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

**Handout #2/Lab 0: Just Warming Up (100 Points)**

Issued: 8/28/2023

Due: 9/11/2023 noon

* This lab is to be completed individually. This lab will ease you into the appreciation for “performance correctness” in addition to “functional correctness”. You will also work through a tutorial on using the Xilinx Vitis IDE.
* During these first two weeks of the semester, you are asked to form a group of 3 to work together on subsequent labs and projects. Please refer to the syllabus (on Canvas) for guidelines regarding lab groups.
* To complete this lab, you are to submit through Canvas a zip file **<your AndrewID>\_lab0.zip** (which should expand to a submission directory named **<your AndrewID>\_lab0**). The directory should contain a pdf file **<your AndrewID>\_lab0.pdf**; additional items to include are prescribed below.
* When a question asks for a numerical answer, please answer in scientific notation; only 1 significant digit is sufficient.
* Points will be deducted for failing to follow submission instructions (e.g., incorrect directory and file names, incorrect file compression, missing submission elements, extraneous submission elements).

**Part 1: Matrix-matrix multiplication**

In this part, you will explore the implementation and performance of matrix-matrix multiplication in single-thread software on a CPU. A ready–to-build textbook implementation to multiply two 4096-by-4096 matrices of doubles is provided for you as a C function ([lab0-mmm.zip](https://drive.google.com/file/d/1pTho9TJz8jnV_SM5byWtpJOxxzVhOcm7/view?usp=drive_link)).

**Step 1**: Before building and running the program, review the basics of matrix multiplication by studying **mmm()** in **mmm.c**. Determine the number of double-precision floating-point operations (multiply and add) performed. Determine the number of bytes read from DRAM and the number of bytes written to DRAM. Provide your answer along with a brief explanation in the submission PDF file under the heading “**Part 1 Step 1”**.

**Step 2:** Pick a [“numbers” cluster machine](https://cmu-enterprise.atlassian.net/wiki/spaces/ITS/pages/2332131370/ECE+Community+Compute+Clusters#ECE-Community-Cluster-Specifications-(ECE-NUMBER-Cluster)) and find its processor and memory specifications online. Based on the specifications, estimate how much time it should take **mmm()** from **Step 1** to multiply two 4096-by-4096 matrices of doubles. Calculate the estimated GFLOPS (billions of floating point operations per second). Provide your answer (estimated execution time and estimated GFLOPS) along with a brief explanation in the submission PDF file under the heading “**Part 1 Step 2”**. (The explanations should enable a grader to understand and reproduce your estimation.)

**Step 3.1:** Build and run the program provided (using the “make” command). The program will report the execution time of **mmm()**. Run the program a few times to see variances in the measurements. *(A good rule of thumb is to report the fastest of the measurements seen. Repeated timing is most important for timing short events, lasting seconds or less, in case of spurious systems interference. For events that last minutes or more, repeated timing becomes less important. When practical, it is a good practice to time on a freshly booted, unloaded system.)*

**Step 3.2:** After unzipping, the program is set to multiply 1024-by-1024 matrices. Edit **mmm.c** to change **SIZE** to **4096**.

1. Rebuild and rerun to measure the actual execution time for 4096-by-4096 matrices.
2. Calculate the achieved GFLOPS.
3. Calculate the fraction of the idealized performance actually achieved.

Provide your answers (measured time, calculated GFLOPS, fraction of ideal) along with a brief explanation in the submission PDF file under the heading “**Part 1 Step 3”**. *(A 4K-by-4K mmm needs to do 64x more operations than 1K-by-1K. You will find that 4K-by-4K mmm also does the operations more slowly. Go get a coffee or something.)*

**Steps 4/5/6:** Edit **main()** in **mmm.c** to call **mmm\_outer()** instead of **mmm()**. Convince yourself that **mmm\_outer()** and **mmm()** should compute the same results (within the allowed floating-point imprecision.). Repeat **Steps 1, 2, 3** as **Steps 4, 5, 6.**

**Step 7**: Explain the performance difference seen in **Step 3** and **Step 6**.

**Part 2: Xilinx Vitis Setup and Tutorial (Video Tutorial on Canvas)**

18-643 labs will use the Xilinx Vitis IDE workflow and the Ultra96v2 single-board computer (which you will receive on week 3). There is nothing to submit in this part. Work through all 5 parts of this [Vitis 2021.1 Getting Started Tutorial](https://xilinx.github.io/Vitis-Tutorials/2021-1/build/html/docs/Getting_Started/Vitis/Getting_Started_Vitis.html). Refer to the following alternate instructions to match the software setup in 18-643. Please post questions and answers on 18643’s Piazza page to help each other get started.

