CSE 598/CEN 598/CSE 494 Lab 2

GEMM accelerator using High Level Synthesis

Objectives

1. Learn how to use AMD/Xilinx Vitis HLS and Vivado tools.
2. Perform high-level synthesis (HLS) on a GEMM (matrix multiplication) function.
3. Learn how to generate designs with different performance characteristics.
4. Connect the generated GEMM accelerator to an on-chip processing system.

Background

GEMM refers to General Matrix Multiply. It is the most common operation in machine learning (ML) workloads. Optimizing GEMM computation is essential to running ML workloads efficiently on any platform (CPU, GPU, FPGA, etc.). A GEMM operation basically consists of three nested loops. The inner loop contains a multiply-and-accumulate operation. If you are interested, you can read through these blogs that talks about GEMM:

https://petewarden.com/2015/10/25/an-engineers-guide-to-gemm/

https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/

Vitis HLS is the tool from AMD/Xilinx that generates HDL (Verilog or VHDL) from a C/C++ description of an application. **Vivado** is the tool from AMD/Xilinx that takes an HDL description of an application and performs a series of steps (synthesis, placement, routing) to generate a bitstream which can be configured onto an FPGA. There are a lot of tutorials for these tools online. But we will provide some presentations that you can go through. You can read more about these tools on the following links:

https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html

https://www.xilinx.com/products/design-tools/vivado.html

Here's the user guide for Vitis HLS: https://docs.xilinx.com/r/2022.1-English/ug1399-vitis-hls/Getting-Started-with-Vitis-HLS

In this lab, we will be using a board called the **PYNQ-Z2**. It has a Zynq-7000 series FPGA. This series of FPGAs is what is called an "SoC-FPGA". A microprocessor (ARM A9) is integrated with the FPGA fabric. The microprocessor can be used to run parts of the application, while other parts of the application are accelerated on the FPGA fabric. The microprocess can also control/observe the application running on the FPGA fabric. You can read more about this board and the FPGA on the following links:

https://www.xilinx.com/support/university/xup-boards/XUPPYNQ-Z2.html

http://www.pynq.io/board

Getting Started

For the labs in this class, you will mainly be using the ASU Research Computing Sol Cluster. Research Computing is a core facility within the Knowledge Enterprise's Research Technology Office dedicated to enabling research, accelerating discovery, and spurring innovation at ASU through the application of advanced computational resources to grand research challenges. Learn more at researchcomputing.asu.edu. Your account on the Sol supercomputer has been created and is ready to use.

Our supercomputers are accessed using your ASURITE login and password. For uninterrupted access to the supercomputer, please connect to the Cisco AnyConnect Client VPN. If you do not have the Cisco AnyConnect Client VPN already installed, visit sslvpn.asu.edu and follow the installation instructions. For additional details and troubleshooting, please read this document. Once connected to the VPN, you can access the supercomputer through the web portal: The web portal for the Sol supercomputer can be found at https://sol.asu.edu. Before submitting jobs to the supercomputers, please review the ASU Research Computing Acceptable Use Policy

Download the lab2.zip archive from Canvas, and transfer the archive to your home directory on the RC Sol Cluster. There are a couple of different options for this. The simplest option is to go to My Interactive Sessions. Ensure you are connected to the ASU VPN if you are not connected to the ASU network directly before visiting the link. Go to {Files > Home Directory to add files from your device to the home directory of your Research Computing account. Go back to My Interactive Sessions and start a Sol Desktop session. Always choose a general partition and a public QOS when starting a session. When choosing resources, remember that you are sharing the nodes will hundreds of other researchers, so be conservative with the number of cores, amount of memory, and session wall-time you choose. You will not need any GPU resources or additional sbatch options. Finally, click Launch and wait for your session to become active. After you launch your session, you can navigate to the directory where you placed the lab2.zip archive file and unzip it with the command: unzip lab2.zip -d <destination_directory>. The zip file provided with this lab document contains:

- A GEMM function with fp32 (float) inputs and outputs. Two files – matmul.cpp and matmul.h
- Two presentations from AMD/Xilinx – Vitis HLS Intro and Using Vitis HLS

[Part 1] Modify C++ code and write a testbench

Modify the C++ code to change the size of the input matrices to 16x16. That is, the code will support multiplying a 16x16 matrix with a 16x16 matrix to generate a 16x16 output matrix.

Develop a C/C++ testbench to test your GEMM function.

This testbench should be in a separate file. It should call the GEMM function and apply some inputs to it, and then compare the outputs with the expected values.

Compile and run the code to check if the GEMM is functioning correctly.

- Open a terminal on SOL machine.
- cd into the folder that contains your files.
- Compile the matrix multiplication code provided along with your testbench, by typing:
  - *g++ <your_testbench_file_name.cpp> matrix_mult.cpp -o matrix_mult.out*
- Execute the program by typing *./matrix_multi.out*

In the lab report:

- Provide a high level overview of the testbench in the report.
- Provide a snapshot of the testbench output

[Part 2] Perform HLS and Co-Simulation

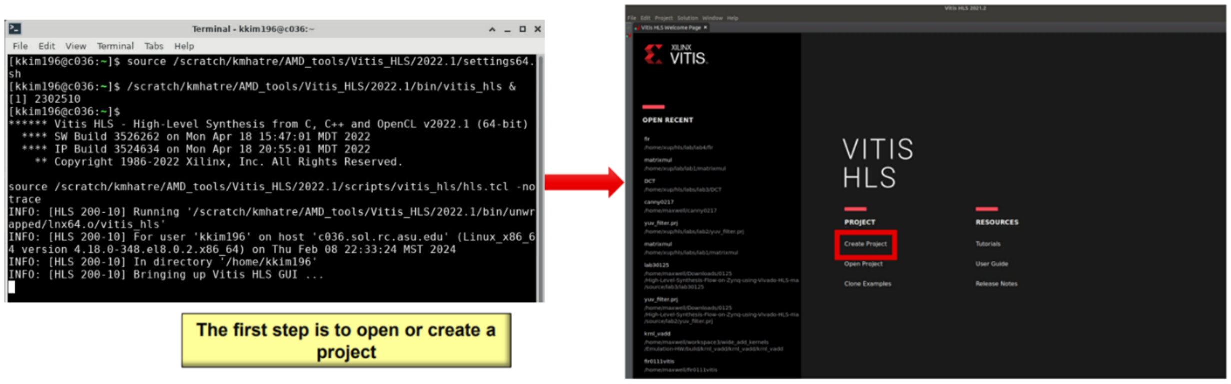
Start Vitis HLS 2022.1 by typing the following commands in the terminal:

```
source /scratch/kmhatre/AMD_tools/Vitis_HLS/2022.1/settings64.sh
/scratch/kmhatre/AMD_tools/Vitis_HLS/2022.1/bin/vitis_hls &
```

Creating a Vitis project

Create a new Vitis HLS project for your design with following settings:

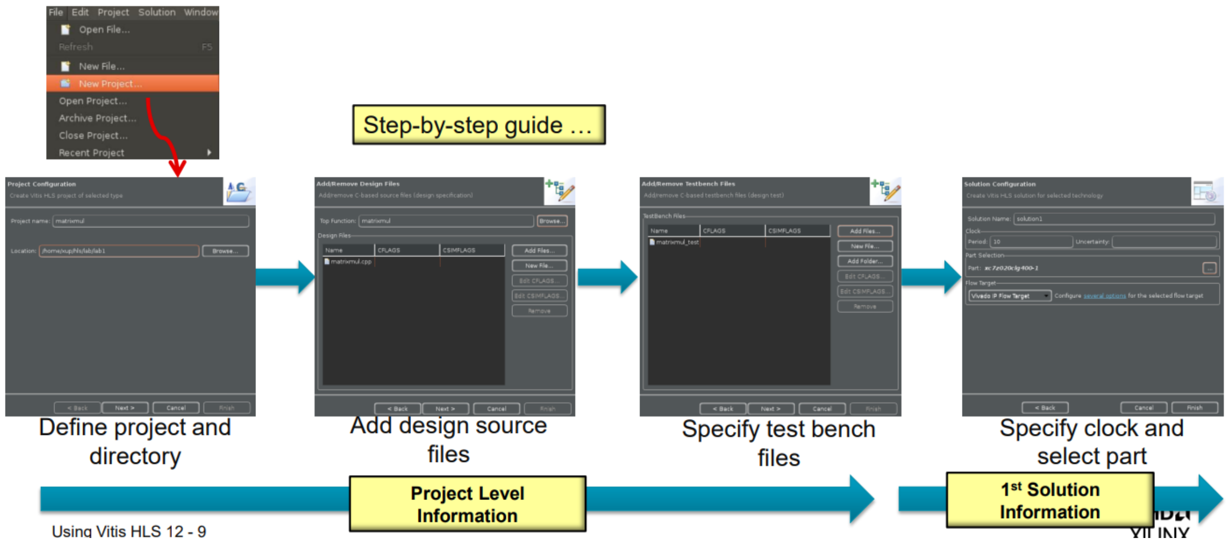
- Project name: hls_gemm (or whatever you want)



```
[kkim196@c036:~]$ source /scratch/kmhatre/AMD_tools/Vitis_HLS/2022.1/settings64.
sh
[kkim196@c036:~]$ /scratch/kmhatre/AMD_tools/Vitis_HLS/2022.1/bin/vitis_hls &
[1] 2302510
[kkim196@c036:~]$
****** Vitis HLS - High-Level Synthesis from C, C++ and OpenCL v2022.1 (64-bit)
  **** SW Build 3526262 on Mon Apr 18 15:47:01 MDT 2022
  **** IP Build 3524634 on Mon Apr 18 20:55:01 MDT 2022
    ** Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.

source /scratch/kmhatre/AMD_tools/Vitis_HLS/2022.1/scripts/vitis_hls/hls.tcl -no
trace
INFO: [HLS 200-10] Running '/scratch/kmhatre/AMD_tools/Vitis_HLS/2022.1/bin/unwr
apped/lnx64.o/vitis_hls'
INFO: [HLS 200-10] For user 'kkim196' on host 'c036.sol.rc.asu.edu' (Linux_x86_6
4 version 4.18.0-348.el8.0.2.x86_64) on Thu Feb 08 22:33:24 MST 2024
INFO: [HLS 200-10] In directory '/home/kkim196'
INFO: [HLS 200-10] Bringing up Vitis HLS GUI ...
```

The first step is to open or create a project

- Location: wherever you want, but it is recommended that you create your projects in your local scratch space under /blah/blah
- Top Function: the name of your top-level GEMM function
- Design Files: add your design files (including the header file) that are supposed to be synthesized
- TestBench Files: add your tsourcestbench files and input data files for test. These files are not synthesized.
- Solution Name: solution1 (keep this as is). In a Vitis HLS project, you can create multiple solutions, and each can have different synthesis options. And you can compare the solutions in Vitis HLS environment.
- Clock Period: There is no requirement for this part of the lab. A good starting point is 10 ns, but the target clock should eventually be one of the parameters driving optimizations.
- Part Selection:
  - Select: Parts -> Browse and select 'xc7z020clg400-1' (Assuming we are using the PYNQ-Z2 board that has the Xilinx Zynq 7000 MPSoC)
- Keep the "Flow Target" as "Vivado IP Flow Target"



Change code and run C simulation

Open the matmul code by double clicking on the file name in the Explorer pane on the left. Now edit the file by converting fp32 to int8 for inputs and int32 for outputs. For doing this, change float to ap_uint types in the header file (matrix_mult.h). Eg. ap_uint<5> is an unsigned 5-bit integer. You will need to include ap_int.h in the header file (matrix_mult.h).

