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

**Handout #4/Lab 1: Up and Running (150 points)**

Issued: 9/11/2023

Due: 9/25/2023 noon

This lab must be completed in a team of 3. This lab will bring you further up to speed on the Ultra96 (V2) board and the Vitis IDE. Please post questions and answers on 18643’s Piazza page to help each other out with tools and board related issues.

There are a lot of steps to follow. They will work if you follow them exactly; be as careful as you can. Read the handout completely (at least section by section) before starting your work. Note the warnings about common mistakes to watch out for. It is a good idea to Zoom record your desktop during a work session; it can help the TA diagnose where you went off script.

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

Begin the lab by borrowing an Ultra96 board from the ECE Receiving (HH 1301).

Before doing anything with your Ultra96 board, review the [Ultra96-V2 Hardware User’s Guide](https://www.avnet.com/wps/wcm/connect/onesite/b85b9556-0b2a-42b3-ad6a-8dcf3eac1ff9/Ultra96-V2-HW-User-Guide-v1_3.pdf?MOD=AJPERES&CACHEID=ROOTWORKSPACE.Z18_NA5A1I41L0ICD0ABNDMDDG0000-b85b9556-0b2a-42b3-ad6a-8dcf3eac1ff9-nDNP5R3) and the [Ultra96-V2 Getting Started Guide v2.0](https://www.avnet.com/wps/wcm/connect/onesite/f21462c6-4997-41a2-a95e-80122b73aea9/Ultra96-V2-GSG-v2_0.pdf?MOD=AJPERES&CACHEID=ROOTWORKSPACE.Z18_NA5A1I41L0ICD0ABNDMDDG0000-f21462c6-4997-41a2-a95e-80122b73aea9-o8Z3.B6) on Avnet’s Ultra96 page ([Ultra96-V2 | Avnet Boards](https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/ultra96-v2/)).

Follow the steps in the Ultra96-V2 Getting Started Guide to check out that your kit is complete and functional. Later on, you will need to repeat similar steps to run your compiled Vitis project.

**Part 2: Building and Running a Vitis project**

Repeat Lab 0’s tutorial instructions, but this time targeting the real Ultra96v2 hardware. Note the following differences:

1. When prompted to “select a directory as workspace”, enter the directory:

“**/scratch/643\_vitis\_<your AndrewID>/lab1**”

1. Under the “**Select a platform from repository**” tab of the “**Platform**” window, select the Ultra96 **cmu\_u96v2\_dfx\_full** platform instead of the zcu102. (Make sure you select the “**dfx**” version.)
2. In the “**Domain**” window, you will find that the Ultra96 platform has set the necessary paths for you already. Do not replace them. Double check they are

* Sysroot: /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
* Root FS:

/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

* Kernel Image:

/afs/ece.cmu.edu/class/ece643/software/xilinxVitis/platforms/2021.1/cmu\_u96v2\_dfx\_full/sw/cmu\_u96v2\_dfx\_full/linux\_domain/image/Image

1. When the project window starts, proceed to build the “**Hardware**” target. (This can take 15 minutes or more)

When the build is finished, proceed with the following instructions to execute the project on the Ultra96.

