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

**Handout #6/Lab 3: Hardware Accelerated Computation (250 points)**

Issued:10/9/2023

Submit by 10/30/2023 (Monday) noon for 5% bonus

Submit by 11/3/2023 (Friday) noon without penalty

This lab must be completed in a team of 3. In this lab, you will accelerate the execution of a simplified 2-layer CNN network on the Ultra96 Zynq SoC using Vitis OpenCL-based flow. Be prepared and start thinking early. Please post questions and answers on 18643’s Piazza page to help each other out with tools related issues. Keep your good ideas to yourself.

Please note that the fall break falls in the middle of the lab period. Please also note that the project proposal is due on 10/30. Please finally note that the midterm is on 10/25.

**Part 1: Getting started**

Import the Vitis archive [**lab3\_cnn\_dfx\_2021\_1.ide.zip**](https://drive.google.com/file/d/1Vo0hz7xl7uGuKtrV95y1kbCCi_joNgOT/view?usp=drive_link). It makes use of essentially the same C++ source code base you worked on in Lab 2, except:

* **\_\_VITIS\_CL\_\_** is defined in **util643.h**
* extensions to optionally allow different kernels to be used in layer 0 and layer 1 by DFX.

| This lab uses a config file to optimize the kernel-platform linking process. This requires a flag to be passed to the linker in the cnn\_system\_hw\_link project for the Hardware configuration.  The archived project should contain the flag already but to check:  In the **Explore** pane, right click the project **cnn\_system**→**cnn\_system\_hw\_link**→**C/C++ Build Settings.** Select “Settings” under “C/C++ Build”. Change the configuration at the top to “Hardware”. Now click “V++ Kernel Linker”, and in the right pane the “All Options” section should read “--config /afs/ece.cmu.edu/class/ece643/f2022/lab3/krnl\_reconnect.cfg”. If not, go to the “Miscellaneous” section, use the “Add” button in the “Other Flags” section to add “--config /afs/ece.cmu.edu/class/ece643/f2022/lab3/krnl\_reconnect.cfg” and check back in the “V++ Kernel Linker” section to see if “All Options” are appropriately updated.  Your Vitis project will work even without this configuration update, but you might see significant resource limitations without it. But note, this config file assumes your kernel is using only one memory AXI port, so DO NOT USE the MAX MEMORY PORTS or any equivalent options. You are otherwise free to effect other changes, such as clock frequency or AXI interface port width for optimization. |
| --- |

| As in Lab 1, you need to switch between the Ultra96 (for generating bitstream) and zcu102 (for SW simulation). Be sure to double check the paths and the config file flag are set correctly after switching platforms.   * Ultra96   + /afs/ece.cmu.edu/class/ece643/software/xilinxVitis/platforms/2021.1/cmu\_u96v2\_dfx\_full/sw/cmu\_u96v2\_dfx\_full/linux\_domain/sysroot/cortexa72-cortexa53-xilinx-linux for “Sysroot path”   + /afs/ece.cmu.edu/class/ece643/software/xilinxVitis/platforms/2021.1/cmu\_u96v2\_dfx\_full/sw/cmu\_u96v2\_dfx\_full/linux\_domain/rootfs/rootfs.ext4 for “Root FS”   + /afs/ece.cmu.edu/class/ece643/software/xilinxVitis/platforms/2021.1/cmu\_u96v2\_dfx\_full/sw/cmu\_u96v2\_dfx\_full/linux\_domain/image/Image for “Kernel Image” * zcu102   + “/afs/ece.cmu.edu/class/ece643/software/xilinxVitis/petalinux/2021.1/sysroots/cortexa72-cortexa53-xilinx-linux” for “Sysroot path”   + “/afs/ece.cmu.edu/class/ece643/software/xilinxVitis/petalinux/2021.1/rootfs.ext4” for “Root FS”   + “/afs/ece.cmu.edu/class/ece643/software/xilinxVitis/petalinux/2021.1/Image” for “Kernel Image” |
| --- |

Import, build and run the starter Vitis project for Ultra96 using the cmu\_u96v2\_dfx\_full platform. Note the execution time and ops/sec performance reported as the baseline. (Expect to see ~0.3 Gops/sec.). \*\*To make sure all of the new goodies work, for your first build in Lab 3, you must perform a complete rebuild and reflash the SD-Card image from scratch following the instructions from Lab 1 “*Running for the First Time on the Ultra96*”.\*\*