Click Project -> Run C simulation. Check the simulation log. A successful simulation will have a "*****CSIM finish*******" message in the end.

Run HLS

You will synthesize the GEMM function using HLS. It will read inputs from local RAM and write outputs to local RAM.

In this part of the lab, don't specify any synthesis directives. Synthesis directives are also known as "pragmas". HLS Pragmas are added to the source code to enable the optimizations in generating different architectures of the hardware. Every time the code is synthesized, it is implemented according to the specified pragmas. You will be setting pragmas to explore different architectural alternatives in the next part of this lab.

Click on "Run C Synthesis" in the "Flow Navigator" in the bottom left of the screen. The settings in this window should match what you entered earlier when you created the project. Click Ok.

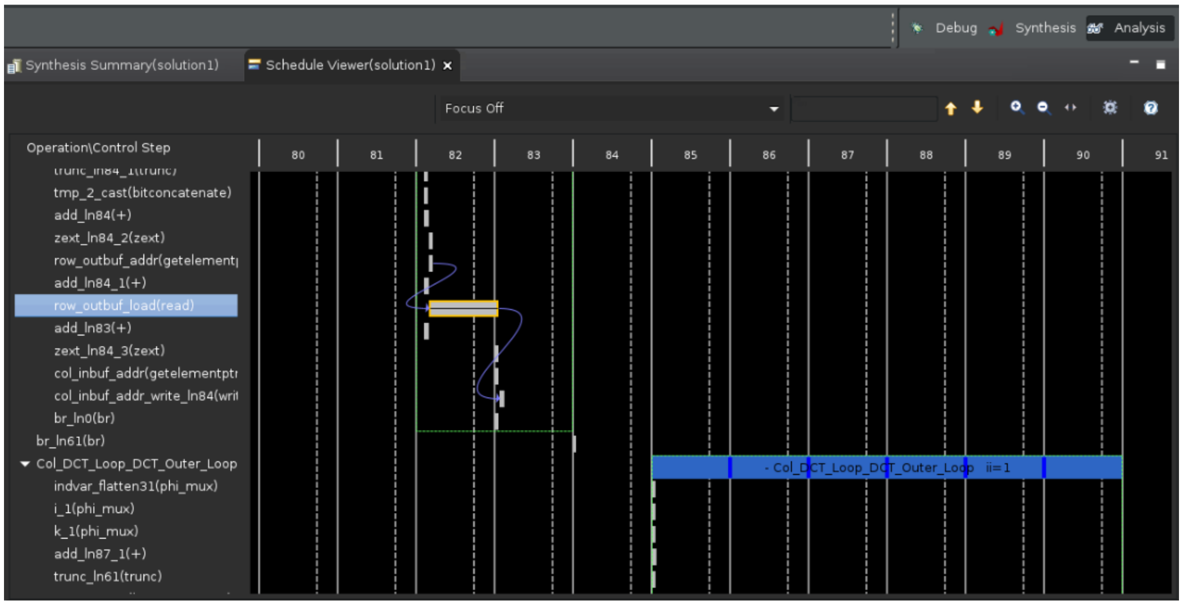
You will see a warning of an II violation. That's okay. Think about why this warning is being generated.

Observe HLS outputs

Observe the results of the synthesis report (resources used, estimated clock period, latency, throughput, etc) that are automatically shown in Vitis HLS after synthesis. How many DSP slices are used? What is the latency? What is the II? In the report, also see the interfaces of the design that were generated by HLS.

Open the schedule viewer, and observe the scheduling and binding done by Vitis HLS.

- ❖ In the “Flow Navigator” pane in bottom left, click “Schedule Viewer” under “C SYNTHESIS” -> “Reports & Viewers”. You can open other reports from here as well.
- ❖ The left side of the Schedule Viewer lists each operation in chronological order in the synthesized function. It displays the design control steps presented horizontally as a timeline starting at step 0 and running through to completion. You can select operations from the list to view the connections between them.
- ❖ The default view shows all of the operations. However, a drop-down menu at the top of the Schedule Viewer lets you select specific functions, loops, or elements of the design that are of interest.



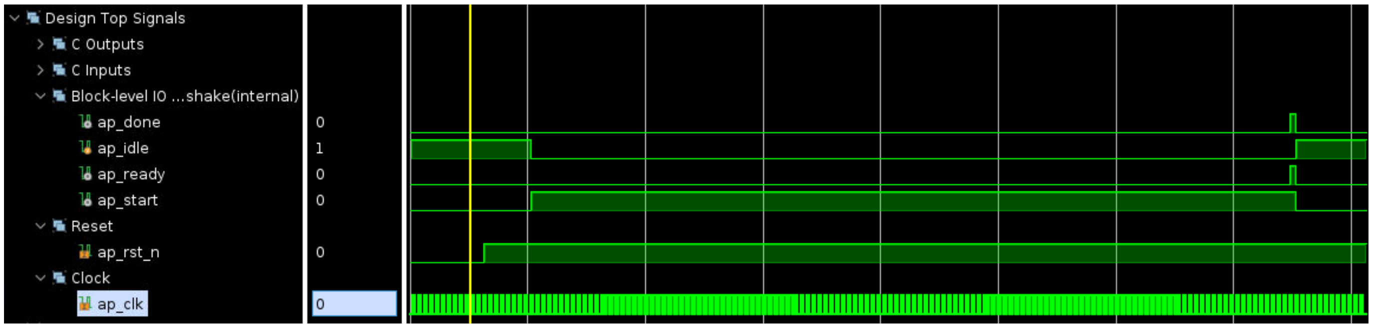
Observe the RTL generated by HLS. Expand “solution1” in the “Explorer” section on the left. Go down to “syn” and see the Verilog files. Browse through the files.

Validate your RTL code running C/RTL co-simulation. Click on Solution -> C/RTL Co-simulation. Set “Dump Trace” to “all” and check “Wave Debug”. Click OK.

Look through the console to see what happened.

A Xilinx Vivado window would have opened up. Look at the output from the simulation in the waveforms. You will need to open the folds in the waveform viewer to see the RTL signals of various parts of the design.

Look at the inputs being applied to the matrix multiplier design and the outputs being generated. Pay special attention to the block-level handshake signals. The signal ap_idle is set when simulation starts. The ap_start signal is asserted to kick off the computation. This lowers down the ap_idle signal. When the computation is finished ap_done and ap_ready are asserted for 1 cycle. Then, ap_idle is asserted by the design again denoting that the design is now ready to be started again.



Observe the testbench generated by Vitis. Expand "solution1" in the "Explorer" section on the left. Go down to "sim" and see the Verilog files. Locate a file named <>.autotb.v. Browse through it.

Try out implementation (which in Vitis HLS tool refers to both RTL Synthesis and RTL Place and Route). You can do this by right clicking on the solution in the Explorer and then clicking on "Implementation" and then following the dialog box. You can select whether you only want to stop at RTL synthesis or also do place & route. This will take a while. So go have a snack!

When it's done, click on "Report (RTL Synthesis)" or "Report (Place & Route)" (depending on what you did). See the achieved clock frequency and resources used. Compare the resource usage and frequency from the HLS report. Will the II or latency change after RTL synthesis compared to the HLS stage?

In the lab report:

- Mention the performance tradeoffs between using fp32 and using integer data types.
- Briefly explain how the RTL GEMM generated by Vitis works. You will do this by observing the synthesis report, the schedule viewer.
  - Provide snapshots of the synthesis report (it should have estimated clock period, performance, resource estimates, hardware interfaces).
  - Provide snapshots of the schedule viewer.
- Mention the name of each Verilog file generated by HLS and mention what each Verilog file contains. You don't have to understand everything in the file, but you should get an idea of what they have.
- Briefly explain how the RTL design is verified by co-simulation.
  - Provide snapshots of the co-simulation output.
  - Provide snapshots of the waveform viewer with the inputs, outputs and handshake signals.
- Create a table comparing the resources and clock frequency after HLS and after RTL synthesis.

[Part 3] Design Space Exploration using HLS

Now you will explore multiple different architectural alternatives of the GEMM design using the synthesis directives offered by Vitis HLS for unrolling, pipelining, and array partitioning. You will try each directive individually first, by changing values of unrolling or pipelining (II value) or partitioning. Just focus on the innermost loop (for unrolling and pipelining) and on the arrays (for partitioning) for now.

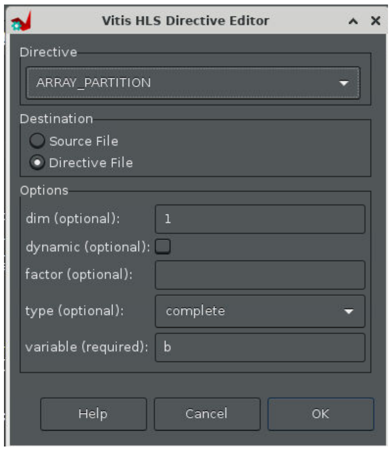
Create one solution in Vitis for each architectural alternative. To do this, click on Project -> New Solution. You can provide a custom name for each solution or just go with the default name. At any time you need to experiment with a previous solution, click on the solution folder in the explorer and click "Set Active Solution".

To add a directive, you can either add a line inline into the code, or use the GUI (which runs TCL commands in the background to add directives). The latter method is preferred, because then you don't have to maintain multiple code files for every solution. To do this, open the code by double clicking on the file in the "Explorer" pane on the left. you will see a "Directive" pane on the top-right panel. It will be next to the "Outline" pane. In the directive pane, you will see all arrays and loops. Right on any and click "Insert Directive". Configure the directive as you wish in the dialog box that opens up.

After configuring the directives, to run HLS, right click on the solution in the Explorer pane in the left and click "C Synthesis" -> "Active Solution".

