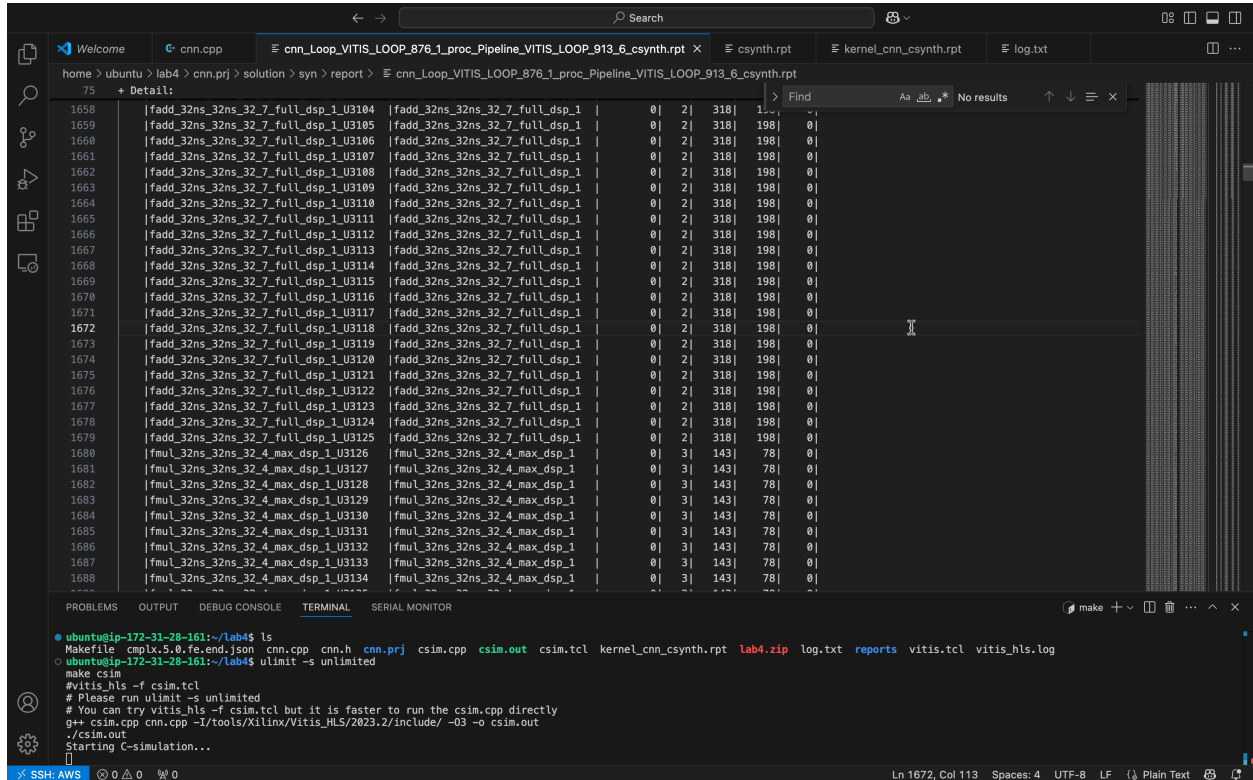
15. unrolling p, q, partition p, q complete, output dimension 1 = 16, input dimension 2/3 = 5, used accumualtor for output[i1][h][w], pipelined w, made h,w iteration's needed input register, permuted the i1 loop, unrolled w loop by 8. This approach had the problem of overutilizing DSP beyond 100% (107%) so wasn't adopted.

$$41104179200/209152466 = 196$$

Bonus:

DSP: The DSP matches my expectation, I unrolled p, q, i fully, and I unrolled w by 4. This case it would be $5 * 5 * 16 * 2 = 800$ operations. By looking at the report for my concurrent loop, each adder needs 2 DSP and each multiplication needs 3 DSP. Thus in total $800 * 5 = 4000$ DSP that I currently use.

BRAM: input = $1 * 228 * 228 * 4 / 18k$ bits = 90 blocks. Output = $16 * 224 * 224 * 4 / 18k$ bits = 1394 blocks. Weight = $16 * 256 * 5 * 5 * 4 / 18k$ bits = 178 blocks. Total: 1662 blocks.



The screenshot shows a code editor with a Verilog HDL file named `cnn_Loop_VITIS_LOOP_876_1_proc_Pipeline_VITIS_LOOP_913_6_csynth.rpt`. The code is a Verilog module for a convolution layer, featuring a loop for processing 1680 elements. The code is divided into two main sections: a loop for processing 1680 elements (lines 1658-1688) and a loop for processing 1680 elements (lines 1689-1714). The code uses a 32-bit data path and a 16-bit multiplier. The code is written in Verilog HDL and is a synthesis report for a Vitis project. The terminal window shows the command `make` being executed, and the output shows the synthesis process. The terminal output includes the command `make` and the output `make csim`. The output also shows the command `#vitis_hls -f csim.tcl` and the output `# Please run ulimit -s unlimited`. The output also shows the command `# You can try vitis_hls -f csim.tcl but it is faster to run the csim.cpp directly` and the output `g++ csim.cpp cnn.cpp -I/tools/Xilinx/Vitis/2023.2/include/ -O3 -o csim.out`. The output also shows the command `./csim.out` and the output `Starting C-simulation...`. The terminal window also shows the command `ls` and the output `Makefile comp1x.5.0.fe.end.json cnn.cpp cnn.h csim.out csim.tcl kernel_cnn_csynth.rpt Lab4.zip log.txt reports vitis.tcl vitis_hls.log`. The terminal window also shows the command `ulimit -s unlimited` and the output `make csim`. The terminal window also shows the command `#vitis_hls -f csim.tcl` and the output `# Please run ulimit -s unlimited`. The terminal window also shows the command `# You can try vitis_hls -f csim.tcl but it is faster to run the csim.cpp directly` and the output `g++ csim.cpp cnn.cpp -I/tools/Xilinx/Vitis/2023.2/include/ -O3 -o csim.out`. The terminal window also shows the command `./csim.out` and the output `Starting C-simulation...`.