

ECE 340
Embedded Systems

Spring 2020

Lab 5

SoC design and optimization

1. Introduction

FPGA fabric can be used to implement hardware accelerators to offload computationally demanding tasks from the CPU. In this lab, you will first run and profile a software application (Bilateral filter) on the ARM processor and you will profile the application to assess its performance footprint. You will perform software optimizations to improve its execution time. However, software running on a processor, no matter how well optimized it is, is usually slower than a well-designed hardware accelerator implementing the same functionality. Based on the outcome of software profiling you will implement the Bilateral filter in an accelerator to reduce its execution time using Vivado HLS (High-Level Synthesis) and the OpenCL API of Vitis Unified Software Platform. Optimizations in hardware can provide additional performance gains. As a last step, you will employ approximate computing techniques to further increase the performance of the hardware accelerator. Approximate computing trades off output quality (i.e. by accepting small errors) with performance.

This lab will be implemented using the Vitis Unified Software Platform (v2019.2), targeting the Xilinx UltraScale+ MPSoC ¹ZCU102 Evaluation Kit. The FPGA includes a quadcore 1.2 GHz ARM Cortex-A53 processor paired with 4 GB DDR4 main memory and is based on Xilinx's 16nm FinFET+ programmable logic fabric. The ARM processors are running Petalinux (a Linux distribution for Xilinx FPGA boards). The access to the ZCU102 board will be done remotely using the GitLab CI/CD ²facility. More details are included in the Appendix of this document. For this lab, you must be familiar with the Vivado HLS flow. Therefore, you need to do the exercises in the HLS tutorial focusing on the following chapters: ch.1 (tool flow), ch.2 (HLS Introduction), ch. 4 (Interface Synthesis), ch. 6 (Design Analysis), ch. 7 (Design Optimization), ch. 9 (Using HLS IP in IP Integrator) and ch. 10, (Using HLS IP in a Zynq Processor Design, lab1 only).

There are two types of deliverables for Lab5. First, the source code for the optimal software and hardware designs organized in two different directories. Second, a detailed report that outlines your experiments, presents the results using tables and figures, and discusses and explains these results. The quality of your report and your final presentation are important.

2. Bilateral filter

The bilateral filter is a stencil-based, edge-preserving image filter that blurs the input image to reduce the effects of noise and invalid depth values (see Figure 1). This filter replaces the intensity of each pixel of the input image with a weighted average of intensity values from nearby pixels

¹ Multiprocessor System On Chip

² Continuous Integration/Continuous Deployment

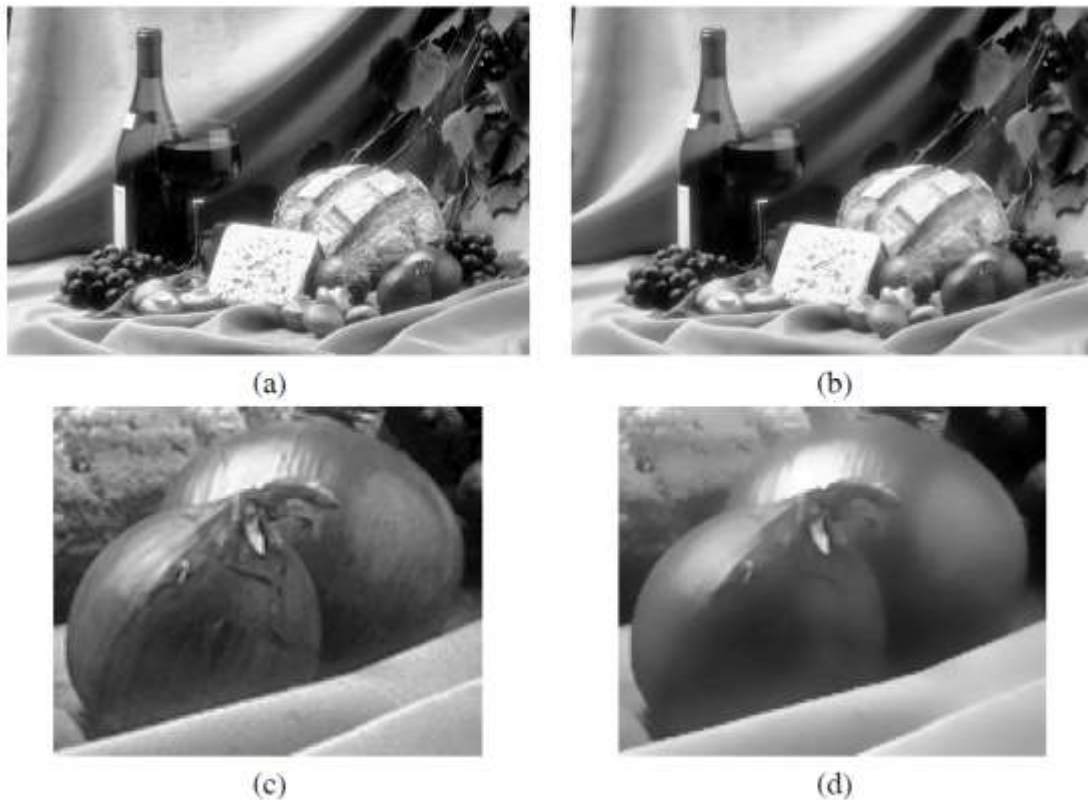


Figure 1. A picture before (a) and after (b) bilateral filtering. Images (c) and (d) are details from (a) and (b).

(see Figure 2). The Bilateral filter operator consists of a 5×5 window of coefficients and follows Gaussian distribution.

According to Wikipedia, *Adobe Photoshop* implements a bilateral filter in its *surface blur* tool. *GIMP* implements a bilateral filter in its *Filters-->Blur* tools; and it is called *Selective Gaussian Blur*.

3. Bilateral Filter in Software (x86)

In this initial step, you will run the Bilateral filter C code (using the *lab5-software* zip file) with the given 320×240 input frame (*input.bin*) on an Intel x86 processor (your laptop/desktop or the ones in the lab). The objective of this step is to familiarize yourself with this application, to profile and analyze the executed code, and to discuss the results. The input frame is a depth image that has been captured by a Microsoft Kinect RGB-D camera. A depth image is an image that contains information relating to the distance of scene objects from the camera (Figure 3). The depth image in *input.bin* is a 2D array of float numbers that shows the depth in meters.