***Running for the First Time on the Ultra96:***

1. Start from Step 6 in the *Ultra96 Getting Started Guide*. Instead of writing the example image, use the SD card image compiled by Vitis for your vector-add project. Find the SD card image from the Hardware/package folder at directory: “**/scratch/643\_vitis\_<your AndrewID>/lab1/test\_drive\_system/Hardware/package/sd\_card.img**”
2. Download the SD card image from the ECE server to your local machine that has the SD card adaptor. (For example, from a Linux local machine, download the sd-card image by issuing “**scp -C <user>@ecexxx.ece.local.cmu.edu:<path to sd\_card.img> ./**”, where **ecexxx** is the ECE server with the Vitis **/scratch** workspace. The image is around 4GB; adding the compression option (**-C**) will significantly speed up the download.)
3. Follow the instructions on [Ultra96v2\_2020\_1\_Factory\_Image\_Write](https://www.avnet.com/wps/wcm/connect/onesite/7339b7a6-0c1c-4bf4-bf60-8e53898ab358/Ultra96v2_Factory_Image_Write_190611.zip?MOD=AJPERES&CVID=nxs6V57&CVID=nxs6V57&CVID=nxs6V57) to write the Vitis project SD card image into the micro-SD card. (On Step 5 of the webpage instructions, choose the **sd\_card.img** you just downloaded instead of the image archive mentioned in the document) When finished writing, insert the micro-SD card into the Ultra96 board and turn it on.
4. Follow Step 8 and Step 11 of the *Ultra96 Getting Started Guide* to connect to the Ultra96 board using **ssh** over WiFi. At the PetaLinux command prompt, change directory to “**/mnt/sd-mmcblk0p1**”. Use “**ls**” to see there are the same files as in Lab 0’s emulated system. Note, “**binary\_container\_1.xclbin**” is the FPGA configuration bitstream (corresponding to the hardware kernel in **krnl\_vadd.cpp**) and the “**test\_drive**” file is the binary executable for the ARM core (corresponding to the program in **vadd.cpp**).
5. Issue “**./test\_drive binary\_container\_1.xclbin**” at the command prompt to start an execution. (You are passing the bitstream to the “**test\_drive**” program as an argument. The “**test\_drive**” program will load the bitstream onto the fabric before invoking the hardware kernel.) You should see the same output as what you saw in Lab 0 in the emulation. If you see “**TEST PASSED**”, congratulations!

***Rerunning after Recompiling:*** Copying the entire 4GB SD card image (that holds the Petalinux image and file system) every time you make a change to the Vitis project is not necessary. After loading the complete SD card image once (hopefully for the entire course - unless you need a clean slate), you only need to update the Vitis project output files (the software executable “**test\_drive**” and the bitstream in “**binary\_container\_1.xclbin**”) to run a different design. *This flow is fairly robust. But when in doubt, revert to the full flow above if something doesn’t go right.*

1. Instead of copying the “**sd\_card.img**” file, copy the files “**test\_drive**” and “**binary\_container\_1.xclbin**” from the ECE server to the machine you are using to access the Ultra96 by WiFi. Later when you have multiple “xclbin” files, you need to copy all the “**xclbin**” files over.
2. Copy the files onto the Ultra96 SD card over the WiFi connection. If you use a Linux local machine, you can run: “**scp <file> root@192.168.2.1:/mnt/sd-mmcblk0p1/**”. (During this, your Ultra96 needs to be booted up in Linux.) This overwrites the previous versions of “**test\_drive**” and “**binary\_container\_1.xclbin**” with the new ones. You can also rename them to have both copies side by side.
3. Run your updated project by issuing “**./test\_drive binary\_container\_1.xclbin**” from **/mnt/sd-mmcblk0p1/**.

**\*\*\*Important Note about Using /scratch\*\*\***

Vitis does not work well with AFS workspaces. As a work around, we will use workspaces in the **/scratch** directory which is on the local disk of the workstation you are logged into. Understand the following:

* **/scratch is shared.** Keep your workspace under **/scratch/643\_vitis\_<your andrewID>** to avoid name conflicts. Check and set permission for privacy. You are responsible for protecting your own work.
* **/scratch is not backed up.** Save away your files at the end of each work session, before you log off from the workstation. You should do this by “exporting” a project archive which creates a very compact zip file of just the essential sources and configurations. (right-click on the top-level project “**mmm\_system**” in the “**Explore**” pane to find “**Export as an archive**”). Move/save this archive to your AFS space. This archive can be easily loaded when you start up again next time by “importing”. It is not recommended that the archive includes generated build files (takes up GBytes). If you do archive build files, be aware that AFS write bandwidth is very low.
* **/scratch is local to a specific machine.** Unlike AFS, you need to transfer your work manually (export/import) if you switch machines.
* **Untouched files in /scratch are purged periodically.** You are *probably* safe for the timeframe of a 2-week lab. If you reuse the same machine each time, you can restart from where you left off. Just to be safe, you should still export an archive (of just sources) at least daily.

