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

**Handout #5/Lab 2: Vitis IP-Flow HLS (200 points)**

Issued: 9/25/2023

Due: 10/9/2023 noon

This lab must be completed in a team of 3. This lab will give you a crash course on Vitis high-level synthesis (HLS). You will work through some tutorials and then try your own hands at developing the compute kernel for a convolutional neural network (CNN) layer. There is a lot to learn about HLS in a short time. Be resourceful and try your reasonable best in the time allowed. If you start on the last day, you are not giving yourself a chance. Performance does matter in this lab.

Please post questions and answers on 18643’s Piazza page to help each other out with tools related issues.

**Part 1: Going for a Quick Spin**

Work through the HLS portion of the Vitis tutorial ([Vitis HLS](https://xilinx.github.io/Vitis-Tutorials/2021-1/build/html/docs/Getting_Started/Vitis_HLS/Getting_Started_Vitis_HLS.html), choose ultra96 instead of Alveo U200 in Step 7 of *Creating a Vitis HLS Project*) to check out that everything works in your Vitis HLS environment. For this part, it is okay to just run through the steps even if you don’t completely understand what is happening. Don’t forget to issue the command: “**source /afs/ece.cmu.edu/class/ece643/software/scripts/setup\_vitis21.lab.sh**” to configure your environment. Choose Ultra96 as the target board when prompted. If your environment is set up correctly and you follow the steps, everything should work as in the tutorial.

**Part 2: Taking a Closer Look**

You can find a collection of helpful information on Vitis high-level synthesis in [Vitis High-Level Synthesis User Guide (UG1399) - 2021.1 English](https://docs.xilinx.com/r/2021.1-English/ug1399-vitis-hls). Read through *Getting Started with Vitis HLS.* Skim through *Vitis HLS Hardware Design Methodology* and *Vitis HLS Command Reference* to get an idea where to find answers later on, though read closely the subsection *Optimization Techniques in Vitis HLS*.

**Part 3: Let’s see what you can do.**

Now is time to see how well the HLS magic works when you are on your own. You don’t need to be an HLS expert to do this.

**To Get Started:** Please download and unzip the [Lab 2 source files](https://drive.google.com/file/d/1QmuaBE5wNtpBBqGmciy5lF5JT9TSkqsR/view?usp=drive_link). The zip file provides a CNN layer kernel implementation based on: C. Zhang, et al., “Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks,” Proceedings of ISFPGA, 2015 (<https://dl-acm-org.cmu.idm.oclc.org/doi/10.1145/2684746.2689060>).

Read through Zhang’2015 once for background on the CNN algorithm and implementation. Understand it well enough to know what you would build by hand in RTL. Your objective is to build the same in HLS with less effort.

To get started, even before HLS, unpack the zip file in a regular Linux environment. Build the Lab 2 source code as a regular C++ program. (“**g++ -Wall -g \*.cpp**”, or its equivalent, should be all that is needed.) In a debugger, step through an execution to see how the code is organized and traversed.

* **krnl\_cnn\_layerX()** in **krnl\_cnn.cpp** is a fleshed-out implementation of Figure 9 in Zhang’2015 with parameterized inputs to handle a layer of any size. **krnl\_cnn\_layerX()** is a tiled implementation that operates on a tile of data at a time to improve arithmetic intensity (*what is that?*). **krnl\_cnn\_layerX()** contains the outer loops and the code to prepare the local data tiles to be operated on by the kernel function **cnn\_blocked\_kernel()** (the function “foo()” in Figure 9). For contrast, refer to **ZhangIsfpga15\_1()** in **cnn\_helper.cpp** for a textbook implementation of a CNN layer.
* \*\*\*The subject of high-level synthesis in this lab is the kernel function **cnn\_blocked\_kernel()** declared in **krnl\_cnn\_tile.cpp**.\*\*\* The kernel function **cnn\_blocked\_kernel()** has three array arguments (1) BufI, (2) BufO, and (3) BufW that are the local buffers holding the active tiles of (1) input feature maps, (2) output feature maps and (3) weights, respectively. With respect to these local buffers, **cnn\_blocked\_kernel()**, as provided, follows the same textbook algorithm as in **ZhangIsfpga15\_1()**. The dimensions of these buffered tiles are controlled by **#define** parameters in **kernel643.h**: (1) **TM**, (2) **TN**, (3) **TR**, and (4) **TC**. They control the tile sizes in terms of the numbers of (1) output feature maps, (2) input feature maps, (3) output rows, and (4) output columns, respectively. (The number of input rows and columns are constrained by other parameters and are calculated accordingly.) Vitis HLS will map these array arguments, by default, to BRAM interfaces of appropriate sizes.

