Lab 5 Report
Dennis Zang - 704766877
(Vitis)

Please explain the parallelization and optimization strategies you have applied. Include the directives (if any) or code segments you have added to achieve this. Evaluate the performance of each parallelization/optimization that you have incrementally applied and explain why it improves the performance. Also, explain how your strategy differs from your Lab 3 CPU and Lab 4 GPU parallelization/optimization strategy and why you chose to apply a different strategy.

In general, I have applied four parallelization strategies:

1) loop unrolling to force parallelism
   The directive I mainly used here is "__attribute__((opencl_unroll_hint(x)))", where x is the number of iterations to unroll. However, note in some cases I manually unrolled the loop myself to better emphasize loop iteration independence for some cases (as the compiler can simply ignore the directive in some cases). This optimization is especially important in that it is the main optimization done on the convolution loop of the program, and has increase the execution speed by a factor of about 2 in my case. This in mainly because, assuming no inter-dependencies between different loop iterations, in hardware, this can be implemented as having n FPGA modules executing in parallel, where n is the number of loop bodies in the unrolled loop.
2) pipelining loop iterations
   The directive I mainly used here is "__attribute__((xcl_pipeline_loop))". This optimization is mainly meant to pipeline loop iterations that have been unrolled (as we know that hardware resources are duplicated to fit multiple iterations). In cases in which the number of loop iterations is far greater than the iterations unrolled, we would need to pipeline in order to make use of the most parallelism (multiplexing inputs to unused resources that are finished with the previous iteration).
3) managing memory representation of arrays
   The directive used to reshape the array representation
4) vectorization
   Like in labs 3 and 4, I made use of vectorization again. Although it didn't make to much of a difference compared to the non-vectorized program (it made a small marginal difference), I was then able to unroll the outer loops more efficiently. (This is done mainly to the small loops of size kKernel.)

The strategy I used in this lab differs greatly from labs 3 and 4, since the code we are to write is meant to be translated directly into hardware. In other words, we need to consider how the code will be run asynchronously (likely in parallel), and how the dependencies between each statement can affect the FPGA configuration generated.

Please report the FPGA resource (LUT/FF/DSP/BRAM) usage, in terms of resource count and percentage of total. Which resource has been used most, in terms of percentage?

Name	BRAM_18K|	DSP48E	FF	LUT	URAM|
DSP	-	1	-	-	-
Expression	-	0	0	33468	-
FIFO	-	-	-	-	-
Instance	30	1000	104742	65077	-
Memory	912	-	0	0	-
Multiplexer	-	-	-	18088	0
Register	-	-	124715	9864	-
Total	942	1001	229457	126497	0
Available SLR	1440	2280	788160	394080	320
Utilization SLR (%)	65	43	29	32	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	21	14	9	10	0

In terms of percentage of total, BRAM has been used the most.

(Merlin)
Please explain the parallelization and optimization strategies you have applied. Include the directives (if any) or code segments you have added to achieve this. Evaluate the performance of each parallelization/optimization that you have incrementally applied.

In general, I have applied two parallelization strategies (similar to Vitis):
1) loop unrolling to force parallelism
   The directive I mainly used here is "#pragma ACCEL parallel factor=x", where x is the number of iterations to unroll.
2) pipelining loop iterations
   The directive I mainly used here is "#pragma ACCEL pipeline". This optimization is usually done on large loops automatically, so here it is really just for emphasis.

Please report the FPGA resource (LUT/FF/DSP/BRAM) usage, in terms of resource count and percentage of total. Which resource has been used most, in terms of percentage?
This information can be found in the SDAccel HLS report:
   build/xilinx_mo/.merlin_prj/run/report/CnnKernel_csynth.rpt

Hierarchy	LUT	FF	BRAM	DSP	URAM	Detail
CnnKernel (CnnKernel.cpp:10)	191247 (16%)	199639 (8%)	2538 (58%)	236 (3%)	0 (~0%)	-

Again, like in Vitis, BRAM is used the most.

Please analyze Merlin log file (build/xilinx_mo/merlin.log), Merlin report file (build/xilinx_mo/merlin.rpt),
and SDAccel HLS report
(build/xilinx_mo/.merlin_prj/run/report/CnnKernel_csynth.rpt).
Try to find the optimization Merlin has performed on your code. Explain why you reached this conclusion. Explain how such optimizations would increase the performance of your kernel.

The main optimization Merlin has performed on my code is inferring memory bursts. One thing I have noticed when comparing the read cycles between the two workflows is that the read cycle percentage is lower in Merlin than in Vitis (in addition to the "INFO" logs Merlin outputs regarding inferring memory bursts). Such an optimization would allow for better throughput of data, as less time is spent on data transfer and is now more heavily influenced by loop unrolling and pipelining.

If you have done both of the flows, in order to get the score for both, you should also include (otherwise, you only get the report score for one flow):
Please concentrate on making comparisons with the Vitis version. That is, if the same strategy was applied as your Vitis version, please mention this in your report and keep the description very brief. If the coding style (or directive) has changed from Vitis version, you will need to show the difference. For the newly applied optimizations, please elaborate and explain why it improves the performance. If some optimizations were removed, please mention this in your report.
Compare the performance of Vitis flow and Merlin flow. Which one is better? Why? Does the result meet your expectation? If not, please discuss possible reasons.

In both cases, I have applied mostly the same strategies for parallelization. The main difference between the two is that the Vitis version is vectorized (as it is in OpenCL). In addition, we don't need to specify pipelining for large loops with small unrolls in Merlin.
In terms of performance, Vitis was better. This didn't completely meet my expectations, as if Vitis weren't vectorized, then Merlin may have been faster due to memory burst optimizations.