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.

* **Tutorial “Part 1”**: Read carefully.
* **Tutorial “Part 2”**: You can skip this step. You do not need to install Vitis on your own. You will use Vitis 2021.1 installed for the ECE Linux cluster. For this, you do need to figure out how to connect remotely to ECE Linux servers and use X-windows [Look here - [ECE ITS Public Resources: FastX from StarNet](https://cmu-enterprise.atlassian.net/wiki/spaces/ITS/pages/2352185345/FastX+from+StarNet) and [ECE Community Compute Clusters](https://cmu-enterprise.atlassian.net/wiki/spaces/ITS/pages/2332131370/ECE+Community+Compute+Clusters)]. You can use the physical workstations in the lab if you cannot get remote X to work.
* **Tutorial “Part 3”**: Read carefully.
* **Tutorial “Part 4”**: Read through but do not follow their instructions for using Vitis by command line. We will do the same thing using Vitis’ GUI interface. For Lab 0, we will target the ZCU102 embedded platform in emulation.
  1. Log into a ECE linux workstation. Test your X-window setup by opening an “**xterm**”.
  2. At the Linux command prompt in the xterm, configure your environment by entering “**source /afs/ece.cmu.edu/class/ece643/software/scripts/setup\_vitis21.lab.sh**”. (Notice that the setup script automatically creates for you a directory “**/scratch/643\_vitis\_<your AndrewID>**”. **/scratch** is on the local disk of the machine you are logged into. The script configures the directory (and its subdirectories) for access by you only. Be careful when creating directories in **/tmp** or **/scratch** manually.)
  3. From the same xterm, start Vitis by entering “**vitis**”
  4. Wait for the Vitis welcome screen. When prompted to “**select a directory as workspace**”, enter the directory “**/scratch/643\_vitis\_<your AndrewID>/lab0**”. (You have to run Vitis from **/scratch** because Vitis doesn’t work properly with AFS workspaces.) Click “**Launch**”.
  5. At the next screen, select “**Create Application Project**”. Click “**Next**” to skip the next overview screen.
  6. Under the “**Select a platform from repository**” tab of the “**Platform**” window, select “**xilinx\_zcu102\_base\_202110\_1**”. Click “**Next**”.
  7. In the “**Application Project Detail**” window, assign a project name (e.g., “**test\_drive**”) then click “**Next**”.
  8. In the “**Domain**” window, enter the following then click “**Next**”.
     + “**/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**”
  9. In the “**Template**” window, select “**Vector Addition**”. Click “**Finish**” and a pre-populated project window for Vector Addition will appear.
  10. In the “**Explorer**” pane, open “**test\_drive\_system→test\_drive\_kernels→src→krnl\_vadd.cpp**”. The C function “**krnl\_vadd**” will be compiled by high-level synthesis to an accelerator module on the FPGA fabric. Its functionality should be readily understandable.
  11. In the “**Explorer**” pane, open “**test\_drive\_system→test\_drive→src→vadd.cpp**”. The C function **main()** is the OpenCL host program that will run on the embedded ARM core to allocate and initialize the input data buffers; invoke the FPGA acceleration kernel; and check the computed results in the output data buffer.
  12. Right-click “**test\_drive\_system**” in the “**Explore**” pane. Select “**Properties**” in the menu. Select “**Run/Debug Settings**” on the left. When presented with the choices, (scroll down to) select “**test\_drive\_system-Default**” then click on “**Edit**”. Click on “**Edit**” corresponding to “**Xilinx Runtime Profiling -- Configuration**”. Select “**OpenCL trace**”. (“**OpenCL summary**” should already be selected. If not, select it too.) Exit out by clicking “**Ok**” to exit out of the “**Xilinx Runtime Profiling**” window. Click “**Apply**” then “**OK**” to exit out of the “**Edit launch configuration properties**” window. Finally click “**Apply and Close**”. (This step is needed to produce the profile and trace files for Part 5 of the tutorial.)
  13. You won’t receive an Ultra96 board until week 3. We will only test emulation in this tutorial. In the “**Explorer**” pane, open “**test\_drive\_system→test\_drive\_system.sprj**”. Make sure the “**Target**” says “**Software Emulation**”. This is the quickest emulation mode serving only to check out the functional correctness of the C code. Right-click “test\_drive\_system” in the “**Explore**” pane then select “**Build Project**”. Watch the build progress notification on the bottom-right of the window.
  14. After the build finishes, launch emulation by right-clicking “**test\_drive\_system**” in the “**Explore**” pane, then select “**Run As→Launch SW Emulator**”. This will start a full system emulation to boot petalinux then launch the host program. You can monitor the kernel messages during boot-up in the “**Emulation Console**” pane.
  15. If all goes well, after booting completes, you will see “**TEST PASSED**” printed by the host program output in the “**Console**” pane.
  16. Below the “**Emulation Console**” is the interactive command prompt textbox for petalinux on the “**emulated**” ARM core. Type “**cd /**” then “**ls**” in the textbox to try it out. Type “**cd /mnt/sd-mmcblk0p1**” then look around with “**ls**”. (This is the directory holding the executable and its files.)
  17. When you are ready to end the emulation, stop the emulator by selecting from the top-left pull-down menu “**Xilinx→Start/Stop Emulator**”.
  18. Now try on your own to repeat the exercise in this part for HW Emulation. This is cycle-level, RTL-based. Building and simulating will take longer. In practice, most of the time, you could go directly from correct software emulation to working design on an FPGA. However, when it doesn’t work on the FPGA, it is a good idea to test hardware emulation. The Vitis HLS compiler is known to produce incorrect RTL (so a design that passes in emulation may not work in RTL simulation or real hardware). Hardware emulation is also needed to gather the detailed performance tracing in the next section.
* **Tutorial “Part 5”**: Continue here only after HW emulation completes successfully. Start Vitis Analyzer by, in the “**Assistant**” pane (lower left, below the “**Explore**” pane), right-click on “**test\_drive\_system→test\_drive→Emulation-HW→test\_drive\_system-Default\_test\_drive→Run Summary (xclbin)**”. Then click “**Open in Vitis Analyzer**”. Continue with the instructions on the tutorial webpage.
* Before logging out of the workstation, delete the directory **/scratch/643\_vitis\_<your\_AndrewID>**. Copy what you like to keep to AFS, but there is no need to keep anything from this tutorial. Type the “**ps**” command in the unix prompt to check that all of your Vitis processes have terminated. Use the “**kill**” command to stop any dangling Vitis processes.

