Jason Lai
204995126
DIS 1A

CS133 Lab 5 Report

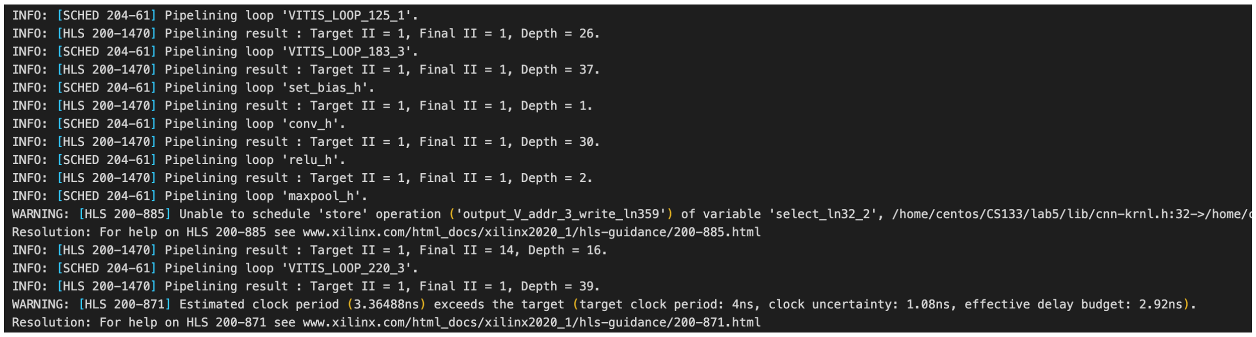
Performance Summary:

Version / Trial	1	2	3
Sequential (Lab 4 Result)	15.9017 s 10.3396 GFlops	15.8858 s 10.3499 GFlops	15.8639 s 10.3642 GFlops
Parallel (FPGA) absolute max latency 164.4 / absolute max latency	0.629 s 261.367249603 GFlops		
Speedup	15.8838 / 0.629 ≈ 25.25x speedup		
Performance Improvement	261.367249603 / 10.3512333333 ≈ 25.25x more Flops		

Here, I assume that the absolute max latency from the readout of `make estimate` is the total execution time of the CNN kernel run on the FPGA. From the table above, we see that the performance of my OpenCL implementation with a Xilinx Ultrascale+ VU9P FPGA is approximately 261 GFlops with an execution time of 0.629 seconds. Averaging the results of the trials for the sequential program, we see that the speedup of the parallel OpenCL implementation is approximately 25.25x over the sequential implementation. As for the performance, we see that the parallel implementation performs approximately 25.25x more floating point operations per second than the sequential implementation.

Parallelization and Optimization Strategies:

My parallelization and optimization strategies for this lab were mostly limited by lack of understanding of how exactly the FPGA works concerning the memory layout and computation units with the various resources such as BRAMs, LUTs, and DSPs. So, my strategy for performing parallelization and optimizations was mostly trial and error while leveraging the `cnn-krnl.rpt` and `vitis_hls.log` files to see where bottlenecks were occurring (in the .rpt) and looking at probably causes for these bottlenecks (in the .log). My main objective in my strategy was to lower the initiation interval for the various loops in the convolution network in order to hide as much of the latency of the consecutive computations as possible when using the `#pragma HLS pipeline` and `#pragma HLS array_partition` pragmas to pipeline the loops.



```
INFO: [SCHED 204-61] Pipelining loop 'VITIS_LOOP_125_1'.
INFO: [HLS 200-1470] Pipelining result : Target II = 1, Final II = 1, Depth = 26.
INFO: [SCHED 204-61] Pipelining loop 'VITIS_LOOP_183_3'.
INFO: [HLS 200-1470] Pipelining result : Target II = 1, Final II = 1, Depth = 37.
INFO: [SCHED 204-61] Pipelining loop 'set_bias_h'.
INFO: [HLS 200-1470] Pipelining result : Target II = 1, Final II = 1, Depth = 1.
INFO: [SCHED 204-61] Pipelining loop 'conv_h'.
INFO: [HLS 200-1470] Pipelining result : Target II = 1, Final II = 1, Depth = 30.
INFO: [SCHED 204-61] Pipelining loop 'relu_h'.
INFO: [HLS 200-1470] Pipelining result : Target II = 1, Final II = 1, Depth = 2.
INFO: [SCHED 204-61] Pipelining loop 'maxpool_h'.
WARNING: [HLS 200-885] Unable to schedule 'store' operation ('output_V_addr_3_write_ln359') of variable 'select_ln32_2', /home/centos/CS133/lab5/lib/cnn-krnl.h:32->/home/
Resolution: For help on HLS 200-885 see www.xilinx.com/html_docs/xilinx2020_1/hls-guidance/200-885.html
INFO: [HLS 200-1470] Pipelining result : Target II = 1, Final II = 14, Depth = 16.
INFO: [SCHED 204-61] Pipelining loop 'VITIS_LOOP_220_3'.
INFO: [HLS 200-1470] Pipelining result : Target II = 1, Final II = 1, Depth = 39.
WARNING: [HLS 200-871] Estimated clock period (3.36488ns) exceeds the target (target clock period: 4ns, clock uncertainty: 1.08ns, effective delay budget: 2.92ns).
Resolution: For help on HLS 200-871 see www.xilinx.com/html_docs/xilinx2020_1/hls-guidance/200-871.html
```