In this phase, you should compile the C source code, run the optimized code in a variable number of CPU cores, and record the execution time using the TICK and TOCK macros³. Make sure to record the single-threaded execution time as well as combinations of multiple threaded executions. The number of threads depends on how many cores are available in your CPU. For example, if your CPU has 4 cores, then you can deploy up to 8 threads in your execution (Intel's x86 CPUs are hyper-threaded, i.e. they can run 2 threads per core). You can indicate the number of threads to be used by the OpenMP runtime by setting the environment variable `OMP_NUM_THREADS` from the Linux shell prompt (or, equivalently, from the Makefile):

If you use the Bash shell: `export OMP_NUM_THREADS=N` % set N = 1, 2, 4, 8...

If you use the csh/tcsh shells: `setenv OMP_NUM_THREADS N`

An interesting observation is that in all previous experiments the OpenMP runtime system made a static, compile-time partitioning of loop iterations. For example, each core is assigned $(320 \times 240)/4$ loop iterations, typically by partitioning the outer loop into four sections (this is in case you have 4 threads). However, in a lot of cases, the workload is dynamic and imbalanced, which means that not all iterations require the same execution time. In that case, the static OpenMP partition is not optimal. For dynamic scheduling, you may want to use the following OpenMP `#pragma` which appends the `schedule(dynamic)` clause at the end of the OpenMP pragma.

```
#pragma omp parallel for shared(out), private(y,i,j,x,pos)
schedule(dynamic)
```

We recommend that you try various OpenMP scheduling approaches to evaluate their effect on performance⁴.

Finally, you may want to experiment with various compilation flags to apply specific optimizations. In some cases, these flags can give you significant speedup.

In this and the following experiments of Lab5, you need to record the execution times of all your designs (software and hardware) in (i) a table and (ii) a graph (e.g. excel spreadsheet). This will enable a fast comparison between different designs and will make it easier to write your final report.

4. Bilateral Filter in Software (ARM Cortex A-53)

In the second step of Lab5, you will run your code in the ARM Cortex A-53 CPU in the ZCU102 FPGA. You can use the same source code and the same OpenMP `#pragmas` as in section 3. From this section onwards, you will need to use the ZCU102 FPGA as described in the Appendix. Again, as in section 3, you will need to use the GitLab CI/CD to compile the C code, run the optimized

³ Use the instructions in file *compile_instructions.txt* to compile your code for x86 CPU. Do not use the *Makefile*

⁴ <http://jakascorner.com/blog/2016/06/omp-for-scheduling.html>

code in one as well as in all four ARM cores, and record the execution time using the macros. You can change the Makefile run rule, to change the `OMP_NUM_THREADS` variable. How do execution times between x86 and ARM differ?

5. Bilateral Filter in FPGA Hardware

This is the main step of Lab5 in which you will design a Bilateral filter hardware accelerator to reduce the execution time with respect to the software ARM implementation. You will use the *lab5-hardware* zip file for this step.

You will work on the hardware accelerator by using the Vivado HLS toolset. File *filterHLS.c* contains the Bilateral filter code which will be synthesized into hardware. You need to use the Vivado HLS tool flow to implement the Bilateral filter (function *bilateralFilterKernel*) in hardware. As a first step, you only need to synthesize a baseline hardware implementation as-is without any optimization directives. Note that Vivado HLS ignores the OpenMP `#pragmas` and, therefore, you do not need to remove them during synthesis. You have to write your own source testbench to instantiate the accelerator and to test the code. Do not use the code in *filterHost.c* as the Vivado HLS testbench.

Note also the granularity of acceleration execution: each invocation of the accelerator will apply the Bilateral filter to the whole image.

Every time you produce a new (and hopefully faster!) hardware accelerator (i.e. you have a new *filterHLS.c* file), you will use the OpenCL-based source code for the host unit (*filterHost.c*) to run the code in the FPGA. The host unit code which is based on the OpenCL programming model is executed on the ARM CPU. Even if the hardware will do most of the heavy-duty computation, the ARM core will still execute all tasks related to initialization, the invocation of and communication with the hardware accelerator, and validation of the results produced by the accelerator (see Appendix A). The OpenCL API will read the image from a file to an array, will perform configuration and triggering of the hardware accelerator, and will block until the hardware accelerator finishes execution. Then, the ARM core will read the output image back from the accelerator and will write it in an output file. Finally, the software running on ARM core checks the correctness of the accelerator output. The host unit code compares the output of the hardware accelerator to the *output.bin* file which is the correct (golden) output. The Mean Square Error (MSE) between the two frames is computed in the *compare()* function and printed out.

The OpenCL API exposes a lot of details of the interfaces between the ARM and the hardware accelerator and is interesting to read and understand (see Appendix A). It is very similar to any OpenCL host unit code that you could run for a GPU accelerator. In fact, OpenCL code can run without any changes in CPUs, GPUs, as well as FPGAs.

5.1. Precise Hardware Optimizations

As we have mentioned in class and the Vivado HLS tutorials, hardware design gives you a lot of potential to design different hardware accelerators that span the performance vs. area space. You should detect the performance bottleneck of the first hardware solution and apply a series of optimizations to improve performance (i.e. to reduce latency). Some ideas for optimizations are the following (note that this is only a partial list):

- a) Apply HLS directives, such as loop unrolling and software pipeline to increase performance (even at the expense of extra area).
- b) Constructs such as if-then-else tend to limit instruction-level parallelism, especially in software pipelined loops. If possible, it is sometimes better to eliminate such constructs. For example, in image processing, there is always the problem of how to treat pixels that are adjacent to the edges of the image. You may want to think about how can you avoid checking for the boundaries of the image.
- c) Most importantly: compute how many times your first hardware solution accesses each pixel of the input image to produce the output image. Redundant memory accesses are a major bottleneck in performance degradation, especially when there is no cache hierarchy to reduce latency (the hardware accelerator connects to the main memory without the benefit of a cache). Bandwidth improvement is critical especially if you have already improved parallelism. There is no point in parallelizing computation if you cannot read and write pixels fast enough to always keep the accelerator busy.
- d) Increase the clock frequency of the FPGA fabric. The default for this lab is 100 MHz.