Read the Zhang paper and study the code to understand the computation taking place (sequence and timing). To produce a high-quality outcome from HLS, the very first step for you is to conceive in your mind a high-quality RTL design. Next, you can introduce pragmas to instruct the compiler to produce the desired parallelism in memory structure and execution. Rewrite the code as necessary (something as simple as reordering the loop nests can help as you saw in Lab 0). Use the compiler feedback to refine the design. *This exercise should not feel like trial and error. Getting good performance has to be an intentional act.*

* Zhang’s paper is based on adding pragmas to the algorithms that a software programmer would find meaningful in performance tuning (starting from a naive loop nest, introducing tiling and loop reordering). In **krnl\_cnn\_tile.cpp**, an alternate implementation **cnn\_blocked\_kernel\_windowed()** is provided for those who wish to explore further. This implementation only works with stride 1 convolution layers (such as in this lab). This implementation is not offered as the “better” or “easier” solution. It illustrates an example of rewriting the algorithm to express the desired hardware structure and timing. (There would be no motivation for a software programmer to ever write this variation of the code.) To work with this example, you need to first understand what the algorithm implies in structure and timing and then figure out how to convince the compiler to follow suit.

| The coding of **cnn\_blocked\_kernel\_windowed()** imparts additional design information to the compiler beyond what to compute functionally. Most noticeably, **cnn\_blocked\_kernel\_windowed()** maintains a **K\_WTS**-by-**K\_WTS** buffer to increase the reuse of data read from the **BufI** block ram. If you fully unroll the **Krow** and **Kcol** inner loops of **cnn\_blocked\_kernel()** hoping to perform **K\_WTS2** multiply-accumulate per cycle, you will also need to read **K\_WTS2** values from **BufI** each cycle. This concurrent reading is tricky to realize because **BufI** is accessed in a sliding window pattern; the **K\_WTS**-by-**K\_WTS** square of values needed is shifted across the 2D **BufI** according to **col\_b**. However, if you recognize that only a column of the values is new after each shift, you can create a special buffer (the array called **window** in **cnn\_blocked\_kernel\_windowed()**) to buffer the reused values so only **K\_WTS** reads of **BufI** in the column-dimension is needed in each cycle. To work with this version successfully, it is even more important you can see in your mind the corresponding RTL structure and timing.  The **fetchNewColumn()** function makes the code more readable, but it could interfere with the compiler’s analysis. Once you understand the code, you can cut and paste the function body directly to the call site to flatten out the code. |
| --- |

Lastly, the function **main()** in **main.cpp** provides a testbench to invoke **krnl\_cnn\_layerX()** two times (the output of the first layer is consumed by the second layer) and to check its results against **ZhangIsfpga15\_1()**. (Note: in Lab 2, **krnl\_cnn\_layerX()** is a part of the “**testbench**”; only **cnn\_blocked\_kernel()** is being synthesized as hardware.)