My methodology in going about this strategy was to apply incremental optimizations, such as placing pragmas in various parts of the loops, and then executing `make estimate` to see the performance and view the report to see where the bottlenecks were. Furthermore, when the program is compiling after running `make estimate`, the `vitis_hls.log` file logs the steps that the HLS tool took as it was generating the kernel. Some of the more interesting points in the log are when certain loops are flattened and also when pipelining of the loops occurs, which can be seen in the image above. To make reading the output easier, I labeled each of the loops in the convnet program to something like `conv_h` in order to know exactly which loops the log is talking about. In this pipelining stage, we see the initiation interval reported, as well as any warnings that indicate why our initiation interval for a given loop is not 1. In the warnings, the HLS tool reports which operation and what data structure is causing problems with the pipelining, which allows me to try and make adjustments, either by moving pragmas around or by applying some partitioning to an array.

On top of trying to get the initiation intervals to be as low as possible all across the board, I also tried to speed up the sequential execution of the program by incrementally applying optimizations, such as `#pragma HLS unroll`, loop permutations, reduction, and a sliding window. However, I experienced very minimal success with this, as the only optimization that was helpful in my implementation in the end was the sliding window.

Performance Evaluation:

To evaluate the performance of the optimizations I applied, I used my highest performing code as a baseline and simply removed each optimization individually to see what kind of impact it would have on my most optimized code. Some of these optimizations come with caveats e.g. for the sliding window optimization, I also had to remove some partitioning in order for the program to synthesize in a reasonable amount of time. The table below shows the performance of my highest performing code (under **Baseline**), and for the optimizations under it, the **Performance** column shows the decrease in performance observed when that optimization **is not** utilized.

Baseline (Most Optimized Code)	
0.629 s 261.367249603 GFlops	
Optimization	**Performance**
Sliding Window	35.538 s 4.62603410434 GFlops
Remove Reduction	0.636 s 258.490566038 GFlops
Tiling Size Adjustment	0.936 s 175.641025641 GFlops
#pragma HLS pipeline	540.357 s 0.30424330581 GFlops
#pragma HLS array_partition	7.720 s 21.2953367876 GFlops

Sliding Window:

For this optimization, I utilized a sliding window (2D array) with kKernel rows and kTileH + kKernel - 1 columns, so that it can store from the `input` all of the elements needed to output an entire row of the `C[h][w]` tile after each element is filtered (multiplied by `weight`) and reduced. It is partitioned completely in all dimensions (dim=0), so it's a collection of local registers; the `weight` array is also partitioned on its third dimension by a factor of 5 and the `input` array is partitioned on its second dimension by a factor of 5 and its third dimension completely. The optimization itself involves me first calling an inline function that populates the sliding window on each iteration of `j`, so as to initialize it with the appropriate values from the `input` array for a given channel `j`. By storing the elements of the `input` array that we need for an entire row of `C[h][w]`, we improve the locality of caching the input values that we need, as for each element of `C[h][w]` in the *first* row that we compute, we only need one additional column of 5 elements of the filter, whereas the other 20 can be reused from the last `C[h][w]` element that we computed. Furthermore, when we compute the next row of `C[h][w]`, we only discard the topmost row of the window and reuse the rest of the window, so we really only need to read in one additional value from the `input` array to compute an element of `C[h][w]`. From the above table, we can see that this increases our performance significantly as there is almost a 57x increase in floating point operations per second computed.