Overview of Directives

❖ Array Partitioning
  ➢ Partitioning breaks an array into smaller arrays or individual registers to improve parallel access to data and remove block RAM bottlenecks



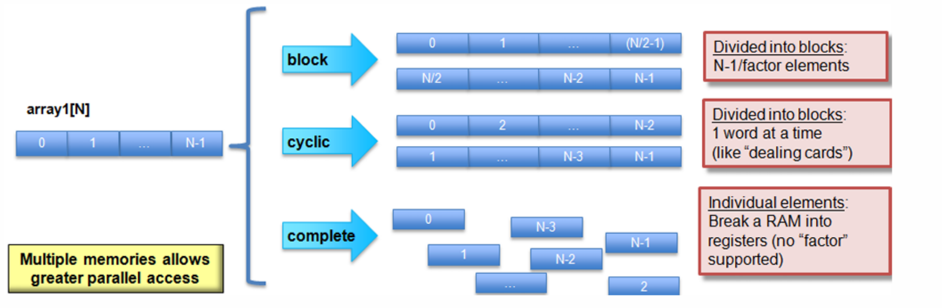
Vitis HLS Directive Editor

Directive: ARRAY_PARTITION

Destination:
- ( ) Source File
- (●) Directive File

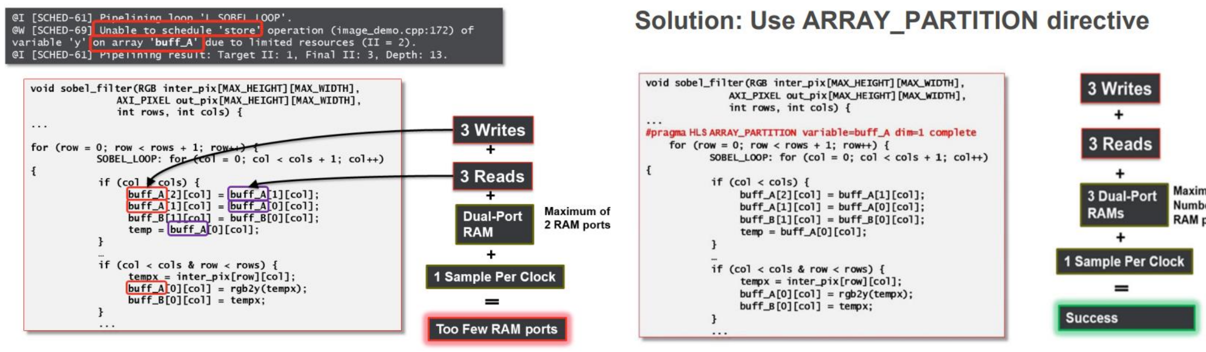
Options:
- dim (optional): 1
- dynamic (optional): ☐
- factor (optional):
- type (optional): complete
- variable (required): b

- Types of array partition:
  - Block: Original array is split into equally sized blocks of consecutive elements of the original array
  - Complete: Default operation is to split the array into its elements. This corresponds to resolving a memory into registers
  - Cyclic: Original array is split into equally sized blocks, interleaving the elements of the original array



array1[N]
0
1
...
N-1
Multiple memories allows greater parallel access
block
0
1
...
(N/2-1)
N/2
...
N-2
N-1
cyclic
0
2
...
N-2
1
...
N-3
N-1
complete
0
N-3
N-1
1
N-2
...
2
Divided into blocks: N-1/factor elements
Divided into blocks: 1 word at a time (like "dealing cards")
Individual elements: Break a RAM into registers (no "factor" supported)

  ➢ Bottleneck Example
    - Array accesses (block RAM) can be bottlenecks inside functions or loops
    - Prevents processing of one sample per clock



```
@I [SCHED-61] Pipelining loop 'L_SOBEL_LOOP'.
@W [SCHED-69] Unable to schedule 'store' operation (image_demo.cpp:172) of
variable 'y' on array 'buff_A' due to limited resources (II = 2).
@I [SCHED-61] Pipelining result: Target II: 1, Final II: 3, Depth: 13.
```

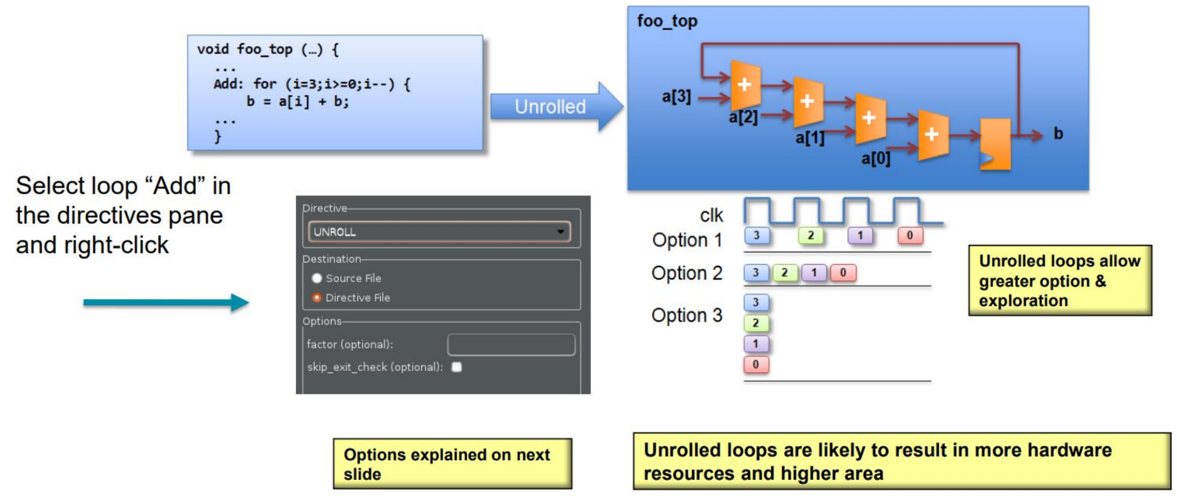
```
void sobel_filter(RGB inter_pix[MAX_HEIGHT][MAX_WIDTH],
                  AXI_PIXEL out_pix[MAX_HEIGHT][MAX_WIDTH],
                  int rows, int cols) {
...

for (row = 0; row < rows + 1; row++) {
           SOBEL_LOOP: for (col = 0; col < cols + 1; col++)
{
           if (col < cols) {
                  buff_A[2][col] = buff_A[1][col];
                  buff_A[1][col] = buff_A[0][col];
                  buff_B[1][col] = buff_B[0][col];
                  temp = buff_A[0][col];
           }
           …
           if (col < cols & row < rows) {
                  tempx = inter_pix[row][col];
                  buff_A[0][col] = rgb2y(tempx);
                  buff_B[0][col] = tempx;
           }
           ...
```

Solution: Use ARRAY_PARTITION directive

```
void sobel_filter(RGB inter_pix[MAX_HEIGHT][MAX_WIDTH],
                  AXI_PIXEL out_pix[MAX_HEIGHT][MAX_WIDTH],
                  int rows, int cols) {
...
#pragma HLS ARRAY_PARTITION variable=buff_A dim=1 complete
    for (row = 0; row < rows + 1; row++) {
          SOBEL_LOOP: for (col = 0; col < cols + 1; col++)
{
          if (col < cols) {
                buff_A[2][col] = buff_A[1][col];
                buff_A[1][col] = buff_A[0][col];
                buff_B[1][col] = buff_B[0][col];
                temp = buff_A[0][col];
          }
          …
          if (col < cols & row < rows) {
                tempx = inter_pix[row][col];
                buff_A[0][col] = rgb2y(tempx);
                buff_B[0][col] = tempx;
          }
          ...
```

The figure above shows the array partition pragma inserted in the code. But as suggested above, please use the method of specifying directives in the right panel. If you lose the panel, open the source code by clicking on the file name in the Explorer pane on the left side. Then when you click anywhere inside the code file, you will see the Directives pane appear on the right.

❖ Loop Unrolling
  ➢ Unrolled Loops can Reduce Latency



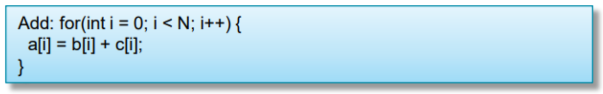
```
void foo_top (…) {
   ...
   Add: for (i=3;i>=0;i--) {
        b = a[i] + b;
   ...
   }
```

Select loop "Add" in the directives pane and right-click

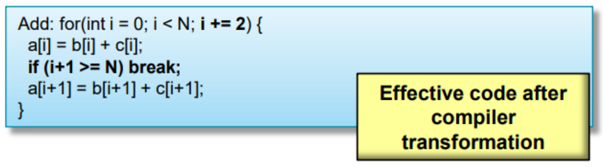
```
Directive
  UNROLL
Destination
  ○ Source File
  ● Directive File
Options
  factor (optional):
  skip_exit_check (optional): ☐
```

Unrolled
foo_top
a[3]
a[2]
a[1]
a[0]
b
clk
Option 1
Option 2
Option 3
3
2
1
0
Unrolled loops allow greater option & exploration
Options explained on next slide
Unrolled loops are likely to result in more hardware resources and higher area

  ➢ Fully unrolling loops can create a lot of hardware
  ➢ Loops can be partially unrolled
  ➢ In the figure, Option 1: Rolled loop, Option 2: Partially unrolled loop, Option 3: Unrolled loop
  ➢ Partial Unrolling
    ▪ A standard loop of N iterations can be unrolled by a factor smaller than N
      • For example, unroll by a factor 2, to have N/2 iterations
        ♦ Similar to writing new code as shown below
        ♦ The break accounts for the condition when N/2 is not an integer
  ➢ If "i" is known to be an integer multiple of N
        ♦ The user can remove the exit check (and associated logic)
        ♦ Vitis HLS is not always be able to determine this is true (e.g. if N is an input argument)

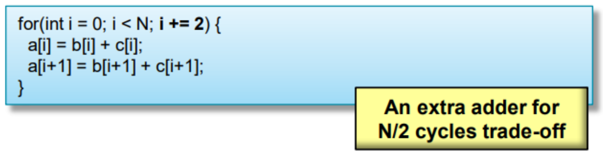


```
Add: for(int i = 0; i < N; i++) {
  a[i] = b[i] + c[i];
}
```



```
Add: for(int i = 0; i < N; i += 2) {
  a[i] = b[i] + c[i];
  if (i+1 >= N) break;
  a[i+1] = b[i+1] + c[i+1];
}
```

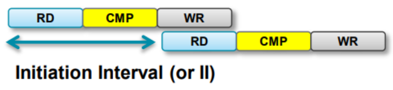
Effective code after compiler transformation

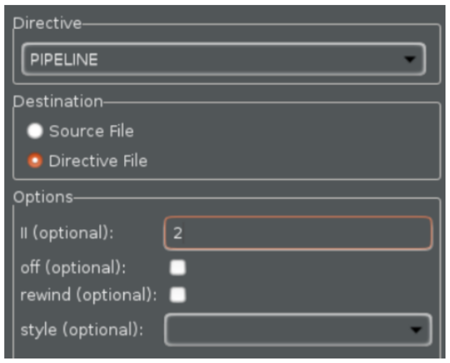


```
for(int i = 0; i < N; i += 2) {
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
}
```

An extra adder for N/2 cycles trade-off

- ❖ Loop Pipelining
  - ➢ This example pipelines the loop with an Initiation Interval (II) of 2
    - ▪ The II is the same as the "1/throughput" but this term is used exclusively with pipeline