[**Creating a Vitis HLS Project**](https://xilinx.github.io/Vitis-Tutorials/2021-1/build/html/docs/Getting_Started/Vitis_HLS/new_project.html)**:** (click the link to see detailed explanation)

1. Start Vitis HLS by issuing the command “**vitis\_hls**”. At the prompt, click “**Create Project**”.
2. Enter a project name and choose the location under “**/scratch/643\_vitis\_<AndrewID>/lab2/**”
3. In the “**Add/Remove Design Files**” window, add “**krnl\_cnn\_tile.cpp**” and all “**.h**” files. Click “**Browse…**” and select **cnn\_blocked\_kernel()** as the Top function for synthesis.
4. In the “**Add/Remove Testbench Files**” window, add “**main.cpp**”, “**krnl\_cnn.cpp**”, “**cnn\_helpers.cpp**” and “**utils.cpp**” as testbench files.
5. In the “**Solution Configuration**” window, click “**...**” under “**Part Selection**” and the “**Device Selection Dialog**” window prompt. Select “**Boards**” and choose “**Ultra96-V2 Single Board Computer**” and click “**OK**”. In the “**Flow Target**”, choose “**Vivado IP Flow Target**”. (\*\*\*Be careful to select **IP-Flow** and not Kernel-Flow in Lab 2.\*\*\*), and click “**Finish**”.

Read [Section 2](https://xilinx.github.io/Vitis-Tutorials/2021-1/build/html/docs/Getting_Started/Vitis_HLS/synth_and_analysis.html#) carefully to learn how to run simulation, synthesis and analyze kernel results.

Your ultimate goal in this lab is to create the highest performing **cnn\_blocked\_kernel()** module using Vitis HLS. You have full flexibility to modify **krnl\_cnn.h** and **krnl\_cnn\_tile.cpp**. You may not make any changes elsewhere in the code. (To use **cnn\_blocked\_kernel\_windowed()**, you have to rename it **cnn\_blocked\_kernel()** since you are NOT allowed to edit **krnl\_cnn\_layerX()** in **krnl\_cnn.cpp**.) In addition, you must observe the following.

1. Your module must compute the correct result relative to **ZhangIsfpga15\_1()** according to the **nearlyEqual()** in both C-sim and Co-sim. *(If you discover your design is correct in C-sim but not Co-sim, reach out to a TA. This is rare. When it happens, the synthesis result is invalid because you have either exercised a C feature disallowed by Vitis HLS, used a pragma incorrectly, or found a bug in Vitis HLS.)*
2. When debugging your design, you should temporarily reduce the problem size in **instance643.h** (by flipping “**#if 1**” to “**#if 0**” in **Line #53**). For final validation, C-sim needs to pass the test at the full size. Co-sim can be tested at the reduced problem size to save time.
3. The operation sequence and timing of your module must be “**data independent**”. (That is, it should dutifully run through all of the same steps regardless of the data values in the input, weight and output buffers)
4. You cannot change the declaration of **cnn\_blocked\_kernel** (arguments, return value, and their types). The input, weight, and output data arrays at the interface must be mapped to use BRAMs. This is the default.
5. You may not embed RTLs or instantiate library IPs. The entire module must be synthesized from C using Vitis HLS. (If you are curious, you can examine the HLS-generated RTL by following the instructions in [Exporting the RTL Design](https://docs.xilinx.com/r/2021.1-English/ug1399-vitis-hls/Exporting-the-RTL-Design))
6. You must target the Ultra96-V2. Your module must fit in the targeted FPGA.
7. On your final submission, your synthesis must not have a timing violation at your selected “**Target**” clock frequency with default **Uncertainty**.
8. **\*\*Embed your pragmas in the source files; do not use a directive file\*\*.**

**Performance:** For performance, we are interested in arithmetic operations per second in a batch processing scenario. Thus, you can assume the kernel is invoked as often as available. You do not need to be concerned with how the input and output are delivered to or extracted from the module’s BufW, BufI, and BufO BRAMs.