This is not as painful as it might sound. It does require you to keep better track of where things are. The upside of using **/scratch** is that Vitis runs much faster than if using AFS. I would probably use **/scratch** for builds even if AFS worked.

**Part 3: Developing Your Own First Application**

Starting from the vector addition example, you are asked to develop a matrix-multiplication acceleration example. For this part, download the “[**lab1\_mmm\_dfx\_2021\_1.ide.zip**](https://drive.google.com/file/d/1SFQaB9ZRC4R-xt15npVuFbWmvnHOC47p/view?usp=drive_link)”. Launch Vitis using **/scratch/643\_vitis\_<your\_andrewID>/lab1** as workspace. Use the “**import**” option on the start page or the pull-down menu to load this archive.

The pre-populated project **mmm\_vadd\_dfx\_system** contains the vadd code example plus the stubs for you to develop your own matrix-matrix multiplication code. You should be able to complete the coding straightforwardly by deriving correspondingly from what you see for the vadd code example. The vadd code has been repackaged into helper functions to make the code more clear (**mmm\_vadd\_dfx\_system→mmm\_vadd\_dfx→src→vadd\_helper.h/.c**).

When **lab1\_mmm\_dfx\_2021\_1.ide.zip** is imported, the project is set up for **SW-Emulation** on the **zcu102**. (Build and run the project to check the vadd portion of the code runs correctly. The mmm portion of the code will, of course, fail until you fill in the missing parts.) The Ultra96 platform does not support emulation. Therefore, you should fully develop and debug your code using zcu102 **SW-emulation** before compiling for the Ultra96 hardware for final testing. *(We have found HW-emulation unreliable and very slow to compile and run. Avoid it. Ask the TAs for help if you find you need to perform HW-emulation for some reason.)*

**ZCU102 SW-Emulation:** Use reduced problem sizes for debugging in SW-emulation to save time. Note that every time you rebuild the project, you need to update the emulated system. The easiest way is to just restart the emulator by clicking "**Xilinx->Start/Stop Emulator**", and then restart from “**Run As**”. *If you don’t restart the emulator, “*Run As*” just starts a run on the already running emulator using the old image it was booted with.*

**Ultra96 Hardware:** When you are ready to compile for the Ultra96 “**Hardware**”, open “**mmm\_vadd\_dfx\_system→ mmm\_vadd\_dfx\_system.sprj**” in the “**Explorer**” pane. Change the platform target from the original zcu102 default to **cmu\_u96v2\_dfx\_full**. When you do, the paths to rootfs, etc. project should, but may not, change automatically. Use the paths listed earlier in the document to double check the sysroots, rootfs and image fields. (If you later switch back to zcu102 for SW-emulation, you will have to reset those paths manually as in Lab 0.)

Note: When you are ready to run, note that there are separate bitstream files for vadd and mmm. Remember to transfer both “**binary\_container\_vadd.xclbin**” and “**binary\_container\_mmm.xclbin**” onto the Ultra96 and execute them as “**./mmm\_vadd\_dfx**”. (We have hardcoded the xclbin filename arguments into main().)

**To complete the mmm\_vadd\_dfx\_system project:**

* Read the vadd host and kernel code example. (See **mmm\_vadd\_dfx\_system→mmm\_vadd\_dfx →src→{main.cpp, vadd\_helper.h/.c**} and **mmm\_vadd\_dfx\_system→mmm\_kernels→src→ krnl\_ mmm.cpp**)
* Read the corresponding mmm host and kernel code stub to see what is missing. (Look for the “TODO” messages. )
* Make the necessary changes to both the mmm host and kernel code to multiply two floating-point matrices. The mmm helper functions include initialization of the input matrices and results checking.
* The acceleration kernel is declared as below. *Do not change the kernel function declaration.*

void krnl\_mmm( const float \*in1,

const float \*in2,

float \*out\_r,

int size );