Directive

PIPELINE

Destination

○ Source File

● Directive File

Options

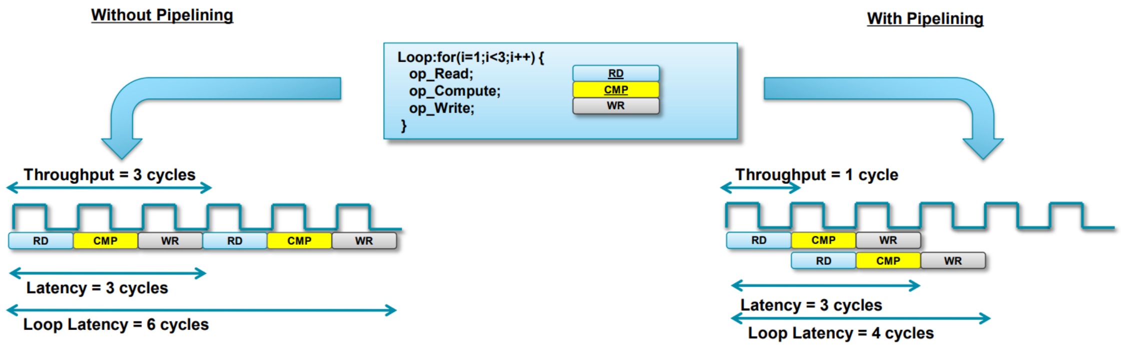
II (optional): 2

off (optional): ☐

rewind (optional): ☐

style (optional):

- ➢ Omit the target II and Vitis HLS will Automatically pipeline for the fastest possible design
  - ▪ Specifying a more accurate maximum may allow more sharing (smaller area)



```
Loop:for(i=1;i<3;i++) {
  op_Read;
  op_Compute;
  op_Write;
}
```

- ➢ Without pipelining
  - ▪ There are 3 clock cycles before operation RD can occur again
    - • Throughput = 3 cycles
  - ▪ There are 3 cycles before the 1st output is written

        - Latency = 3 cycles
        - For the loop, 6 cycles
- With pipelining
    - The latency is the same
    - The throughput is better
        - Less cycles, higher throughput
- The latency for all iterations, the loop latency, has been improved

Now use a combination of directives. Your goal is to come up with an area-delay optimal design. Create one solution in Vitis for each architectural alternative. There is a reasonably large design space here: **3 directives, 3 loops, 3 arrays, and many value settings of each directive.** Examples of value settings are factor=2 for array partitioning, unrolling with a factor=4, pipelining with an II=2, etc. You can try other values and settings. I suggest limiting yourself to the main settings of each directive. That is, I recommend not changing things that you don't understand (e.g. "rewind" in the pipelining directive pane, or "dynamic" in the array partitioning pane, or "skip_ext_check" in unroll pane).

Every time (when trying individual directives or combinations), you can open the schedule viewer to see what is going on. Understand how performance (area, delay, latency, throughput) changes as you specify different synthesis directives. Overall, you should try at least 8 solutions.

Can you reduce the achieved clock frequency by trying different directives?

Data collection

Draw a table showing the metrics – CLBs used, DSPs used, BRAMs used, Latency (cycles), Throughput or II (cycles), Clock Frequency (MHz) – for each architectural alternative.

Draw a chart plotting all of the solutions. The x-axis should be Area and y-axis should be Latency. Annotate the II number for each solution in this chart. Use the resource usage to generate an area estimate. Assume area of 1 CLB = 100 units, area of 1 DSP = 400 units and area of 1 18k BRAM = 400 units. Latency should be in units of time (ns or us or ms), not in cycles.

Can you see a pareto front from this plot of solutions?

In the lab report:

- Discuss your approaches to different solutions and compare them in terms of various design metrics, i.e. area (resource usage), latency, throughput, and operating clock frequency.
    - Specify what synthesis directive settings you used for each solution in the report.
    - Provide snapshots of the directives pane for each solution
    - Provide reasoning for changes that happen in each metric (resources, latency, throughput, operating frequency) for each synthesis directive that you use (loop unrolling, pipelining, array partitioning).
- Include the table and chart mentioned above (yellow highlight).
- Comment on the pareto front of solutions.

[Part 4] Integrating the GEMM accelerator with an on-chip CPU

In this section, we will create a simple GEMM accelerator and connect it with an on-chip CPU and control/observe the accelerator using the CPU.

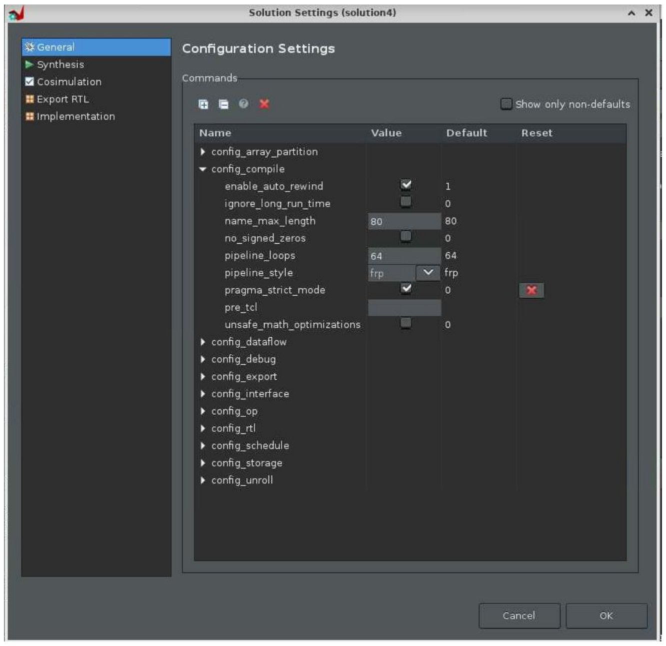
Make some code modifications

Before we do this, let's make some modifications to the code that will make it easier for us to build the system we want to build, and to workaround some tool issues. This is just being done for this lab. Otherwise, you don't need to make such changes.

Note: Our accelerator is not driven by performance goals. It's just a demonstration.

Create a new solution in Vitis.

Change 1: Open settings of the solution by right clicking on the solution in the Explorer pane and then click "Solution Settings". A window like shown below will appear. Check the box next to "pragma_strict_mode" and click Ok.



Name	Value	Default	Reset
config_array_partition			
config_compile			
enable_auto_rewind	✓	1	
ignore_long_run_time	☐	0	
name_max_length	80	80	
no_signed_zeros	☐	0	
pipeline_loops	64	64	
pipeline_style	frp	frp	
pragma_strict_mode	✓	0	✗
pre_tcl			
unsafe_math_optimizations	☐	0	
config_dataflow			
config_debug			
config_export			
config_interface			
config_op			
config_rtl			
config_schedule			
config_storage			
config_unroll			

Change 2: In your matrix multiplication header file, change the input precision to 32 bits and output precision to 32 bits (i.e. ap_uint<32>).

Change 3: Add these pragma settings for this solution by either editing the code or using the Directives pane.

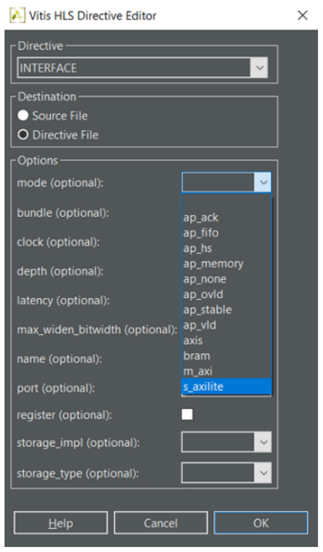
```
#pragma HLS INTERFACE s_axilite port=return bundle=control
#pragma HLS INTERFACE mode=bram port=a storage_type=ram_1p
#pragma HLS INTERFACE mode=bram port=b storage_type=ram_1p
#pragma HLS INTERFACE mode=bram port=prod storage_type=ram_1p
```

This adds an AXI Lite Slave port on your design that you can use to start/stop the accelerator. Through this port, you can:

- Write a register to start the GEMM accelerator block
- Read a register to observe that the GEMM accelerator block is done

Note that a "register" here does not refer to a Flip-Flop. The usage of the word "register" is similar to how in an embedded systems course, you write "registers" in various peripherals to perform operations.

AXI is a protocol for on-chip communication. There are 3 main types of AXI interfaces: AXI Memory Mapped, AXI Lite Memory Mapped and AXI Stream. We are using an AXI4-Lite slave memory mapped interface here (s_axilite). This will allow the processor to control and observe some parts of the matrix multiplier IP. For example, you can write "1" to bit 0 of the address offset 0x00 to assert the "ap_start" signal. You can read bit 2 of the address offset 0x00 to see if the matrix multiplier is idle or not.

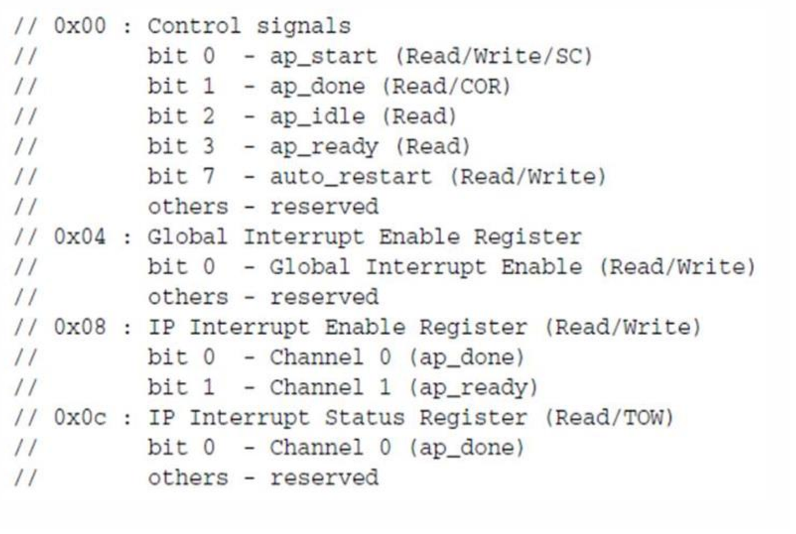


Vitis HLS Directive Editor

Directive: INTERFACE

Destination: (●) Source File ( ) Directive File

Options:
- mode (optional): s_axilite
- bundle (optional):
- clock (optional):
- depth (optional):
- latency (optional):
- max_widen_bitwidth (optional):
- name (optional):
- port (optional):
- register (optional):
- storage_impl (optional):
- storage_type (optional):



```
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/SC)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read)
//        bit 7  - auto_restart (Read/Write)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0  - Channel 0 (ap_done)
//        others - reserved
```

Also note that through the pragmas listed above, we are asking HLS to generate single-port "bram" ports for the interfaces. These will enable making connections easier later.

Run HLS and export the IP

Now, synthesize the solution. Observe the synthesis report. Note the hardware interfaces section specifically. You will a control interface (AXI slave lite) and a few BRAM interfaces.