5.2. Approximate Hardware Optimizations

Not all computations and not all data of an application are equally critical, requiring to be performed or maintained at 100% accuracy or correctness. Several application domains offer the opportunity to trade-off the quality of output (QoO) for significant improvements in performance. For such applications, it may be possible to only approximate the final output (or part of it), rather than computing the exact output result. Such applications include visualization, video/imaging, machine learning, etc. All optimizations you applied in section 5.1 are precise since they do not modify the correctness of the output. In other words, the output image produced by the hardware accelerators of section 5.1 should have an MSE which is very close to 0 wrt. to the *output.bin* image.

Bilateral filtering is a potential target for approximate computing. Your objective is to further increase performance by identifying computations whose accuracy can be relaxed. For example, FP operations that can be executed with data type smaller than 32-bit float. You need to be creative and think out of the box for this part of the Lab. Moreover, to speed up your exploration, you can quickly execute your approximate code in an x86 CPU to detect if the new MSE becomes unacceptably large.

When you have concluded the series of optimizations (precise and approximate), you need to record your performance and compare all hardware and software implementations. The comparison should include both performance and area overhead.

Appendix A. Introduction to Xilinx Vitis Application Development Flow

The Vitis application acceleration development flow provides a framework for developing and delivering FPGA accelerated applications using standard programming languages for both software and hardware components. The software component, or host program, is developed using C/C++ to run on x86 or embedded processors (like ARM), with OpenCL API calls to manage runtime interactions with the accelerator. The hardware component, or kernel, can be developed using C/C++, OpenCL C, or RTL (Verilog/VHDL). The Vitis software platform accommodates various methodologies, letting you start by developing either the application or the kernel.

We will focus on the details on using this framework for embedded systems, and especially for our target device, the ZCU102 UltraScale+ board.

Execution Model

In the Vitis core development kit, an application program is split between a host application and hardware-accelerated kernels with a communication channel between them. The host program, written in C/C++ and using API abstractions like OpenCL, runs on a host processor (such as an x86 for server or an ARM processor for embedded platforms), while hardware-accelerated kernels run within the programmable logic (PL) region of a Xilinx device.

The API calls, managed by Xilinx Runtime (XRT), are used to process transactions between the host program and the hardware accelerators. Communication between the host and the kernel, including control and data transfers, occurs across the AXI bus for embedded platforms. While control information is transferred between specific memory locations in the hardware, global memory is used to transfer data between the host program and the kernels. Global memory is external to the FPGA device and is implemented using DDR4 technology. On the other hand, BRAM is internal to the FPGA device. In previous labs, we may have referred to the global memory as main memory or host memory.

For instance, in a typical application, the host (i.e. the ARM CPU) first initializes/prepares data to be operated on by the kernel and, then, stores them into global memory. The kernel subsequently operates on the data, storing results back to the global memory. Upon kernel completion, the host reads the data and continues with other tasks (which may include calling the accelerator more times).

The target platform contains the FPGA accelerated kernels, global memory, and direct memory access (DMA) for memory transfers. Kernels can have one or more global memory interfaces and are programmable. The Vitis core development kit execution model can be broken down into the following steps:

1. The host program prepares the input data needed by a kernel and stores them into the global memory.
2. The host program sets up the kernel with its input parameters.
3. The host program triggers the execution of the kernel function on the FPGA.
4. The kernel performs the required computation while reading data from global memory, as necessary.
5. The kernel writes data back to global memory and notifies the host unit that it has completed its task.
6. The host program uses the data and continues operations with other tasks.

Build Process

The Vitis core development kit offers all of the features of a standard software development environment:

- Compiler or cross-compiler for host applications running on x86 or ARM processors.
- Cross-compilers for building the FPGA binary.
- Debugging environment to help identify and resolve issues in the code.
- Performance profilers to identify bottlenecks and help you optimize the application.

The build process follows a standard compilation and linking process for both the host program and the kernel code. As shown in the following figure, the host program is built using the GNU C++ compiler (g++) or the GNU C++ ARM cross-compiler for MPSoC-based devices, like the ZCU102 UltraScale+ board. The FPGA binary is built using the Vitis compiler (v++).

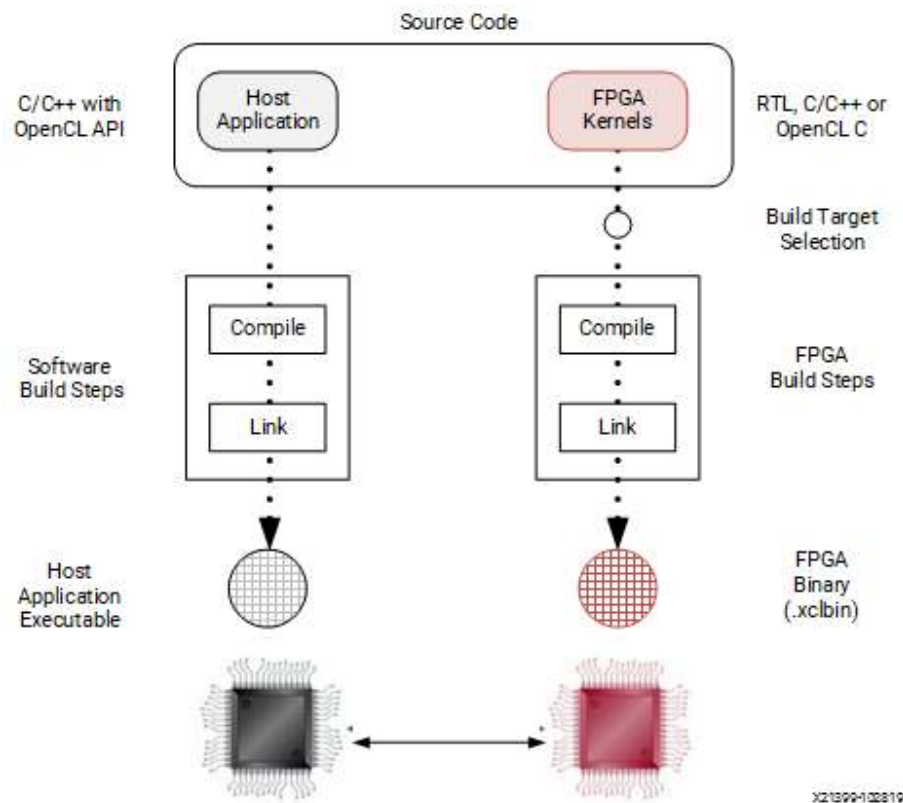


Figure 4. Software/Hardware Build Process

Host Program Build Process. The main application is compiled and linked with the `g++` compiler, using the following two-step process:

1. Compile any required source code into object files (.o).
2. Link the object files (.o) with the XRT shared library to create the executable.

