October 23, 2020

CMU 18-643: Reconfigurable Logic: Technology, Architecture and Applications

Handout #9/Lab 4: From HPC to Hardware Accelerator (200 points)

Due Monday, 11/9/2020 noon

This lab can be completed individually or in a team of 2.

In this lab, you will approach FPGA acceleration from an HPC (high performance computing) programming starting point.[1] You were given in Lab 1 a valid OpenCL implementation of the CNN algorithm (as a single work item).[2] Although the OpenCL language and compiler can shield you from the low-level hardware datapath details, a performance tuning discipline—based on understanding and exerting control over how/when/where compute, data buffering, and data movement take place—is nevertheless necessary to achieve performance. This is true whether you are using Verilog, Vivado HLS, or OpenCL to develop on FPGA; this is true whether you are developing for performance on FPGA, GPU, or any spatially concurrent platform.

This lab is again in part graded on a competition. Be prepared and start thinking early. Please feel free to post questions and answers on 18643's Piazza page to help each other out with tools related issues. Keep your good ideas to yourself.

We want to thank Intel Mindshare Curriculum Program for supporting the development of this lab exercise and the corresponding lecture materials[3]. We want to thank Intel DevCloud: (https://software.intel.com/en-us/devcloud/FPGA) for providing access and support to 18-643 students.

Part 1: Getting started

Download lab4.zip from Canvas.[4] Run through the instructions from Handout #5 again (substituting "lab4" for "lab1" where appropriate) to re-familiarize yourself with the Intel DevCloud (Arria 10 1.2.1) and Intel FPGA SDK for OpenCL. Build and run the starter code in lab4/cnn/ (identical to lab1/cnn/). Note the performance.

The directory lab4/cnn_ref/ contains a precompiled bitstream and executable of a reference solution (provided without source code). To try it out, execute from lab4/cnn_ref/,

```
$ECE643/scripts/run.sh lab4_ref
```

Note the difference in performance.

Download *Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide* (BPG for short) from https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf. Read the table of contents to see the topics addressed. More than a language reference, BPG is instructive on high performance design thinking. Chapter 3 is a must read for this lab. Read the rest if you are motivated.

---

[1] Intel FPGA SDK for OpenCL also supports a kernel coding style with a strong structural design flavor, e.g., describing concurrent subblocks connected using registers and channels. Please see Aydonat, et al., *An OpenCL Deep Learning Accelerator on Arria 10*, Proceedings of ISFPGA, 2017 (https://dl.acm.org/doi/10.1145/3020078.3021738).

[2] Zhang, et al., "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," Proceedings of ISFPGA, 2015 (http://dl.acm.org/citation.cfm?id=2689060).

[3] http://users.ece.cmu.edu/~jhoe/course/ece643/F20handouts/L11.pdf;
http://users.ece.cmu.edu/~jhoe/course/ece643/F20handouts/L12.pdf;
http://users.ece.cmu.edu/~jhoe/course/ece643/F20handouts/L13.pdf.

[4] http://users.ece.cmu.edu/~jhoe/course/ece643/F20handouts/lab4.zip.

Unlike in structural methodologies, when expressing a computation in OpenCL, your control over the resulting hardware is mediated by a compiler. It is important to inspect how your code is interpreted in hardware and if it is as you intended. Read BPG Chapter 2. Open lab4/cnn/bin/cnn/reports/report.html with a browser. Can you use the information to explain the observed performance resulting from the starter code? Later, when attempting to improve the performance of the kernel, use the report to explain the performance and locate the bottleneck[5].

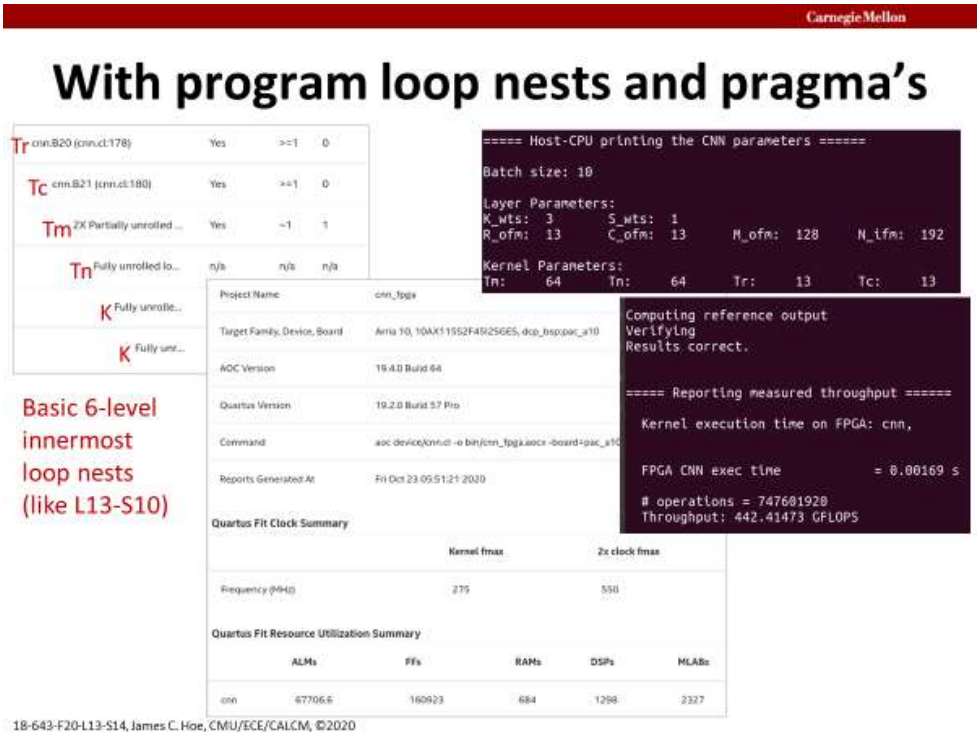
Part 2: Let's see what you can do.