Now, export the IP by clicking the "Export RTL" option under the "IMPLEMENTATION" section in the "Flow Navigator". Provide a path to save the exported IP. This is where we will import the IP from, in Vivado. Exporting IP will take some time. You will see a message when it's done.

Now you can close Vitis.

Create a Vivado project

Now, launch Vivado using the following commands:

```
source /scratch/kmhatre/AMD_tools/Vivado/2022.1/settings64.sh

/scratch/kmhatre/AMD_tools/Vivado/2022.1/bin/vivado &
```

Vivado is the tool that is typically used to write designs in Verilog and then run synthesis, place and route, and generate a bitstream. Vivado also provides a feature called the IP integrator. This makes doing IP based design much easier. You just have to drag and drop various IP blocks and connect them.

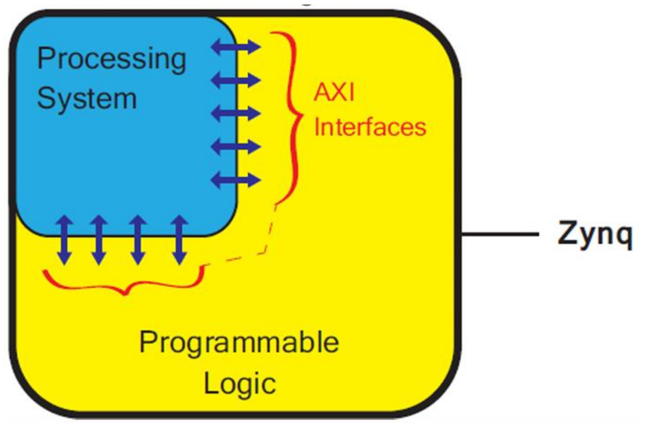
You can read more about the IP integrator here:

https://docs.xilinx.com/r/2020.2-English/ug994-vivado-ip-subsystems/Getting-Started-with-Vivado-IP-Integrator

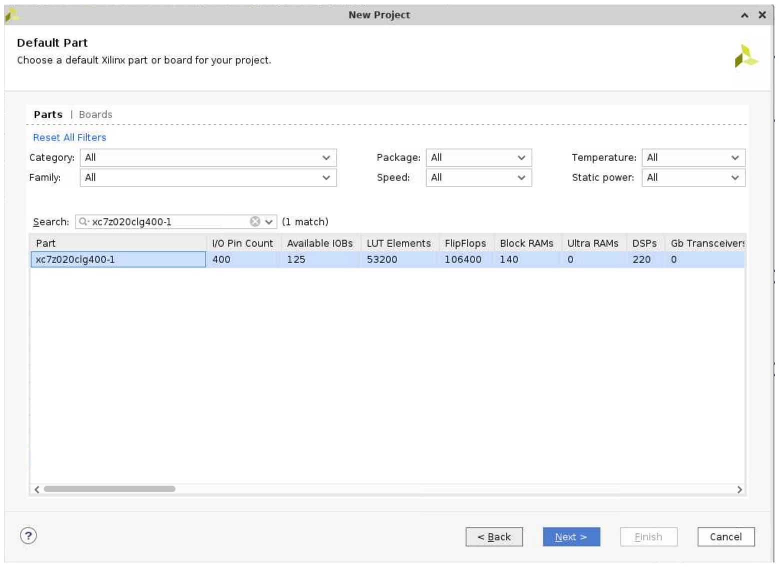
The FPGA we are using is what is called an SoC-FPGA. The Zynq™ 7000 SoC-FPGA family integrates the software programmability of an ARM®-based processor with the hardware programmability of an FPGA, enabling hardware acceleration controlled through a CPU.

There are two words that will be frequently used – The PS (processing system aka the CPU) and the PL (programmable logic aka the FPGA fabric).

- The defining features of Zynq:
    - Processing System (PS): Dual-core ARM Cortex-A9 CPU
    - Programmable Logic (PL): Equivalent traditional FPGA
    - Advanced eXtensible Interface (AXI): Hardware interfacing between PS and PL.
    - PS and PL can each be used for what they do best. PL fabric is good for static parallel tasks and peripheral controls. PS are more proper for dynamic tasks and complicated logic controls.



- Create a Vivado Project
    - Click Create Project to start the wizard. You will see the Create a New Vivado Project dialog box. Click Next.
    - Enter a name in the Project Name field and specify a project location. Make sure that the Create Project Subdirectory box is checked. Click Next.
    - In the Project Type form select RTL Project. Click Next. Do not specify sources and constraints at this time. Click Next on the next two screens.
    - In the Default Part window, select the ‘xc7z020clg400-1’ part.



Default Part

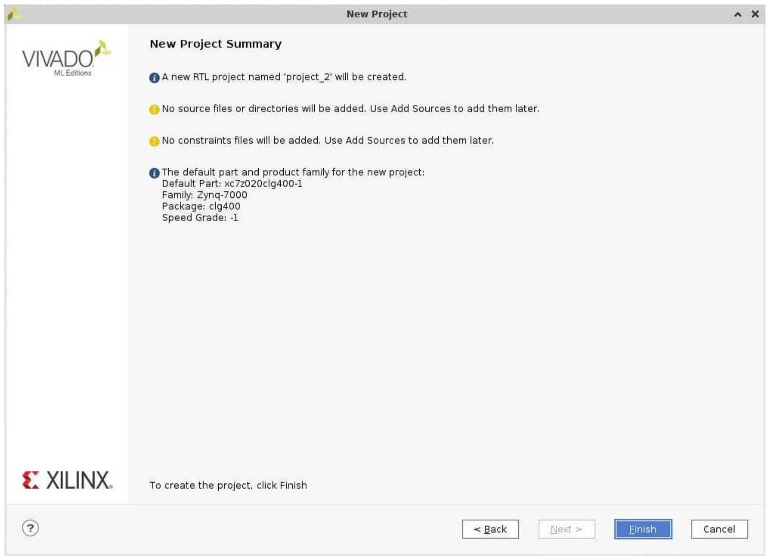
Choose a default Xilinx part or board for your project.

Category:	All	Package:	All	Temperature:	All
Family:	All	Speed:	All	Static power:	All

Search: xc7z020clg400-1 (1 match)

Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	Ultra RAMs	DSPs	Gb Transceivers
xc7z020clg400-1	400	125	53200	106400	140	0	220	0

➢ Check the Project Summary (should be similar to what you see below) and click Finish to create an empty Vivado project.



New Project Summary

A new RTL project named 'project_2' will be created.

No source files or directories will be added. Use Add Sources to add them later.

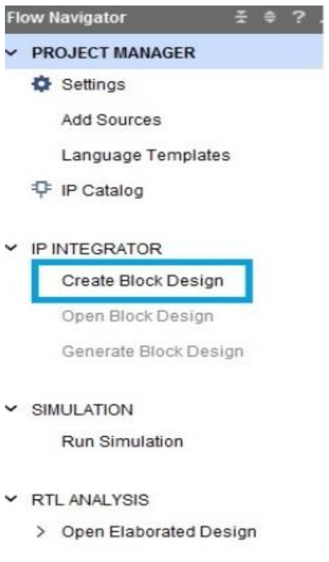
No constraints files will be added. Use Add Sources to add them later.

The default part and product family for the new project:
Default Part: xc7z020clg400-1
Family: Zynq-7000
Package: clg400
Speed Grade: -1

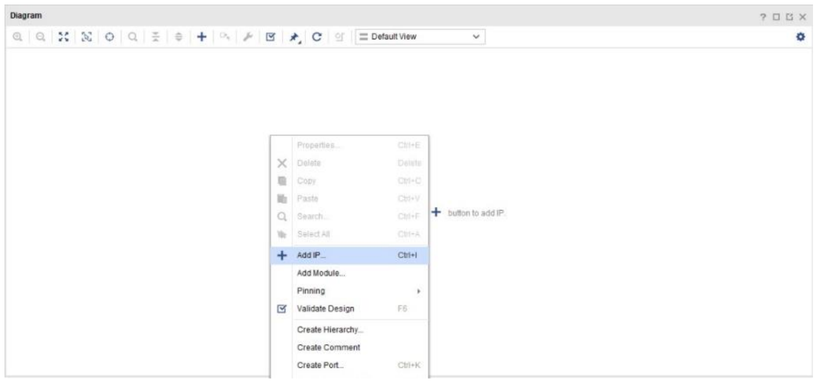
To create the project, click Finish

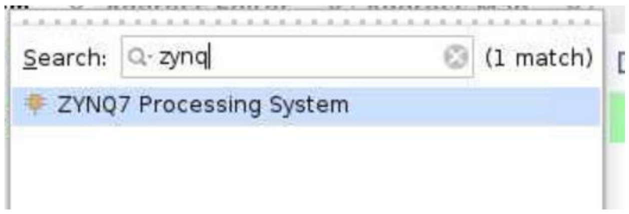
Use the IP integrator to connect various IPs including the matmul IP from Vitis

❖ Creating the System Using the IP Integrator
   ➢ In the Flow Navigator, click Create Block Design under IP Integrator.
   ➢ Enter any design name and click OK.
   ➢ Right-click anywhere in the Diagram workspace and select Add IP.

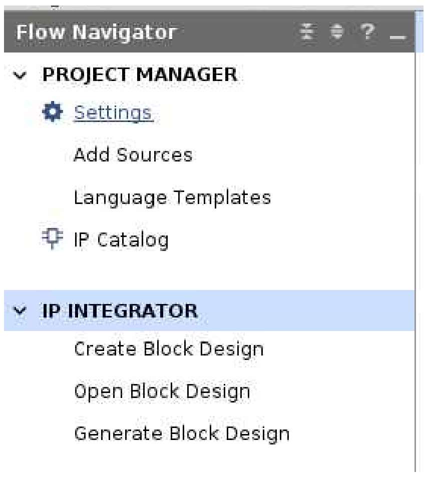


- ➢ When the IP Catalog opens after clicking the "+" button, type "zynq" into the Search bar, find and double click on ZYNQ7 Processing System entry, or click on the entry and hit the Enter key to add it to the design.

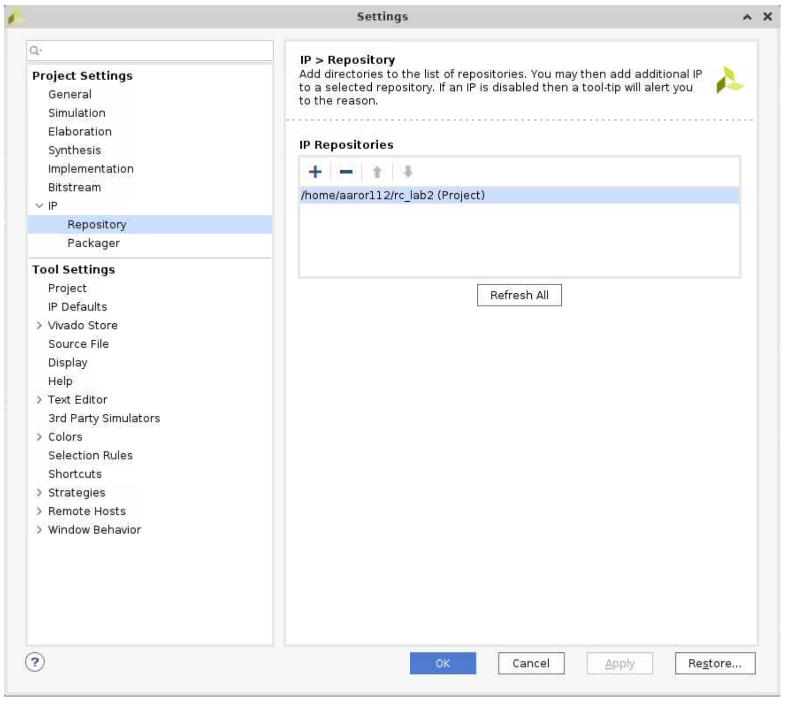




❖ Import your matrix multiplier IP
- ➢ Click on Settings in the Project Manager section in the Flow Navigator.