FPGA Binary Build Process. Kernels can be described in C/C++, or OpenCL C code, or can be created from packaged RTL designs. Each hardware kernel is independently compiled to a Xilinx object (.xo) file. Xilinx object (.xo) files are, then, linked with the hardware platform to create an FPGA binary file (.xclbin) that is loaded into the Xilinx device on the target platform. The .xclbin file contains the bitstream as we have seen in previous labs. The key to building the FPGA binary is to determine the build target you are producing.

Build Targets. The Vitis compiler build process generates the host program executable and the FPGA binary (.xclbin). The nature of the FPGA binary is determined by the build target.

- When the build target is software or hardware emulation, the Vitis compiler generates simulation models of the kernels in the FPGA binary. These emulation targets let you build,

- run, and iterate the design over relatively quick cycles; debugging the application and evaluating performance.
- When the build target is the hardware system, Vitis compiler generates the .xclbin for the hardware accelerator, using the Vivado Design Suite to run synthesis and implementation. It uses these tools with predefined settings proven to provide good quality of results. Using the Vitis core development kit does not require knowledge of these tools; however, hardware-savvy developers can fully leverage these tools and use all the available features to implement kernels.

The Vitis compiler provides three different build targets, two emulation targets used for debugging and validation purposes, and the default hardware target used to generate the actual FPGA binary:

- Software Emulation (sw_emu): Both the host application code and the kernel code are compiled to run on the host processor. This allows iterative algorithm refinement through fast build-and-run loops. This target is useful for identifying syntax errors, performing source-level debugging of the kernel code running together with the application, and verifying the behavior of the system.
- Hardware Emulation (hw_emu): The kernel code is compiled into a hardware model (RTL), which is run in a dedicated simulator. This build-and-run loop takes much longer but provides a detailed, cycle-accurate view of kernel activity. This target is useful for testing the functionality of the logic that will go into the FPGA and getting initial performance estimates. Please be aware that, this emulation needs a significant time to complete, especially for large data sets. For that reason, it will not be used in this lab.
- System (hw): The kernel code is compiled into a hardware model (RTL) and then implemented on the FPGA, resulting in a binary that will run on the actual FPGA.

C/C++ Kernels

In the Vitis core development kit, the kernel code is generally a compute-intensive part of the algorithm and meant to be accelerated on the FPGA. During the runtime, the C/C++ kernel executable is called through the host code executable.

IMPORTANT: Because the host code and the kernel code are developed and compiled independently, there could be a name mangling issue if one is written in C and the other in C++. To avoid this issue, wrap the kernel function declaration with the extern "C" linkage in the header file, or wrap the whole function in the kernel code.

```
extern "C" {
    void kernel_function(int *in, int *out, int size){
        // Code goes here
    }
}
```

Figure 5. HLS kernel functions are required to be wrapped with extern “C”.

Interfaces. Two types of data transfer occur between global memory and the kernels on the FPGA. Data are transferred between the host CPU and the accelerator through global memory banks. Scalar data is passed directly from the host to the kernel.

Memory-Mapped Interfaces. The main data processed by the kernel, often in a large volume, should be transferred through the global memory banks on the FPGA board. The host machine transfers a large chunk of data to one or more global memory bank(s). The kernel accesses the data from those global memory banks, preferably in burst mode. After the kernel finishes the computation, the resulting data is transferred back to the host machine through the global memory banks.

When writing the kernel interface description, pragmas are used to denote the interfaces coming to and from the global memory banks.

```
void cnn( int *pixel, // Input pixel
         int *weights, // Input Weight Matrix
         int *out, // Output pixel
         ... // Other input or Output ports )

#pragma HLS INTERFACE m_axi port=pixel offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=pixel bundle=control

#pragma HLS INTERFACE m_axi port=weights offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=weights bundle=control

#pragma HLS INTERFACE m_axi port=out offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=out bundle=control
```

Figure 6. Necessary pragmas for data pointer arguments.

In the example in Figure 6, there are three data pointer interfaces. The two inputs are *pixel* and *weights* and the output is *out*. These inputs and outputs connected to the global memory bank are specified in C code by using *HLS INTERFACE m_axi* pragmas as shown above. Also, each memory-mapped interface port needs to have a second interface pragma, declared as a scalar input through *s_axilite*, using the same port name and *bundle=control*.

The bundle keyword on the pragma HLS interface defines the name of the port. The system compiler will create a port for each unique bundle name, resulting in a compiled kernel object (.xo)

that has a single AXI interface, *m_axi_gmem*. When the same bundle name is used for different interfaces, this results in these interfaces being mapped to the same port. In this lab, you will use only one bundle.

Scalar Inputs. Scalar inputs are typically control variables that are directly loaded from the host machine. They can be thought of as programming data or parameters under which the main kernel computation takes place. These kernel inputs are write-only from the host side. These interfaces are specified in the kernel code as shown below:

```
void process_image(int *input, int *output, int width, int height) {  
    ...  
    #pragma HLS INTERFACE s_axilite port=width bundle=control  
    #pragma HLS INTERFACE s_axilite port=height bundle=control  
    #pragma HLS INTERFACE s_axilite port=return bundle=control  
    ...  
}
```

Figure 7. HLS pragmas for scalar arguments.