Take a look around in <**workspace>/cnn/src/** and **<workspace>/cnn\_kernels/src/**.The default OpenCL kernel function in Lab 3, **krnl\_cnn\_layerX()** in **krnl\_cnn.cpp,** is a blocked version of the canonical CNN loop nests, corresponding to Figure 9 in Zhang’2015. In Lab 2, you focused on mapping only **cnn\_blocked\_kernel()**—the blocked inner loop kernel operating on BRAM—to FPGA. In Lab 3, the entire **krnl\_cnn\_layerX()** is synthesized in kernel-mode for the FPGA to address the full layer with input, weight, and output buffers in DRAM and invokes **cnn\_blocked\_kernel()** for processing. (**krnl\_cnn\_layerX()** was a part of the testbench in Lab 2.)

**Part 2: Let’s see what you can do.**

The goal of this lab is to improve the throughput of computing the 2 CNN layers (as configured in **instance643.h**) on a batch of **N** inputs. You can benchmark your final performance on any batch size greater than **10**; you can change **N** in **instance643.h** to change the number of inputs. The timing measured and reported by **main.cpp** include host-accelerator data transfer overhead incurred at the start and end of the batch.

In Lab 2, the same **krnl\_cnn\_layerX()** function (in **krnl\_cnn.cpp**) receives the CNN layer parameters as runtime arguments to compute layer 0 vs layer 1. This is the default behavior in Lab 3. Optionally, Lab 3 provides **krnl\_cnn\_layer0()** and **krnl\_cnn\_layer1()** (in **krnl\_cnn\_layer0.cpp** and **krnl\_cnn\_layer1.cpp**, respectively) with hardcoded layer parameters. You have the option to load these separate kernel functions one at a time by DFX. Using DFX has at least 2 advantages: (1) you can optimize the two kernels differently for the 2 layers, and (2) HLS has an easier time working with fixed-loop bounds. The main disadvantage of DFX is that you have to pay for the programming time of the second kernel. To use DFX or just to give it a try, find and uncomment “**#define ENABLE\_DFX**” in **lab3\_kernels.h**.

| It may seem like using DFX requires you to do twice the work. Keep in mind, you only have to learn how to solve the CNN design problem once and then apply it twice. Coming up with 2 separately optimized designs individually can be much easier than trying to find the best compromise.  Note: The kernel code has been organized to avoid unnecessary hardware recompilation so that only the kernel that is updated is recompiled for the FPGA. |
| --- |

**\*\*In general,** **you have full flexibility to modify the files in the cnn\_kernels/src/ directory.\*\*** You can rewrite the code and add pragmas in the .cpp and .h files in **cnn\_kernels/src/**. Very importantly, you can change the input, weight, and output array layout in DRAM by altering the array access macros **ARRAY**{**i**,**o**,**w}\_{X,0,1}** in **krnl\_cnn{, \_layer0, \_layer1}.h**. For example, you can improve DRAM access efficiency by changing the layout so the DRAM reads and writes generated by your kernel functions are to consecutive addresses.

**\*\*You may not add or modify files outside of cnn\_kernels/src/ in the final submission. If you change the number of inputs (N) for benchmarking, you will simply indicate that in the reporting.**\*\* During functional debugging, to reduce turn-around time, you may want to temporarily reduce the problem size; you may want to temporarily disable results checking during performance tuning.

**To do:**

**Step 1: Understand krnl\_cnn\_layer{X,0,1}().** This lab is primarily about managing the usage of in-fabric BRAM capacity (~1 Mbyte) and the off-chip DRAM bandwidth (~4 GB/sec best case). It is imperative that you understand thoroughly **krnl\_cnn\_layer{X,0,1}()** (based on Figure 9 of Zhang’2015). Therefore, read Zhang’2015 again carefully. Afterwards, return to your finished Lab 2 project. Build the Lab 2 source code as a regular C++ program in your favorite C++ environment. In a debugger, step through an execution to review how the code in **krnl\_cpp\_layerX()** is organized and traversed.