The goal of this lab is to improve the throughput of computing a CNN layer (as configured in instance643.h) on a batch of inputs (by default 10). The timing measured and reported by main.cpp does not include host-accelerator data transfer overhead incurred at the start and end of the batch. Working from the starter project in lab4/cnn/, you have full flexibility to modify cnn.cl and kernel643.h by rewriting the code and adding pragmas. You can change the array layout by altering the array macros in kernel643.h. You may not add or modify other files.

Hints: Based on what you have seen in lectures and earlier labs, visualize at a conceptual level what should happen—that is, how/when/where compute, data buffering, and data movement take place. Unrolling and pipelining are *goto* optimizations for spatially concurrent execution of regular loop nests. Their high potential performance is only realizable if the memory path can keep up with the rate of consumption and production of data. To increase arithmetic intensity over costly DRAM accesses, the CNN kernel provided to you in cnn.cl is a blocked version of the canonical CNN loop nests, corresponding to Figure 9 in Zhang'2015. Unlike in Lab 3, the outer loops do not explicitly prepare local data tile buffers to be operated on by the inner loops. Nevertheless, this version—if transformed appropriately to improve access pattern and maximize data reuse—can perform acceptably with the aid of compiler-inserted load/store units and caches (BPG 3.7). Alternatively, lab4/cnn/device/CNN.cl.scratchpad provides a version of cnn.cl that makes use of explicitly managed SRAM local memory buffers (BPG 3.3).[6]

[5] What is preventing parallelization? What is causing stalls?

[6] Copy over cnn.cl with cnn.cl.scratchpad to use it.



During development, use emulation for functional debugging; don't synthesize unless you know the code is legal and correct. (You may have to substitute reduced sizing parameters during emulation to keep it from blowing up.) When synthesizing to FPGA, the OpenCL report is available as soon as the OpenCL compile pass finishes (in minutes). Use the fast compile option to shorten place-and-route time during intermediate design explorations; the result will be slightly bigger and slower (clock) by 20~40%. Remove the fast option for your final result.

Submit: Create a submission directory <team_name>_lab4 to turn in the artifacts requested. <team_name> should be a concatenation of the team members' Andrew IDs in alphabetical order connected by '_'. One member of a team should submit a single file called <team_name>_lab4.zip (that is the zip of your submission directory) through Canvas.

In a subdirectory called lab4_source, include cnn.cl and kernel643.h. Comment extensively so the intention of your program is clear.

Include a copy of the execution output run.sh.o<job-id>.

Include a report.pdf with the following. Number the sections accordingly.

1. Provide a Google drive link to a copy of the directory lab4/cnn/bin/cnn/report. The folder should be set so anyone at Carnegie Mellon with link can view.
2. Explain the major changes and optimizations employed in your final design. Explain especially how the computation and data accesses in the inner loops are organized.
3. Explain any changes and optimizations attempted but ineffective (say why?)
4. Provide a sketch of the hardware datapath resulting from your code. (Best effort.)
5. Explain how you determined the optimum tile dimensions (Tm, Tn, Tr, Tc).
6. Report the achieved FLOPS.
7. Report the average DRAM BW utilized.
8. Summarize your resource utilization.
9. Discuss what and why you would do differently if you had more time for another try.

10. Any interesting insights from working with OpenCL

The report need not be long or polished, as long as it gets the points across. Pay most attention to highlighting the changes you tried and how effective they were.

Grading: 60% of the grade is subjective based on the quality of the effort as represented by the files submitted. The other 40% is based on the throughput achieved relative to the rest of the class.

If you are in the highest (best) quartile, you will receive the full 40%. If you are in the second highest quartile, you will receive 30%. If you are in the third highest quartile, you will receive 20%. If you are in the lowest quartile AND within a factor of 2 of the lowest ranked group in the third quartile, you will receive 10%. Otherwise, you will receive 0%. An exception is that you will receive the score of the higher (better) quartile if your throughput is within 10% of the lowest throughput in that quartile.

Part 3: Do a little more?

In your kernel, the CNN parameters and kernel tile sizes are compile-time constants. Consequently, compiler analysis has an easier time deducing the fixed control flow of the static loops and the dependency (or lack of) between array accesses based on their loop indices.

You will receive 10% bonus if your solution's performance fall in the first quartile and R, C, M, N are runtime settable variables. (Set FIX_R, FIX_C, FIX_M, FIX_N to 0 in kernel643.h.) In other words, the same OpenCL compiled kernel can execute CNN layers with different feature map size parameters (R, C, M, N).[7]

The reference solution uses local memory buffers.
ne arguments, e.g., -rofm=10. See main.cpp for other settings.