In Figure 7 two scalar inputs specify the image *width* and *height*. These inputs are specified using the *#pragma HLS INTERFACE s_axilite*. These data inputs come to the kernel directly from the host machine and are not using a global memory bank.

IMPORTANT: Currently, the Vitis core development kit supports only one control interface bundle for each kernel. Therefore, the *bundle=* name should be the same for all scalar data inputs, including the function return. In the preceding example, *bundle=control* is used for all scalar inputs. The above requirements are absolutely necessary for the Vitis compiler. Every other aspect of kernel development and optimization is the same as the Vivado HLS tutorials.

Host Application

In the Vitis core development kit, the host code is written in C or C++ language using the industry-standard OpenCL API. The Vitis core development kit provides an OpenCL 1.2 embedded profile conformant runtime API. In general, the structure of the host code can be divided into three sections:

1. Setting up the environment.
2. Core command execution including executing one or more kernels.
3. Post-processing and release of resources.

Setting Up the OpenCL Environment. The host code in the *Vitis* core development kit follows the OpenCL programming paradigm. To set the environment properly, the host application needs to initialize the standard OpenCL structures: *target platform*, *devices*, *context*, *command queue*, and *program*.

Platform. The OpenCL API call [clGetPlatformIDs](#) is used to discover the set of available OpenCL platforms for a given system. Thereafter, [clGetPlatformInfo](#) is used to identify Xilinx device-based platforms by matching *cl_platform_vendor* with the string “Xilinx”. Though it is not explicitly shown in the following code examples, it is always a good coding practice to use error checking after each of the OpenCL API calls. This can help to debug and improve productivity when you are debugging the host and kernel code in the emulation flow, or during hardware execution.

```
cl_platform_id platform_id;           // platform id

err = clGetPlatformIDs(16, platforms, &platform_count);

// Find Xilinx Platform
for (unsigned int iplat=0; iplat<platform_count; iplat++) {
    err = clGetPlatformInfo(platforms[iplat],
        CL_PLATFORM_VENDOR,
        1000,
        (void *)cl_platform_vendor,
        NULL);

    if (strcmp(cl_platform_vendor, "Xilinx") == 0) {
        // Xilinx Platform found
        platform_id = platforms[iplat];
    }
}
```

Figure 8. Platform initialization.

Devices. After a Xilinx platform is found, the host code needs to identify the corresponding Xilinx devices. The following code demonstrates finding all the Xilinx devices, with an upper limit of 16, using the API [clGetDeviceIDs](#).

```

cl_device_id devices[16]; // compute device id
char cl_device_name[1001];

err = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_ACCELERATOR,
    16, devices, &num_devices);

printf("INFO: Found %d devices\n", num_devices);

//iterate all devices to select the target device.
for (uint i=0; i<num_devices; i++) {
    err = clGetDeviceInfo(devices[i], CL_DEVICE_NAME, 1024, cl_device_name, 0);
    printf("CL_DEVICE_NAME %s\n", cl_device_name);
}

```

Figure 9. Device initialization.

Context. The [clCreateContext](#) API is used to create a context that contains a Xilinx device that will communicate with the host machine.

```
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
```

In the code example, the *clCreateContext* API is used to create a context that contains one Xilinx device. Xilinx recommends creating only one context per device or sub-device.

Command Queues. The [clCreateCommandQueue](#) API creates one or more command queues for each device. The FPGA can contain multiple kernels, which can be either the same or different kernels. When developing the host application, there are two main programming approaches to execute kernels on a device:

1. Single out-of-order command queue: Multiple kernel executions can be requested through the same command queue. XRT dispatches kernels as soon as possible, in any order, allowing concurrent kernel execution on the FPGA.
2. Multiple in-order command queue: Each kernel execution will be requested from different in-order command queues. In such cases, XRT dispatches kernels from the different command queues, improving performance by running them concurrently on the device.

The following is an example of standard API calls to create in-order and out-of-order command queues.

```
// Out-of-order Command queue
commands = clCreateCommandQueue(context, device_id,
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &err);

// In-order Command Queue
commands = clCreateCommandQueue(context, device_id, 0, &err);
```

Program. As described in Build Process, the host and kernel code are compiled separately to create separate executable files: the host program executable and the FPGA binary (.xclbin). When the host application runs, it must load the .xclbin file using the [clCreateProgramWithBinary](#) API. Figure 10 example code shows how the standard OpenCL API is used to build the program from the .xclbin file.


```

unsigned char *kernelbinary;
char *xclbin = argv[1];

printf("INFO: loading xclbin %s\n", xclbin);

int size=load_file_to_memory(xclbin, (char **) &kernelbinary);
size_t size_var = size;

cl_program program = clCreateProgramWithBinary(context, 1, &device_id,
                                                &size_var, (const unsigned char **)
&kernelbinary,
                                                &status, &err);

// Function
int load_file_to_memory(const char *filename, char **result)
{
    uint size = 0;
    FILE *f = fopen(filename, "rb");
    if (f == NULL) {
        *result = NULL;
        return -1; // -1 means file opening fail
    }
    fseek(f, 0, SEEK_END);
    size = ftell(f);
    fseek(f, 0, SEEK_SET);
    *result = (char *)malloc(size+1);
    if (size != fread(*result, sizeof(char), size, f)) {
        free(*result);
        return -2; // -2 means file reading fail
    }
    fclose(f);
    (*result)[size] = 0;
    return size;
}

```

Figure 10. Build a program from .xclbin file, produced by Vitis compiler.

The example performs the following steps:

1. The kernel binary file, .xclbin, is passed in from the command line argument, **argv[1]**.

TIP: Passing the .xclbin through a command line argument is one approach. You can also hardcode the kernel binary file in the host program, define it with an environment variable, read it from a custom initialization file, or another suitable mechanism.
2. The **load_file_to_memory** function is used to load the file contents in the host machine memory space.

3. The `clCreateProgramWithBinary` API is used to complete the program creation process in the specified context and device.

Executing Commands in the FPGA. Once the OpenCL environment is initialized, the host application is ready to issue commands to the device and interact with the kernels. These commands include:

1. Setting up the kernels.
2. Buffer transfer to/from the FPGA.
3. Kernel execution on FPGA.
4. Event synchronization.

Setting Up Kernels. After setting up the OpenCL environment, such as identifying devices, creating the context, command queue, and program, the host application should identify the kernels that will execute on the device, and set up the kernel arguments.