* Use **2×Tm×Tn×Tr×Tc×K×K** for operation count when computing ops-per-sec. To determine execution time in seconds, multiply **cnn\_blocked\_kernel**’s cycle counts----smaller of INTERVAL or LATENCY----by its clock period. (For clock period, use the “**Target**” value, not “**Estimated**”.).
* You need to manually account for the BRAMs needed to map BufW, BufI, and BufO. Vitis HLS does not include the interface BRAM cost in its resource report. To determine the number of interfacing BRAMs (BRAM\_18K) used:
* For each I/O BRAM buffer, look at the Interface report to see how many BRAM ports are generated. (If you partitioned an input or output array, Vivado HLS will produce multiple BRAM ports for that one array argument.)
* For each BRAM port, look up its data width and height (i.e., 2address\_width).
  + calculate capacity (in **bits**) as data width (in **bits**) \* height
  + calculate **A=⌈capacity / 214 ⌉**, *i.e., min # BRAM needed for capacity*
  + calculate **B=⌈data width / 32⌉**, *i.e., min # BRAM needed for data width (assume that 1 BRAM\_18K only gives 1 data port and the port width is 32-bit)*
  + the estimated number of BRAM\_18K is **the greater of A or B**
* Sum up the BRAM\_18K counts over all ports. There are 432 B18Ks total.

If one instance of your module consumes less than **45%** of each of the resource types, find the largest integer **Q** such that **Q** copies of your module would still consume less than **(100-(Q-2)×5-10)%** of each of the resource types. Your total ops-per-sec is **Q** times the ops-per-sec of one module. (The hold back when **Q**>1 is to account for the external overheads of having multiple modules operating in parallel.)

**Submit:** Create a submission directory **<team\_name>\_lab2** 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>\_lab2.zip** (that is the zip of your submission directory) through Canvas.

Include in your submission directory **krnl\_cnn.h** and **krnl\_cnn\_tile.cpp**. **Embed your pragmas in the source files; do not use a directive file.** Comment extensively so the intention of your code is clear. You should not submit the entire project.

Include the HLS synthesis report (**Explorer→solutionX→syn→report→cnn\_blocked\_kernel\_csynth.rpt**).

Include a **report.pdf** with brief discussions of the following. Number the sections accordingly.

1. Explain the major changes and optimizations employed in your final design.
2. Explain any changes and optimizations attempted but ineffective (say why?)
3. Provide a sketch of the hardware datapath resulting from your code (focus on the arrangement of the compute elements (i.e., the multiply-and-accumulate units), the memory structure (i.e., mapping of array to SRAM banks), and the interconnectivity (show multiplexing when multiple signals converge)).
4. Explain how you determined the optimum tile dimensions (**TM**, **TN**, **TR**, **TC**).
5. Show and explain your calculation of ops-per-sec for 1 module.
6. Show and explain your calculation of interface BRAM usage for 1 module. (Do this by constructing a table with the following columns: “**port instance**” “**data width**” “**address width**” “**height**” “**capacity**” “**A**” “**B**” “**# BRAM\_18K**”. List one port instance per row and sum total in the last row.)
7. Report resource utilization for 1 module, broken down by resource type, in absolute count and in percentage of available (as in the synthesis report); don’t forget to add the interface BRAMs.
8. Show and explain your calculation of **Q** and the resulting total performance and resource utilization.
9. Show and explain your calculation of Arithmetic Intensity (in this case, AI should be defined as the number of ops by 1 iteration of the kernel divided by the total size of the 3 tile buffers in bytes).
10. Any interesting insights from working with HLS
11. At this moment, would you consider using HLS 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.

**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 performance achieved. You will receive the full 40% if you achieve greater than one-half of the ideal peak performance. (For this purpose we will assume all DSPs and only DSPs are used for multiply-and-accumulate and the DSPs run at 500MHz.) You will receive 30% if you achieve greater than one-quarter of the ideal peak performance. You will receive 20% if you achieve greater than one-eighth of the ideal peak performance. You will receive 10% if you achieve greater than 3x the performance of the starter project (unmodified synthesized at 3ns clock). Otherwise, you will receive 0%.