Remove Reduction:

For this optimization, it was more of a rollback of a previous optimization I applied which is reduction. Originally, I constructed a 1D array partitioned as registers and for each 5 x 5 filter we apply to input and reduce to get an element of `C[h][w]`, I took the 25 computed values and stored them in the reduction array. After the loops `p` and `q` finish a full 5 x 5 tile, I then reduce the values stored in the reduction array using a tree structure; this is done using inline, pipelined functions that are simply unnested, unrolled for loops. To perform the reduction, I first reduce the latter 24 elements in the array by adding half the elements (add two elements across *n* elements) with the other half of the pertinent elements with each function call (as the professor showed in lecture. Once I'm down to 4 relevant elements, I then call a different function to reduce the last 4 elements by halving, except now we include the element at index 0, instead offsetting by one element. Once I have the sum computed at index 0, I set `C[h][w]` to this value. However, although I initially saw a performance improvement when I was first using this optimization, I observed that this optimization was actually hurting my performance as I was performing tests for this report. In the table, we can see that the difference in performance isn't as significant relative to the other optimizations, as it is only a few GFlops difference, however, the difference was noticeable enough to warrant rolling back the reduction optimization. I suspect that the difference in performance is that the control overhead or actual computation of the reduction is not actually much faster than simply adding to `C[h][w]` directly in this scenario, despite using local registers to reduce access to a single cycle.

Tiling Size Adjustment:

For this optimization, I simply adjusted the size of the tile values `kTileH` and `kTileW` to 112 and 56 respectively. This is an increase from the values initially set to `kTileH` and `kTileW` in the comments at the top of the file: 28 and 56 respectively. From my understanding, increasing the tile size can improve performance up to a point, because as I pipeline the `h` loop of the convolution layer, less resources are wasted on control, and we execute longer runs of pipelined computations due to the deep pipeline enabled by the larger loop bound. We can see from the results above that this optimization improves our performance somewhat, as we achieve nearly a 1.5x boost in performance.

#pragma HLS pipeline:

For this optimization, I placed `#pragma HLS pipeline` inside of the loops I want to pipeline. What this effectively does is pipeline the execution of the loop's body, which also comes with the caveat that any nested loops are unrolled in the process. I placed this pragma at the `h` loop level of each level of the convnet program. While some loops had nominal improvement after viewing the `cnn-krnl.rpt` file to see the effect of these pragmas, I found the convolution layer loop, which has been the bottleneck throughout labs 3, 4, and 5, was drastically improved by the use of pipelining. I surmise that this is due to the fact that a significant amount of computation occurs in this loops, so a deep pipeline helps hide the latency by overlapping the computation, which is compounded by the fact that I managed to have nearly all of the loops in my program have an initiation interval of 1, through partitioning. From the table above, we can see that this optimization provides the greatest improvement to performance, as the increase is nearly 858x over the program without the implementation.

#pragma HLS array_partition:

For this optimization, I partitioned my sliding window array into registers, so that accesses to it are achieved in a single cycle. I also partitioned the `bias` and `weight` arrays, as well as the `input` array on two dimensions, in order to improve the initiation interval of the respective loops that they are used in by overcoming the hardware limitations of BRAM. That is, because BRAM is limited by only two reads at a time, or one read and one write, this can hurt our pipelining performance, as our OpenCL implementation will require multiple accesses in order to perform computations. So, by using partitioning, we can utilize more BRAMs, to increase the number of accesses available to us for these resources and drop the initiation interval for certain loops. It's also worth noting that the HLS tool provides some optimizations automatically, and I observed in the `vitis_hls.log` file that the tool automatically partitioned the `C[h][w]` array that we store the tile in when I placed a pipeline pragma in the `h` loop of the convolution layer, which is helpful because I had difficulty trying to partition it manually. From the table above, we see that this optimization provides about a 12x boost to our performance, which is very good.

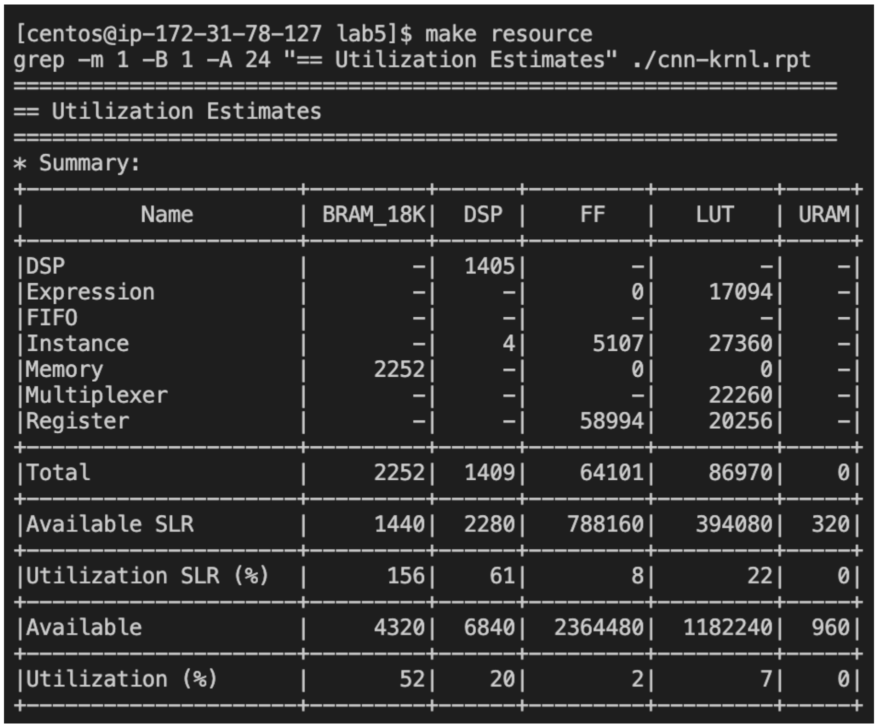
How my Strategies Differ from Lab 3 and Lab 4:

To reiterate, since designing the FPGA has so much freedom and I don't understand it nearly as well as I did OpenCL, I implemented optimizations and parallelization very conservatively, but mostly got lucky that I found optimizations that work (such as good pipeline pragma placements, etc). My strategies for this lab differ from Lab 3 and Lab 4, in that I mainly focused on improving the code's caching performance with the sliding window and pipelining the loops as much as possible in order to hide as much latency of the computation as possible. This is in contrast to my strategies for Lab 3 and 4, where I focused on how to balance the use of computation units and my work group / work item assignments, while also improving the basic performance of the sequential code using loop transformations, vectorization, etc.

This difference in approach to strategy mostly come from the fact that in the GPU and CPU case when using OpenCL, the architecture of the hardware is fairly rigid in that there is little to no opportunity to change the hardware to fit our needs; rather, we must somehow map our problem and solution to the architecture of the hardware in order to achieve good resource usage and parallelization. On the other hand, creating OpenCL implementations for the FPGA is much more freeform, wherein we are designing how we want the program to be parallelized on the FPGA, so the strategy changes from trying to make our program fit the hardware, to making the hardware into suitable configuration to accept our program.

In the end, my performance was nowhere near as fast as my Lab 4 OpenCL implementation on the Nvidia Tesla M60 GPU, and my performance was only somewhat better than the performance I achieved with an OpenCL implementation using 4 cores (8 threads) of a CPU. If this is an indication of anything, it is that my strategy for FPGA acceleration did not leverage the capabilities of the FPGA enough to get anywhere near the full potential of its computational power. I knew it was going to be a difficult task and I'm happy with what I achieved, but I do think that a good implementation of FPGA acceleration would have performance much closer to that of the GPU.

<u>FPGA Resource (LUT/FF/DSP/BRAM) Usage:</u>



```
[centos@ip-172-31-78-127 lab5]$ make resource
grep -m 1 -B 1 -A 24 "== Utilization Estimates" ./cnn-krnl.rpt
================================================================
== Utilization Estimates
================================================================
* Summary:
+---------------------+---------+------+---------+---------+-----+
|         Name        | BRAM_18K| DSP  |    FF   |   LUT   | URAM|
+---------------------+---------+------+---------+---------+-----+
|DSP                  |        -|  1405|        -|        -|    -|
|Expression           |        -|     -|        0|    17094|    -|
|FIFO                 |        -|     -|        -|        -|    -|
|Instance             |        -|     4|     5107|    27360|    -|
|Memory               |     2252|     -|        0|        0|    -|
|Multiplexer          |        -|     -|        -|    22260|    -|
|Register             |        -|     -|    58994|    20256|    -|
+---------------------+---------+------+---------+---------+-----+
|Total                |     2252|  1409|    64101|    86970|    0|
+---------------------+---------+------+---------+---------+-----+
|Available SLR        |     1440|  2280|   788160|   394080|  320|
+---------------------+---------+------+---------+---------+-----+
|Utilization SLR (%)  |      156|    61|        8|       22|    0|
+---------------------+---------+------+---------+---------+-----+
|Available            |     4320|  6840|  2364480|  1182240|  960|
+---------------------+---------+------+---------+---------+-----+
|Utilization (%)      |       52|    20|        2|        7|    0|
+---------------------+---------+------+---------+---------+-----+
```

Resource	Resource Count	Usage
BRAM_18K	2252	52%
DSP	1409	20%
FF	64101	2%
LUT	86970	7%

In the table above and the provided readout image, we see that the resource that has been used most in terms of percentage is BRAM, as it’s usage is 52%. This is probably due to the fact that I partitioned several arrays cyclically, with some having high factors, so that I use quite a few BRAM resources in order to lower the initiation interval of pipelined loops in my program.