The OpenCL API `clCreateKernel` should be used to access the kernels contained within the .xclbin file (the "program"). The `cl_kernel` object identifies a kernel in the program loaded into the FPGA that can be run by the host application. The following code example identifies two kernels defined in the loaded program.

```
kernel1 = clCreateKernel(program, "<kernel_name_1>", &err);
kernel2 = clCreateKernel(program, "<kernel_name_2>", &err); // etc
```

Figure 11 Create a kernel handler from the program.

Creating buffers. Any non-scalar data type must be passed to kernels buffer objects. You must first create a buffer object using `clCreateBuffer`. It returns a `cl_mem` objects, which can be used as a kernel argument. For this lab, we will be using different `clCreateBuffer` arguments, alongside with `clEnqueueMapBuffer`. An example is shown in Figure 12.

```
// Create memory buffers
cl_mem dev_buf1 = clCreateBuffer(context, CL_MEM_WRITE_ONLY, size, &host_mem_ptr1,
NULL);
cl_mem dev_buf2 = clCreateBuffer(context, CL_MEM_READ_ONLY, size, &host_mem_ptr2,
NULL);
```

Figure 12. Create simple buffers.

Setting Kernel Arguments. In the Vitis software platform, two types of arguments can be set for `cl_kernel` objects:

1. Scalar arguments are used for small data transfers, such as constant or configuration type data. These are write-only arguments from the host application perspective, meaning they are inputs to the kernel.
2. Memory buffer arguments are used for large data transfer. The value is a pointer to a memory object created with the context associated with the program and kernel objects. These can be inputs to or outputs from the kernel.

Kernel arguments can be set using the [clSetKernelArg](#) command as shown below. The following example shows setting kernel arguments for two scalar and two buffer arguments.

```
int err = 0;
// Setup scalar arguments
cl_uint scalar_arg_image_width = 3840;
err |= clSetKernelArg(kernel, 0, sizeof(cl_uint), &scalar_arg_image_width);
cl_uint scalar_arg_image_height = 2160;
err |= clSetKernelArg(kernel, 1, sizeof(cl_uint), &scalar_arg_image_height);

// Setup buffer arguments
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &dev_buf1);
err |= clSetKernelArg(kernel, 3, sizeof(cl_mem), &dev_buf2);
```

Figure 13. Creating two buffer variables, and setting kernel arguments.

IMPORTANT: Although OpenCL allows setting kernel arguments any time before enqueueing the kernel, you should set kernel arguments as early as possible. XRT will error out if you try to migrate a buffer before XRT knows where to put it on the device. Therefore, set the kernel arguments before performing any enqueue operation (for example, [clEnqueueMigrateMemObjects](#)) on any buffer.

Buffer Transfer to/from the FPGA Device. Interactions between the host program and hardware kernels rely on transferring data to and from the global memory in the device. The method to send data back and forth from the FPGA is using [clCreateBuffer](#), [clEnqueueWriteBuffer](#), and [clEnqueueReadBuffer](#) commands.

Note: Xilinx recommends using [clEnqueueMigrateMemObjects](#) instead of [clEnqueueReadBuffer](#) and [clEnqueueWriteBuffer](#).

A suitable approach for creating and managing buffers in embedded systems, like ZedBoard or UltraScale+ board, is to use [clEnqueueMapBuffer](#). The [clEnqueueMapBuffer](#) API maps the specified buffer and returns a pointer created by XRT to this mapped region. Then, it fills the host side pointer with your data, followed by [clEnqueueMigrateMemObject](#) to transfer the data to and from the device. The following code example uses this style, and that is the preferred way to use it in this lab.

```

// Two cl_mem buffer, for read and write by kernel
cl_mem dev_mem_read_ptr = clCreateBuffer(context,
                                         CL_MEM_READ_ONLY,
                                         sizeof(int) * number_of_words, NULL, NULL);

cl_mem dev_mem_write_ptr = clCreateBuffer(context,
                                           CL_MEM_WRITE_ONLY,
                                           sizeof(int) * number_of_words, NULL, NULL);

// Setting arguments
clSetKernelArg(kernel, 0, sizeof(cl_mem), &dev_mem_read_ptr);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &dev_mem_write_ptr);

// Get Host side pointer of the cl_mem buffer object
auto host_write_ptr =
clEnqueueMapBuffer(queue, dev_mem_read_ptr, true, CL_MAP_WRITE, 0, bytes, 0, nullptr, nullptr,
                  &err);
auto host_read_ptr =
clEnqueueMapBuffer(queue, dev_mem_write_ptr, true, CL_MAP_READ, 0, bytes, 0, nullptr, nullptr,
                  &err);

// Fill up the host_write_ptr to send the data to the FPGA
for(int i=0; i< MAX; i++) {
    host_write_ptr[i] = <.... >
}

// Migrate
cl_mem mems[2] = {host_write_ptr, host_read_ptr};
clEnqueueMigrateMemObjects(queue, 2, mems, 0, 0, nullptr, &migrate_event));

// Schedule the kernel
clEnqueueTask(queue, kernel, 1, &migrate_event, &enqueue_event);

// Migrate data back to host
clEnqueueMigrateMemObjects(queue, 1, &dev_mem_write_ptr,
                          CL_MIGRATE_MEM_OBJECT_HOST, 1, &enqueue_event,
                          &data_read_event);

clWaitForEvents(1, &data_read_event);

// Now use the data from the host_read_ptr

```

Figure 14. A complete example of buffer creation, setting arguments, transferring data, and executing.

Kernel Execution. Often the compute-intensive task required by the host application can be defined inside a single kernel, and the kernel is executed only once to work on the entire data range.

Because there is an overhead associated with multiple kernel executions, invoking a single monolithic kernel can improve performance. Though the kernel is executed only once and works on the entire range of the data, the parallelism (and thereby acceleration) is achieved on the FPGA inside the kernel hardware. If properly coded, the kernel is capable of achieving parallelism by various techniques such as instruction-level parallelism (loop pipeline) and function-level parallelism (dataflow).