**Step 2: Decide on an execution strategy.** Although HLS 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 or HLS to develop on an FPGA; this is also true whether you are developing for performance on an FPGA, GPU, or any spatially concurrent platform.)

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. All basic strategies should result in the same total number of arithmetic operations (2×M×N×R×C×K×K) in a layer. Performance difference between two strategies arise mainly from their differences in (1) the number of time a data element (weights or feature map) is copied between DRAM and fabric; (2) the DRAM access pattern when transferring data elements between DRAM and fabric; (3) how data on-chip are buffered and accessed; and (4) the number of concurrent arithmetic operations per cycle when not waiting for data transfers. DFX allows you to make different tuning choices for layer 0 and layer 1, as opposed to adopting an in-between compromise.

**Step 3: Implement your strategy.**

Follow the lowest-hanging-fruit-first principle when optimizing performance. It is useful to review the concepts of latency, throughput, overhead, amortization, latency hiding, as well as the memory lectures. At each refinement iteration, first determine the performance-limiting bottleneck to be addressed next.

You may want to reconsider the design and optimization decisions of **cnn\_blocked\_kernel()** made in Lab 2. **\*\*\***Re-associate the pragmas you had on the interface arrays arguments of **cnn\_blocked\_kernel()** instead to their instantiations in **krnl\_cnn\_layerX()**.\*\*\* You can change the declaration of the arrays to better suit your needs in reshaping, partitioning, and replicating. You do not need to preserve the **cnn\_blocked\_kernel()** function boundary. The variables are named such that you can replace the call to **cnn\_blocked\_kernel()** in **krnl\_cnn()** by **cnn\_blocked\_kernel()**’s body block directly.

You want to develop your code in the most expedient way possible. This means first as a regular C++ project, then in the HLS environment, and only finally on real hardware.

* At any time, you can copy all of the .cpp and .h files from **<workspace>/cnn\_kernels/src/** and **<workspace>/cnn/src/** to a temporary directory. You can build and debug the program as C++ in the temporary directory by commenting out **\_\_VITIS\_CL\_\_** in **util643.h**.
* Once you are sure your C++ code is functionally correct, you should next attempt SW-emulation (zcu102) to double check everything is functionally correct in Vitis. To return to working in Vitis, reintroduce the .cpp and .h files back to the workspace and remember to restore **\_\_VITIS\_CL\_\_** in **util643.h**.
* After successful SW-emulation, set the build configuration to Hardware and select the Ultra96 platform. If the kernel is not already built, first in the **Assistant** pane, right click on **cnn\_system**→**cnn\_kernels**→**Hardware** and select **Build** to perform only C-to-RTL synthesis. (This shouldn’t take very long.) Once the kernel is built, in the **Assistant** pane, right click on the kernel you want to optimize (e.g., **cnn\_system**→**cnn\_kernels**→**Hardware**→**krnl\_cnn\_layerX[C/C++]**) to select **Open HLS Project** to work on optimizing the kernel in the Vitis HLS environment.
* After you are reasonably happy with the HLS results, close Vitis HLS and return to Vitis to build the complete runnable design for Ultra96 by selecting **Build Project** on **cnn\_system** in the **Explore** pane. This is a lengthy process (includes place-and-route). You want to do this as few times as possible; do it in the background while you work on something else.

| While working in Vitis HLS, you can edit the copy of **krnl\_cnn{, \_layer0, \_layer1}.cpp** that opens in Vitis HLS. When you close Vitis HLS to return to Vitis, Vitis’ copy of **krnl\_cnn{, \_layer0, \_layer1}.cpp** will be updated correctly, provided you only add pragmas to the source file not the directive file. \*\*Do not edit **krnl\_cnn{, \_layer0, \_layer1}.cpp** in Vitiswhile it is open in Vitis HLS.\*\* The **.h** files in Vitis HLS are sourced from the same files as Vitis in **<workspace>/krnl\_cnn/src** or **<workspace>/cnn/src** so there is no incoherence issue.  If you want to run C-sim and Co-sim in Vitis HLS, you need to manually add **main.cpp**, **utils.cpp**, and **cnn\_helper.cpp** from **<workspace>/cnn/src/** to the Vits HLS project as “Test Bench”. You also need to edit **<workspace>/cnn/util643.h** to comment out **\_\_VITIS\_CL\_\_** temporarily to build C-sim and Co-sim correctly. If you are using DFX, you will work on one layer at a time in Vitis HLS. When working on one layer, e.g., **krnl\_cnn\_layer0.cpp**, you need to add **krnl\_cnn\_layer1.cpp** as testbench. (If you are confused by how to run C-sim and Co-sim as explained here, there is not a real need to do it. Develop and debug the functionality of your C-code using g++ and gdb.) |
| --- |