* The buffers ***in1***, ***in2*** and ***out*** should be treated as 2D arrays of ***size***-by-***size*** in a row-major layout. The value of ***size***is always positive.
* You need to add timing measurements to enable you to compute the “arithmetic-ops-per-second” performance of the matrix-multiplication kernel (when running on Ultra96 hardware).
  + For ***size*=4096**, measure the end-to-end time, including loading of input data and unloading of results (i.e., **q.enqueueMigrateMemObjects**, but not the time for allocating/initializing/checking the buffers). FYI, a standard matrix multiplication algorithm requires **2\**size3*** arithmetic operations. (In the starter code, the matrix size is a parameter defined in **main.cpp**.)
  + In addition, still for ***size*=4096**, determine and report the “arithmetic-ops-per-second” performance of just the kernel itself (discounting the overhead of data migration).
* Be sure to comment extensively so the intention of your code is clear.
* You will submit Part 3 code together with Part 4.
* Optional: The mmm kernel is more elaborate than the vadd kernel. Measure the time for reconfiguring the FPGA. Does reconfiguration time depend on the kernel complexity? Does the bitstream size depend on the kernel complexity?

Note: Your initial hardware run should start at a small size, say 128 or 256. Calculate ops-per-second at the small size then estimate how long a run at size=4096 would take. If the estimated time to run the kernel is much more than 5 minutes and you don’t like to wait, you have the option to scale down the benchmarking problem size from the stipulated 4096 to a smaller 2-power size such that the run time of the kernel is at least 5 minutes. In the reporting, clearly indicate the actual size you used for performance measurements.

**Performance is not an objective for this part of the lab.** If you do decide to optimize for performance, please keep in mind the zcu102 SW-emulation target is a larger device than the Ultra96v2. If your optimized design fully utilizes the zcu102 logic resources, it will not fit on the Ultra96v2.



**Part 4: Performance Benchmarking**

You will develop Part 4 by extending the Vitis project from Part 3 with additional partial-reconfiguration (PR) kernels. These hardware kernels should be closely modeled after mmm. You will modify the host program to invoke these new kernels.

To add an additional PR hardware kernel (e.g., for Experiment 1 Part A),

* Open “**mmm\_vadd\_dfx\_system→mmm\_vadd\_dfx\_system.sprj**” in the “**Explorer**” pane. In the “**Application Projects**” section of the “**System Project Settings**” you should be able to see all the projects including the two kernel projects: “**mmm\_kernels**” and “**vadd\_kernels**”.
* In the “**Application Projects**” section, click the dropdown between the “**+**” and “✖” signs, and choose “**New hw kernel project…**”. Enter a name for your new kernel project (e.g., **exp1a\_kernels**) and click Finish. You should now be able to see the new project in the “**Application Projects**” section and also the “**Explorer**” window.
* To create a kernel function and register it with Vitis in the newly created project:
  + Copy “**mmm\_vadd\_dfx\_system→mmm\_kernels→src→krnl\_mmm.cpp**” to “**mmm\_vadd\_dfx\_system→exp1a\_kernels→src→krnl\_exp1a.cpp**”.
  + Edit the function name **krnl\_mmm** in **krnl\_exp1a.cpp** to the desired new kernel name (e.g. **krnl\_exp1a**).
  + Open “**mmm\_vadd\_dfx\_system→exp1a\_kernels→exp1a\_kernels.prj**”. In the “**Hardware Functions” section** (you should not see any entries yet), click the (filled blue circle with a lightning bolt and a + symbol - “**Add Hardware Function…**”) icon. Choose the function you just created from the list of “**Matching items**” and press “**Ok**”.
  + You should now see your kernel function in the “**Hardware Functions**” list.
* Now to create a new binary container for the new kernel:
  + Open “**mmm\_vadd\_dfx\_system→mmm\_vadd\_dfx\_system\_hw\_link→mmm\_vadd\_dfx\_system\_hw\_link.prj**”. You should be able to see the two binary containers for vadd and mmm in the “**Hardware Functions**” section. The new kernel is, by default, added to one of them; right click on the new kernel (**krnl\_exp1a**) and click “**Remove**”.
  + In the “**Assistant**” window (usually right below the “**Explorer**” window), navigate to “**mmm\_vadd\_dfx\_system→mmm\_vadd\_dfx\_system\_hw\_link**”. Right click any of the build configurations (Emulation-SW, Emulation-HW, or Hardware), and click “**Add Binary Container…**” - edit the name from the default to something more meaningful (e.g. **binary\_container\_exp1a**). You should now be able to see a new binary container in the “**Hardware Functions**” section.
  + Choose the newly created binary container and click the “**Add Hardware Function…**” icon ( ), choose the new hardware kernel and click “**Ok**”. The new kernel should appear under the new binary container, and you’re all set to create a new bitstream “**xclbin**” file. 
* Finally edit your host cpp code to read in the extra command line arguments (i.e. the “**xclbin**” file names) and use the template at the end of the main.cpp file to integrate with your new hardware kernel. (You also need to expand xclbinFilename[]’s declaration at line 46 of main.cpp to add the new container names.) You can create copies of the helper headers and cpp files as needed. (mmm and experiment kernels should be able to share many helper functions.)
* Follow the same steps as before to build your project. Note: You’ll now have to move all the “**xclbin**” files along with the host executable to the Ultra96.