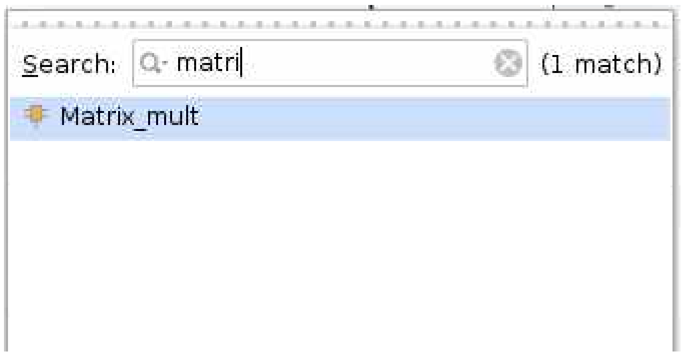
➢ Then click on the IP -> Repository and add the path of the folder where you exported the IP. Click ok.



IP > Repository
Add directories to the list of repositories. You may then add additional IP to a selected repository. If an IP is disabled then a tool-tip will alert you to the reason.

/home/aaror112/rc_lab2 (Project)

➢ Now go back to the block diagram. Right click in an empty space and click “Add IP”. Now search for the name of your IP. Press Enter to select and add it to the design.

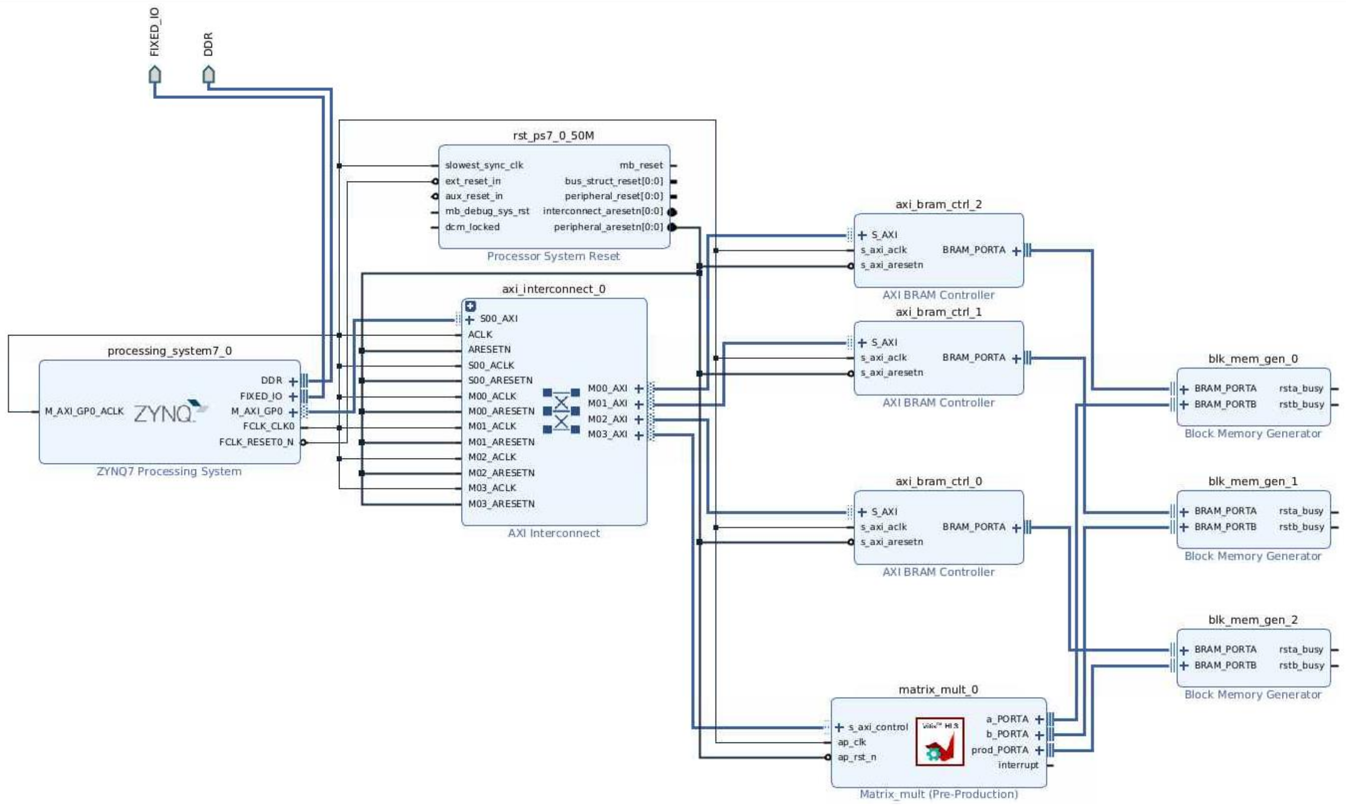


- Adding more IP blocks
  - After this, you need to add the following IP blocks:
    - 1 Processor System Reset
    - 3 AXI BRAM Controllers
    - 3 Block Memory Generators
    - 1 AXI Interconnect
  - When you insert the Block Memory Generator, you will see that you get a single-ported memory in the diagram. Right click on it and click "Customize Block". Then change the "Memory Type" to "True Dual Port RAM". Keep the "Mode" as "BRAM Controller".
  - Similarly, customize the AXI Interconnect block to have 1 slave interface and 4 master interfaces.

The execution model of the whole system is this:

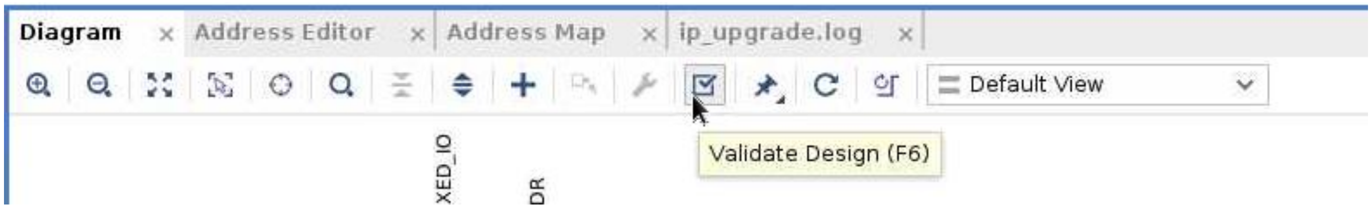
- The processor is the master. It is connected to one port of the block RAMs through the AXI interconnect and the AXI BRAM controllers. It is also connected to the matrix multiplier accelerator through the AXI interconnect.
- The matrix multiplier is connected to the processor on one side (through the AXI slave interface). It is connected to the second port of the block RAMs.
- The processor will execute a program (which we will write later) to write to the block RAMs and then trigger the matrix multiplier. The matrix multiplier will perform its operation (i.e. multiply matrices by reading two block RAMs and write the result into the third block RAM). Then the processor will read the result from the third block RAM.

- Connecting all the blocks
  - Now you have all the blocks you need. You need to connect these blocks. You can do this all manually, but the tool can help you. I generally connect the main things manually:
    - The connections between the matrix multiplier and the block RAMs
    - The connections between the matrix multiplier and the AXI interconnect
    - The connections between the AXI BRAM controllers and the AXI interconnect
    - The connections between the AXI BRAM controllers and the Block Memory Generator (Block RAMs)
  - After that, I use the automated connectivity maker. You will see a message at the top of the Diagram window in a green label saying that "Designer Assistance available. Run Connection Automation." Click on that, select all boxes in the window that appears and click Ok.
  - Once the connection automation completes, you will see connections were automatically done. You may need to fix them. Your eventual diagram will look like this:
  - You can do a right click on FIXED_IO and DDR of ZYNQ block (processing_system7_0) and click on "Make External" for each FIXED_IO and DDR.

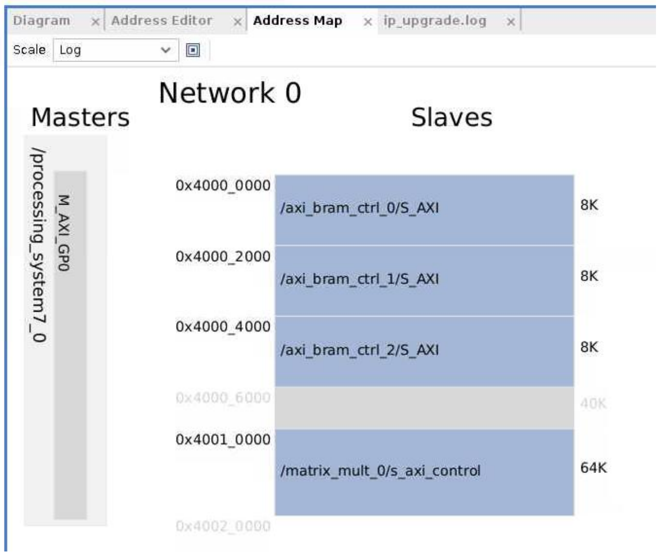


❖ Generate block design

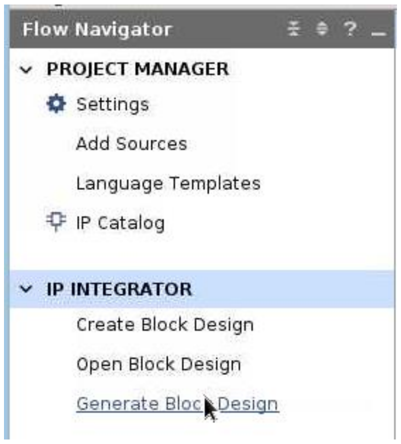
➢ Click on "Validate Design" to validate all the connections. It'll ask you that if you want to assign address map. Say yes.