| Outside of Lab 3 (for fun), on your own, you can try repeating this design task using RTL.  You have a few options. (1) Use Vitis host and HLS kernel for processor/fabric interaction through AXI and introduce RTL as “black boxes” that operate against local signal and SRAM buffers (see [Adding RTL Blackbox Functions - 2021.1 English](https://docs.xilinx.com/r/2021.1-English/ug1399-vitis-hls/Adding-RTL-Blackbox-Functions)). (2) Use Vitis host to invoke an RTL kernel that implements compliant AXI-based kernel interfaces (see [RTL Kernel Wizard](https://docs.xilinx.com/r/2021.1-English/ug1393-vitis-application-acceleration/RTL-Kernel-Wizard)). (3) Begin work using the HLS flow and finish the work by fine-tuning the HLS generated RTL (see [Exporting the RTL Design](https://docs.xilinx.com/r/2021.1-English/ug1399-vitis-hls/Exporting-the-RTL-Design)). |
| --- |

**Submit:** Create a submission directory **<team\_name>\_lab3/** to turn in the artifacts requested. <team\_name> should be a concatenation of the team members’ AndrewIDs in alphabetical order connected by ‘\_’. One member of a team should submit a single file called **<team\_name>\_lab3.zip** (that is the zip of your submission directory) through Canvas.

Include in the submission directory a Vitis project archive **lab3\_cnn\_submit.ide.zip**.

Include a **report.pdf** with the following. Number the sections accordingly. When appropriate, please expand to answer separately for layer 0 and layer 1 if you are using DFX.

1. Are you using DFX? (yes or no).
2. Draw a block diagram of the synthesized kernel datapath.
3. Explain your final execution strategy (how it works? why did you choose to do it this way?)
4. Discuss any ideas you tried but not adopted for the final implementation.
5. Report your runtime and ops/sec (as reported by the test wrapper) for the default 10 inputs and the number of inputs the final performance is based on.
6. Find and summarize the fabric resource utilization.
7. Separately for layer 0 and layer 1 (even if not using DFX), analyze and report the following:
   1. The average (that is, total value for the batch divided by **N**) number of times a data value is transferred between DRAM and fabric---count separately DRAM reads and DRAM writes and separately for weights, input feature maps and output feature maps. (6 numbers for each layer.)
   2. The average number of arithmetic operations performed for each data value transferred to or from DRAM. (This is arithmetic intensity.)
   3. The average DRAM bandwidth (read and write together) utilized.
8. Discuss the difference between the estimated peak performance of **cnn\_blocked\_kernel()** in Lab 2 vs in Lab 3. What motivated the design changes, if any, from Lab 2 to Lab 3 (e.g., re-tuned **Tm**, **Tn**, **Tr**, **Tc** and different pragmas) in the blocked kernel?
9. Discuss the difference between the achieved end-to-end performance in Lab 3 vs the estimated peak performance of **cnn\_blocked\_kernel()** in Lab 3. Is your overall design efficient/balanced?
10. Discuss what and why you would do differently if you had more time for another try.
11. At this moment, would you consider using Vitis for your project (yes or no)?

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.

You are also asked to complete an online form summarizing the above.

| We only ask you to report on the final design, but it behooves you to consider carefully the answers to the questions above (especially #7) for each design point you visit. It will provide important guidance on where the bottlenecks are that most need your attention. |
| --- |

**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 at your choice of **N**.

You will receive the full 40% if you achieve greater than 8x the performance of the starter project (unmodified and run at its maximum valid frequency). You will receive 30% if you achieve greater than 4x the performance of the starter project. You will receive 20% if you achieve greater than 2x the performance of the starter project. You will receive 10% if you achieve greater than 1.5x the performance of the starter project. Otherwise, you will receive 0%.