When the kernel is compiled to a single hardware instance (or CU) on the FPGA, the simplest method of executing the kernel is using *clEnqueueTask* as shown below.

```
err = clEnqueueTask(commands, kernel, 0, NULL, NULL);
```

XRT schedules the workload, or the data passed through OpenCL buffers from the kernel arguments, and schedules the kernel tasks to run on the accelerator on the Xilinx FPGA.

Post-Processing and FPGA Cleanup. At the end of the host code, all the allocated resources should be released by using proper release functions.

```
clReleaseCommandQueue(Command_Queue);  
clReleaseContext(Context);  
clReleaseDevice(Target_Device_ID);  
clReleaseKernel(Kernel);  
clReleaseProgram(Program);  
free(Platform_IDs);  
free(Device_IDs);
```

Summary. As discussed in earlier topics, the recommended coding style for the host program in the Vitis core development kit includes the following points:

1. Add error checking after each OpenCL API call for debugging purposes, if required.
2. In the Vitis core development kit, one or more kernels are separately compiled/linked to build the *.xclbin* (bitstream) file. The API *clCreateProgramWithBinary* is used to build the *cl_program* object from the kernel binary.
3. Use buffer for setting the kernel argument (*clSetKernelArg*) before any enqueue operation on the buffer.
4. Transfer data between the host unit and the kernel by using *clEnqueueMigrateMemObjects* or *clEnqueueMapBuffer*.

5. Preferably use the out-of-order command queue for concurrent command execution on the FPGA.
6. Invoke the accelerator with `clEnqueueTask`.
7. Use event synchronization commands, `clFinish`, and `clWaitForEvents`, to resolve dependencies of the asynchronous OpenCL API calls.
8. Release all OpenCL allocated resources when finished.

Vitis Compiler Configuration File

A configuration file can be used to specify the Vitis compiler options. A configuration file provides an organized way of passing options to the compiler by grouping similar flags together and minimizing the length of the `v++` command line. Some of the features that can be controlled through configuration file entries include:

- HLS options to configure kernel compilation.
- Connectivity directives for system linking such as the number of kernels to instantiate or the assignment of kernel ports to global memory.
- Directives for the Vivado Design Suite to manage hardware synthesis and implementation.

In general, any `v++` command option can be specified in a configuration file. However, the configuration file supports defining sections containing groups of related commands to help manage build options and strategies. For more information, you may visit https://www.xilinx.com/html_docs/xilinx2019_2/vitis_doc/Chunk1193338764.html#ariaid-title8.

For this lab, we provide you with a basic configuration file `design.cfg`, that contains the basic configuration that is needed to run on our infrastructure. It setups the connectivity of the kernel and the clock frequency that will be used for your design. By default, it is clocked at 100 MHz.

For more details on the material of Appendix A, you may consult [Xilinx UG1393: Vitis Unified Software Platform Documentation: Application Acceleration Development](#)

Appendix B. Introduction to Git & GitLab

Version Control

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. Ideally, we can place any file in the computer on version control.

A Version Control System (VCS) allows you to revert files to a previous state, revert the entire project to a previous state, review changes made over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also means that if something goes wrong or some files are lost, you can generally recover a previous version of the files.

What is Git?

Git is a version-control system for tracking changes in computer files and coordinating work on those files among multiple people. Git is a Distributed Version Control System. Git does not necessarily rely on a central server to store all the versions of a project's files. Instead, every user "clones" a copy of a repository (a collection of files) and has the full history of the project on their hard drive. This clone has all of the metadata of the original while the original itself is stored on a self-hosted server or a third-party hosting service like *GitLab*. In this lab, we will be using our own private GitLab instance.

Git helps you to keep track of the changes you make to your code. It is the *history log* for your code. If at any point while coding you encounter a fatal error and don't know what is causing it, you can always revert back to a stable state. So it is very helpful for debugging. Or you can simply see what modifications you made to your code over time.

```
list1 = [1,2,3]
list1.append(4)
print(list1)
```

#initialFile

```
list1 = [1,2,3]
list1.append(4)
list1.append(5)
print(list1)
```

#addedALine

```
list1 = [1,2,3]
list1.pop()
print(list1)
```

#makingChanges

In the example above, all three cards represent different versions of the same file. We can select which version of the file we want to use at any point in time. So I can jump to and from to any version of the file in the git repository.

Git also helps you synchronize code between multiple people. So imagine you and your friend are collaborating on a project and you are both working on the same project files. Now Git takes those changes you and your friend made independently and merges them to a single “master” repository. By using Git you can ensure you both are working on the most recent version of the repository. So you don’t have to worry about mailing your files to each other and working with a large number of copies of the original file.

What is a Repository?

A repository (repo) is a collection of source code files.

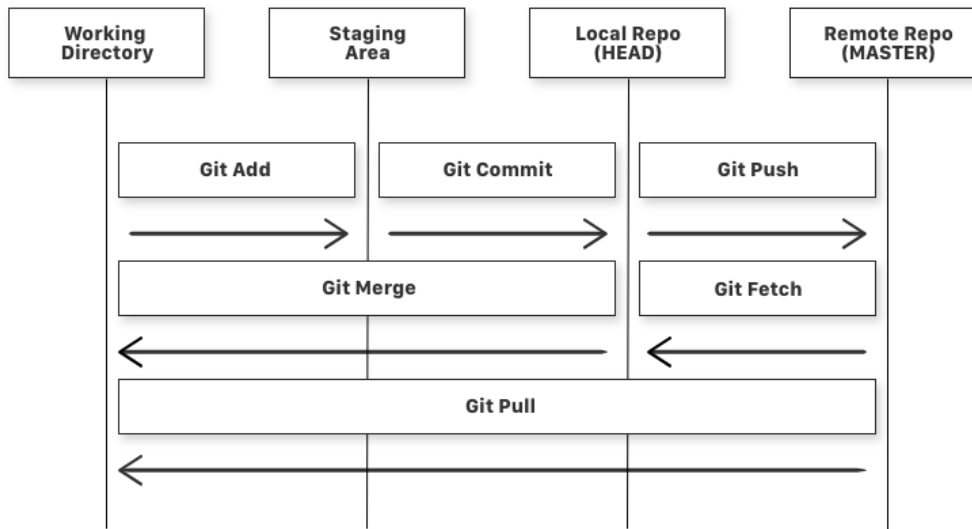
What is a Branch?