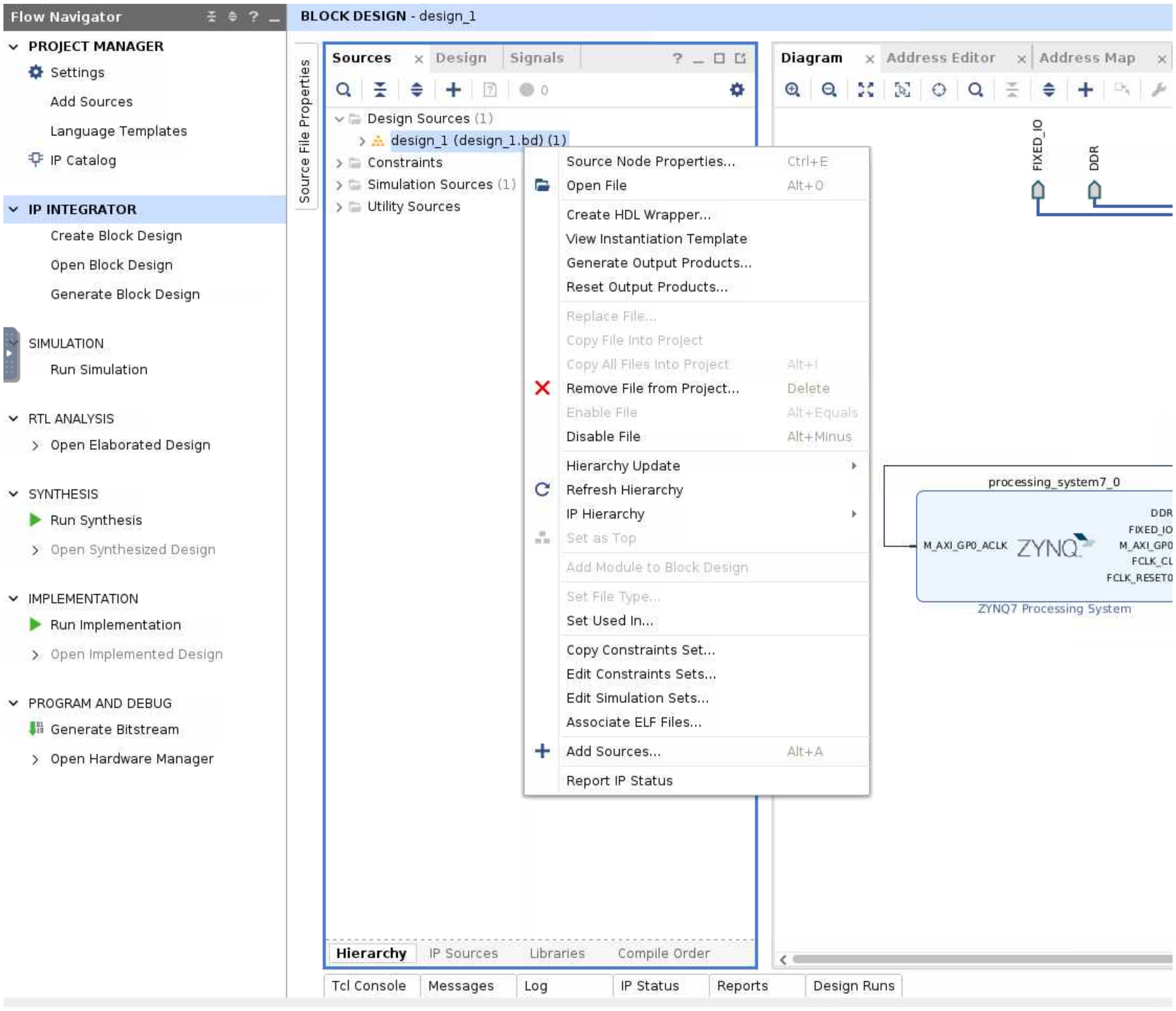
➢ After this is done, take a look at the address map as shown below. You will need the addresses of the block RAMs and the matrix multiplier to write and read values to them (from the CPU side, using Python, in the Pynq environment, later).



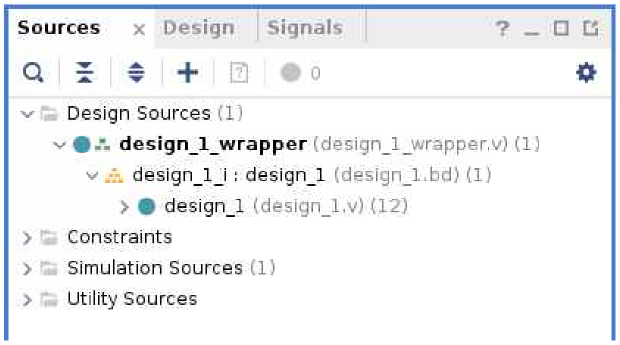
- ➢ Click on “Generate Block Design” in the Flow Navigator. Then click Ok on the resulting window.



- ❖ Generate top level wrapper
  - ➢ Vivado doesn’t use the block design as the top level. So, we need to add a wrapper. Right click on the block design name in the “Sources” pane, then click “Create HDL Wrapper”.
  - ➢ Keep the default settings (“Let Vivado manage wrapper and auto-update”) and click Ok.



After this you'll see a wrapper over the block design:


Sources
Design
Signals
Design Sources (1)
design_1_wrapper (design_1_wrapper.v) (1)
design_1_i : design_1 (design_1.bd) (1)
design_1 (design_1.v) (12)
Constraints
Simulation Sources (1)
Utility Sources

Generate the bistream for this design

- Generate Bitstream and Export Bitstream File
  - Click on Generate Bitstream, and click Yes if prompted to Launch Implementation (Click Yes if prompted to save the design). This will take some time. Go have a snack!
  - Export the bitstream by clicking File > Export > Export bitstream. Save this file in a folder. This will be a .bit file.
  - Export the block design by clicking File > Export > Export Block Design. Save the file to the same folder as above. This is a .tcl file. If it has a different name, rename it to have the same name as the bitfile (of course, it's extension.
  - Go to the directory that contains the Vivado project. Let's say it is called <proj_name>. Then, go to <proj_name>.gen/sources_1/bd/design_1/hw_handoff. There will be a .hwh file. Copy that to some location and rename it so its name matches the same of the bitfile (.bit).
  - Download these .bit, .hwh and .tcl files from the SOL cluster machine to your local desktop/laptop.

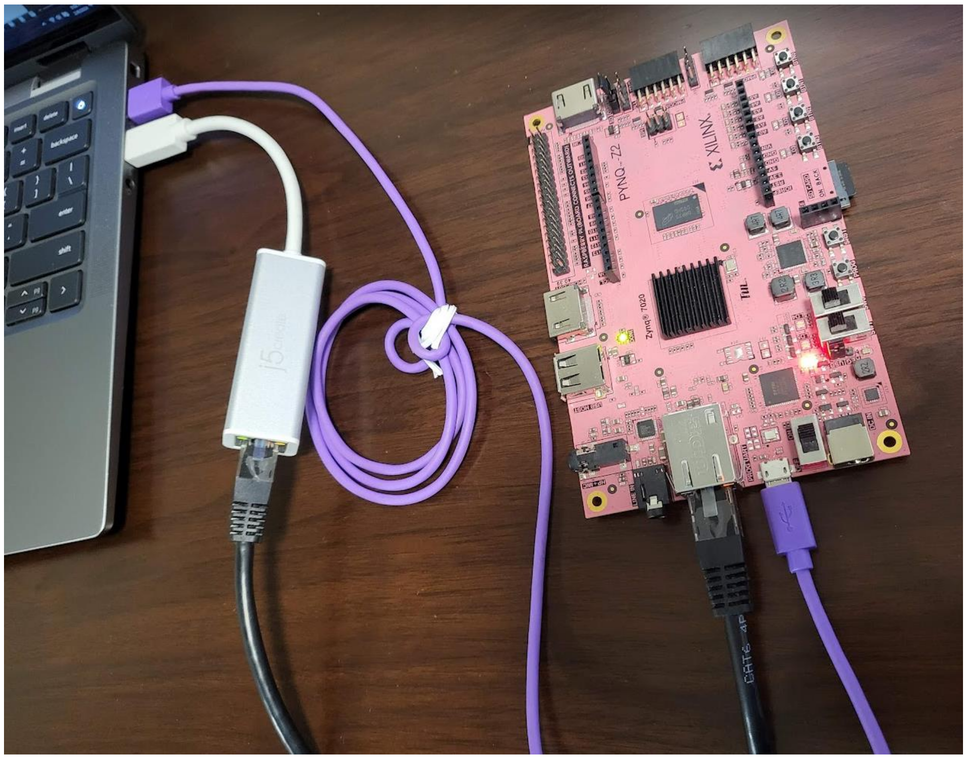
The whole design on the FPGA is referred to as the "overlay" in the rest of the document. The "overlay" mainly contains the Zynq PS, the matrix multiplier (or GEMM) IP block and the block RAMs.

Open Jupyter notebook on Pynq board

We will start working on with an FPGA board (a PYNQ board). We will provide the boards to you (we will likely have 8-9 boards that all groups will share). They will already be set up.

Here's the basic setup you need to be aware of.

- Connect the mini-USB (purple cable in the figure below) to the board and a computer.
- Connect the ethernet port to an ethernet port on your computer. If you don't have an ethernet port on your laptop, use an Ethernet-to-USB cable as shown in the figure below (black and white wire).
- Switch the power button to the ON position.
- The board has a static IP address 192.168.2.99. You need to set the IP address of your laptop/pc to be in the same range as the board. That is, the laptop/PC can be 192.168.2.x where x is 0-255 (excluding 99, as this is already taken by the board).
  - To do this (on a Windows machine), go to Settings > Network & internet > Ethernet. Click on the ethernet network that just appeared when you connected the board. Then click on IP assignment and change it to manual. Set the IP address to 192.168.2.1 (or any other address in the same range as the board). Set the subnet mask to 255.255.255.0. Leave everything else as-is (blank). Click Ok.



You can read more about setting up the board here: [Getting Started — Python productivity for Zynq (Pynq)]()

Now open a web browser like Chrome or Edge and type 192.168.2.99 (the IP address of the board) to open Jupyter Notebook. The initial password is "xilinx" (all lower case). Create a folder by clicking New. Give it a name (let's say 'lab2'). This moves these files into the SD Card in the PYNQ board. The path of this folder is /home/xilinx/jupyter_notebooks/<your_folder_name>.

Now click the "Upload" button on the Jupyter notebook and upload the bit file, tcl file and the hwh file you downloaded earlier.

Make sure tcl file and hwh file are in the same directory (folder) on PYNQ board. For example, if you stored your bitfile to "/home/xilinx/ jupyter_notebooks/gemm/your_bit.bit" then they should be in same directory and have same name namely "/home/xilinx/jupyter_notebooks/gemm/your_bit.hwh" and "/home/xilinx/jupyter_notebooks/gemm/your_bit.tcl".

Now start a new notebook by clicking "New" and then "Python 3". Now, from this notebook you will control and observe the GEMM accelerator IP using the PYNQ environment.

Note that this Jupyter code is running on the microprocessor in the PS on the FPGA. By executing commands in the Jupyter notebook, you are basically running Python code on the processor.