**Experiment #1.** Develop two new hardware kernels krnl\_exp1a and krnl\_exp1b (based on the mmm kernel) to read through the entire 4096x4096 *in1* array once. Determine (by measurements and calculations) the best-achievable memory read **bandwidth** (bytes/sec) for performing the reads (1) row-by-row and (2) column-by-column.

**Experiment #2.** Develop additional hardware kernels (krnl\_exp2a and krnl\_exp2b) to repeat the above for writing into the *out* array.

**Experiment #3.** Develop another hardware kernel (krnl\_exp3) to determine the best achievable **latency** of reading 1 integer value from DRAM in the FPGA kernel, that is from the time when a load is issued by the kernel to when the data is available for use by the kernel.

In the above, the host program code should include the calculation and reporting of the findings. In this part of the lab, performance does matter. You are asked to design the correct experiment to determine the best achievable values. Note that none of the above metrics can be measured directly. In Experiment #1 and #2, you are measuring time to use to calculate the bandwidth. Think carefully about what to do in Experiment #3. It is useful to have an idea of what to expect before you begin. Look up the spec for LPDDR4.

**To Submit**: Create a submission directory **<team\_name>\_lab1** 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>\_lab1.zip** (that is the zip of your submission directory) through Canvas.

Include in the submission directory a single Vitis project archive named **lab1.ide.zip**. Building then running the project should run through all of the kernels (vadd, Part 3 mmm, Part 4 experiments). Part 3 and Part 4 should be for 4096-by-4096 matrices. The host program should include the performance calculations and \*\*print\*\* out the requested results (1~5 below) for inspection.

Include a PDF file <team\_name>.pdf which reports:

1. Part 3: end-to-end performance (unit: op/sec)
2. Part 3: kernel-only performance (unit: op/sec)
3. Part 4: Experiment 1 (1) row-major read BW, and (2) column-major read BW (unit: byte/sec)
4. Part 4: Experiment 2 (1) row-major write BW, and (2) column-major write BW (unit: byte/sec)
5. Part 4: Experiment 3 DRAM read latency (unit: sec)
6. The frequency of the fabric and the ARM core in **your** Part 3 project.
7. What is the associativity, block-size, and capacity of the ARM core’s L1 and L2 cache.
8. Provide a summary of the FPGA fabric resource utilization of the Part 3 kernel.
9. Explain the different memory bandwidths in Part 4 Experiment 1 and 2.
10. Optional: Reconfiguration time. Is it dependent on the kernel?

For 1~5, you need to explain your methodology (experimental design, timing instrumentation, calculations) for determining the answer values. (Be especially careful with Part 4 Experiment 3. You can determine 6 and 7 by experimentation or by looking it up. For 8, tell us how you found this information and how you figured out how to find this information.

We will also ask you to summarize quantitative results in a Canvas questionnaire.