Congratulations. You are ready for 18-643. You can explore Vitis further by importing the Vitis project archive files of [Rosetta Benchmarks](https://www.csl.cornell.edu/~zhiruz/pdfs/rosetta-fpga2018.pdf) located under the [18643 lab directory](https://drive.google.com/drive/folders/1R8batFp2O7dkMxppGPALsOMi7DZGGDF2?usp=drive_link). (Use the “**import**” option on the start page or the pull-down menu to load these archives.) You can follow the other tutorials in [Vitis Hardware Acceleration](https://xilinx.github.io/Vitis-Tutorials/2021-1/build/html/docs/Hardware_Acceleration/Hardware-Acceleration.html) to learn how to design for the Ultra96 using traditional RTL flows. You can also find overviews of Xilinx’s high-level domain-specific environment for ML.

To learn more about the Ultra96 board that you will receive on week 2, begin by visiting [Ultra96-V2 | Avnet Boards](https://www.avnet.com/wps/portal/us/products/avnet-boards/avnet-board-families/ultra96-v2/). There is a large Ultra96 user community of tinkerers and hackers online. You will be surprised how often googling a specific question phrase will return exactly the answer you need.

**Submit:** There is nothing to submit for this part.

**Part 5: Team**

**Submit:** List the names and AndrewIDs of your group members (including yours) in <your AndrewID>\_lab0.pdf under the section heading Part 5.

**Part 6: (Optional)**

For most of you, the tutorial in Part 4 will just work as described if you followed the instructions exactly. Even though there is nothing to hand-in in Part 4, it is very important you actually work through the steps (and try to understand what is actually happening). The purpose of Part 4 is to ensure you are ready to go onto Lab 1. Lab 1 does not budget time for you to become familiar with Vitis and to iron out tools issues.

If you want a head start, begin tinkering with the **vadd** Vitis project. Try changing the vector size to make sure you do know how I/O buffers are allocated and passed as arguments. Try changing the kernel to take an additional argument *in3* to perform *for all i, out\_r[i]=in1[i]+in2[i]\*in[3]*.