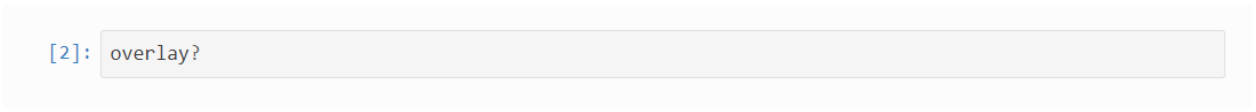
Inspect the overlay

The following steps show an example of how to interact with the IP using the PYNQ environment.

- To interact with the IP first we need to load the overlay containing the IP.

```
[1]: from pynq import Overlay

     overlay = Overlay('/home/xilinx/tutorial_1.bit')
```

- Creating the overlay will automatically load it on the Programmable Logic (PL) part of the SoC FPGA. We can now use a question mark to find out what is in the overlay.



```
[2]: overlay?
```

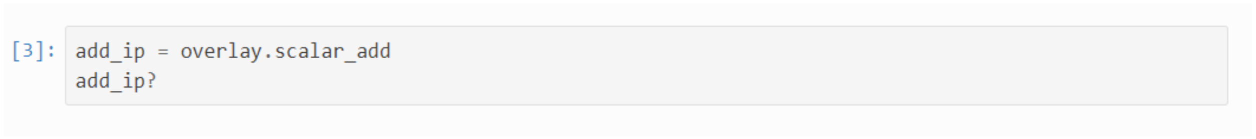
For the tutorial_1.bit file that was loaded, it contained an IP called scalar_add and the processing system.

In your GEMM accelerator case, you will see a window pop up with information similar to the following:

```
IP Blocks
----------
axi_bram_ctrl_0       : pynq.overlay.DefaultIP
axi_bram_ctrl_1       : pynq.overlay.DefaultIP
axi_bram_ctrl_2       : pynq.overlay.DefaultIP
matrix_mult_0         : pynq.overlay.DefaultIP
processing_system7_0 : pynq.overlay.DefaultIP
```

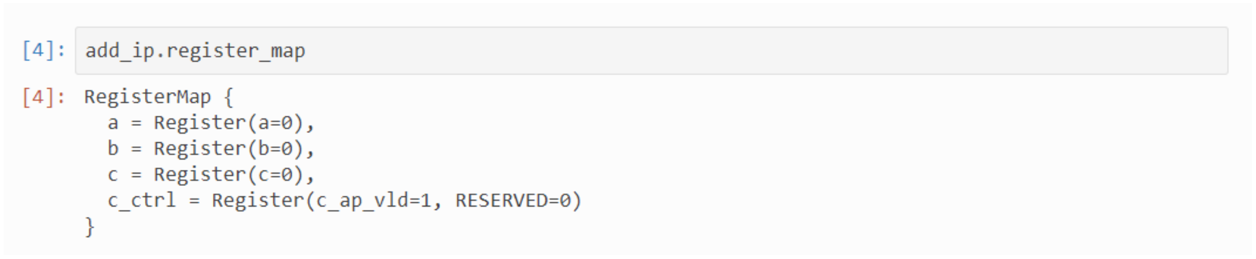
This shows the 3 BRAMs, the matrix multiplier and the processing system (PS). That's what we had put together in the IP integrator.

- All of the entries are accessible via attributes on the overlay class. To access the scalar_add IP, we just use the scalar_add attribute of the overlay. Then we can inspect what's in it by doing "add_ip?".



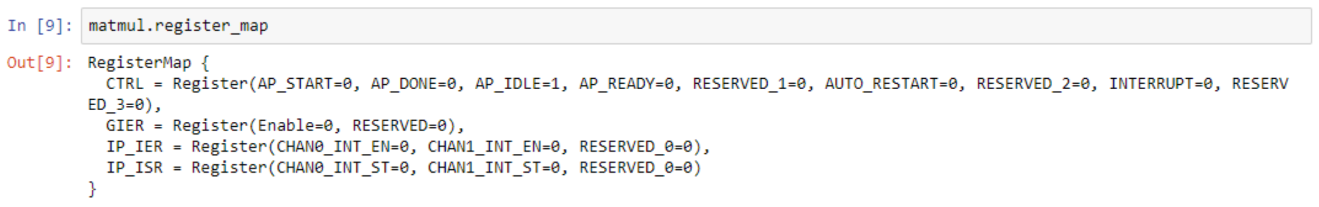
```
[3]: add_ip = overlay.scalar_add
     add_ip?
```

- You can see the register map associated with IP as well by using the "register_map" attribute like so:



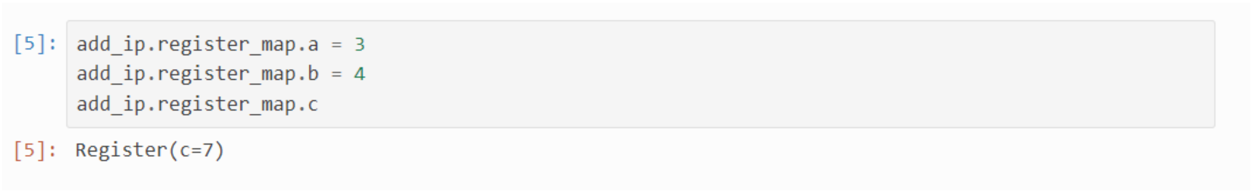
```
[4]: add_ip.register_map

[4]: RegisterMap {
       a = Register(a=0),
       b = Register(b=0),
       c = Register(c=0),
       c_ctrl = Register(c_ap_vld=1, RESERVED=0)
     }
```

For the matmul/GEMM accelerator, you will see something like:



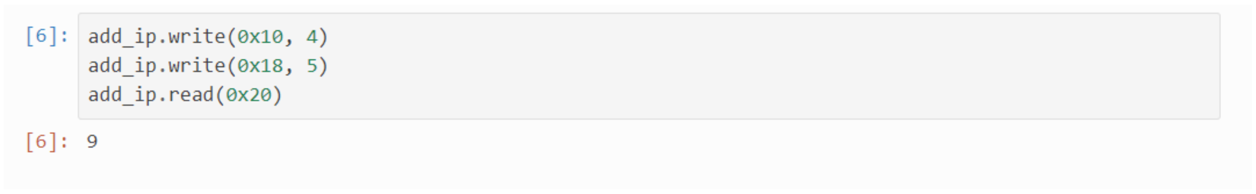
```
In [9]: matmul.register_map

Out[9]: RegisterMap {
          CTRL = Register(AP_START=0, AP_DONE=0, AP_IDLE=1, AP_READY=0, RESERVED_1=0, AUTO_RESTART=0, RESERVED_2=0, INTERRUPT=0, RESERV
        ED_3=0),
          GIER = Register(Enable=0, RESERVED=0),
          IP_IER = Register(CHAN0_INT_EN=0, CHAN1_INT_EN=0, RESERVED_0=0),
          IP_ISR = Register(CHAN0_INT_ST=0, CHAN1_INT_ST=0, RESERVED_0=0)
        }
```

- We can interact with the IP using the register map directly. For example, to set the register "a" to 3 and register "b" to 4, and to print the register "c", you can use the following commands:



```
[5]: add_ip.register_map.a = 3
     add_ip.register_map.b = 4
     add_ip.register_map.c

[5]: Register(c=7)
```

- ➢ Alternatively by reading the information of the address space of the IP generated by HLS, we can determine that offsets of each register. As an example, the following code writes 4 to address offset 0x10, 5 to address offset 0x18 and reads address offset 0x20.



```
[6]: add_ip.write(0x10, 4)
     add_ip.write(0x18, 5)
     add_ip.read(0x20)

[6]: 9
```

Based on the steps listed above, inspect the overlay that you loaded to this FPGA. See what IPs exist in the overlay. In the register map of the matmul/GEMM IP in the overlay, you will see address locations that you saw earlier in the address map in Vivado.

Write code and evaluate speedup

Now write two sets of Python code:

(A) Software-only GEMM execution

This will be a Python translation of the C++ code we provided for the lab (but with int32 inputs and int32 outputs). Running this will execute the matrix multiplication operation on the CPU in the Zynq PS. No hardware IP involved. You should generate random matrices of the required size and compute the matrix multiplication of these matrices.

Run this Python code in the Jupyter notebook on the Zynq processor. Measure the time taken. You can use the time.time() to get current time. Make sure you include the library “import time” on the top of your python code on Jupyter Notebook.

(B) Software+Hardware GEMM execution

Write Python code to:

- Generate random matrices of the required size.
- Write these values into the input BRAMs of the GEMM accelerator
- Write a value to a register address in the Matmul IP in the overlay to trigger the computation
- Read a register address to observe that the GEMM accelerator block is done. *(note that for this part, you should just observe the ap_idle bit, instead of the ap_done bit because the ap_done bit is not latched into the register in the IP as seen from the waveforms of cosimulation. We could change the logic to have the ap_done latched. That’d complicate the design. We are keeping things simple.)*
- Read data from the output BRAM of the GEMM accelerator. Compare the results to the expected values. Ensure that the output coming out of the accelerator matches the expected value.

Run this code that performs GEMM on the accelerator. Measure the time taken.

For this part of the lab, you can get extra credit of 5% by doing both the things below:

1. The various commands we described above to interact with the overlay are very simplistic. Create classes and functions (composed of these basic commands) to design a "driver" to interact with the matrix multiplier and BRAMs in the overlay.
2. There are many inefficiencies in this accelerator design flow. You can change something either during HLS or during the block design, and demonstrate speedup obtained by using your changes. (For example, in this part, we started with a simple set of HLS pragmas which results in a design with low performance. But can you use the most performant HLS design from Part 3 in this part of the lab?)

In the lab report:

- Provide a snapshot of your GEMM accelerator modified with inline pragmas for this part of the lab
- A table comparing the time taken for the Software-only and Software+Hardware execuction.
- Explain why is one higher than the other. What can be done to reduce the Software+Hardware execution time?
- List some things that can be improved in this accelerator design (right from the HLS stage, all the way to actually running it on the board).

Deliverables

Create a zip file containing 4 folders (part1, part2, part3, part4) and a report. Name this file lab2_ <firstname1>_<lastname1>_<firstname2>_<lastname2>.zip. Upload this file to Canvas. See below for details.

Code

Provide all your code (.cpp, .h, .py, etc.). Include any input/output or test files. Separate them into folders: part1, part2, part3, part4. In each folder, provide a README file on how to compile/run/verify your code. The TA should be able to reproduce the code.

For part3, include the directives.tcl file for each solution. These are present under the Solution_# directory.

For part4, provide the .tcl, .bit and .hwh files.

Report

This should be a PDF file. One file separated into sections. One section for each part. Provide the information asked in each part.

Explicitly in red color, mention anything you've done to get extra credit.

How will you be graded

- The correctness, completeness and performance of your solutions/code (50%)
- The correctness, completeness and clarity of your report (50%)
    - Please make sure you include the reasoning/commentary about your observations of the results. If you just put the results in the report without any discussion/commentary/conclusions, you will not get points.