A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process. You can think of them as a way to request a brand new working directory, staging area, and project history. New commits are recorded in the history of the current branch, which results in a fork in the history of the project. The default branch for a repository is called *master*.

Git Workflow

There are four fundamental elements in the Git Workflow.

- Working Directory
- Staging Area
- Local Repository
- Remote Repository



If you consider a file in your Working Directory, it can be in three possible states.

- It can be **staged**. This means the files with the updated changes are marked to be committed to the local repository but not yet committed.
- It can be **modified**. This means the files with the updated changes are not yet stored in the local repository.
- It can be **committed**. This means that the changes you made to your file are safely stored in the local repository.

Git commands

- **git add** is a command used to add a file that is in the working directory to the staging area.
- **git commit** is a command used to add all files that are staged to the local repository.
- **git push** is a command used to upload all committed files in the local repository to the remote repository. So in the remote repository, all files and changes will be visible to anyone with access to the remote repository.
- **git fetch** is a command used to get files from the remote repository to the local repository but not into the working directory.

- **git merge** is a command used to get the files from the local repository into the working directory.
- **git pull** is the command used to get files from the remote repository directly into the working directory. It is equivalent to a git fetch and a git merge.
- **git branch** command lets you create, list, rename and delete branches.
- **git checkout <branch_name>** command used to change the branch that you are working on.
- **git checkout <branch_name|commit_id> <path/to/file>** command lets you to retrieve a specific file version from a specific branch or commit.

Installation:

For Linux(deb), enter the following in the terminal:

```
$ sudo apt install git-all
```

For Windows, follow the link:

<https://gitforwindows.org/>

Initial setup

Enter your git username and email address, since every Git commit will use this information to identify you as the author.

```
$ git config --global user.name "YOUR_USERNAME"
$ git config --global user.email "example@uth.gr"
$ git config --global --list # To check the info you just provided
```

Initialize Git:

And to place it under git, enter:

```
$ touch README.md # To create a README file for the repository
```

```
$ git init # Initiates an empty git repository
```

Add files to the Staging Area for commit:

Now to add the files to the git repository for commit:

```
$ git add . # Adds all the files in the local repository and stages them for commit  
$ git add README.md # To add a specific file
```

Before commit check what files are staged:

```
$ git status # Lists all new or modified files to be committed
```

Commit Changes you made to your Git Repo:

Now to commit files you added to your git repo:

```
$ git commit -m "First commit" # The message in the " " is given so that the other users can read  
the message and see what changes you made
```

Uncommit Changes you just made to your Git Repo:

Now suppose you just made some error in your code or placed an unwanted file inside the repository, you can unstage the files you just added using:

```
$ git reset HEAD~1 # Remove the most recent commit
```

Add a remote origin and push:

Now each time you make changes in your files and save it, it won't be automatically updated on GitLab. All the changes we made in the file are updated in the local repository. Now to update the changes to the master:

```
$ git remote add origin remote_repository_URL# sets the new remote
```

The **git remote -v** command lists the URLs of the remote connections you have to other repositories.

```
$ git remote -v# List the remote connections you have to other repositories.
```

Now the **git push** command pushes the changes in your local repository up to the remote repository you specified as the origin.

```
$ git push -u origin master # pushes changes to origin
```

See the changes you made to a file:

Once you start making changes on your files and you save them, the file won't match the last version that was committed to git. To see the changes you just made:

```
$ git diff # To show the files changes not yet staged
```

Revert back to the last committed version to the Git Repo:

Now you can choose to revert back to the last committed version by entering:

```
$ git checkout .  
OR for a specific file  
$ git checkout -- <filename>
```

View Commit History:

You can use the **git log** command to see the history of commit you made to your files:

```
$ git log
```

Cloning a Git Repo:

Locate to the directory you want to clone the repo. Copy the link of the repository you want and enter the following:

```
$ git clone remote_repository_URL
```

So to make sure those changes are reflected on my local copy of the repo:

```
$ git pull origin master
```

`gitignore` tells git which files (or patterns) it should ignore. It's usually used to avoid committing transient files from your working directory that aren't useful to other collaborators, such as compilation products, temporary files IDEs create, etc.

For more information: <https://git-scm.com/doc>

GitLab

GitLab is a web-based tool that provides a Git-repository manager providing wiki, issue-tracking, and continuous integration/continuous deployment pipeline features, using an open-source license, developed by GitLab Inc. A popular commercial alternative is GitHub. For more general information you can visit: <https://about.gitlab.com/>

You will use a private GitLab instance for this course.

Appendix C. Instructions on how to run your designs

In this lab, you will use git versioning system and a private GitLab instance, that has an integrated CI/CD system that will automate the whole process of building, emulating, and running your designs on FPGAs. You will be able to commit any changes to your design, by using the Web IDE that is provided with GitLab, or by using the git command-line interface (CLI). However, you will be able to see the results of your runs on GitLab Web UI only.

Repository

Each team will work on their own repository, independently from each other and will contain 3 branches:

- *software*: It contains your ARM software implementation.
- *hardware_precise*: It contains your FPGA implementations with precise optimizations only.
- *software_precise*: It contains your FPGA implementations with precise and approximate optimizations.

The repository for each group will be already initialized with the necessary files for you to begin your work.

File structure for each branch

Branch software:

- Directory **lib**: Contains the *utils.mk* makefile that provides the necessary functions of cross-compiling the source code to ARM architectures (aarch32/aarch64). You must not make any alterations in this directory.
- File **.gitlab.ci.yml**: It is a script that is used by the GitLab CI/CD system. It contains all the steps and the commands that run after every commit you make on this branch. You should not make any changes to this file. If needed, please ask your instructor first.
- File **Makefile**: It is the main makefile for building and running the design on FPGA ARM CPU. It includes the makefile that resides in the **lib** directory. It can work only with a suitable set up on the system that will be run, which includes Vitis Core Development Kit and Petalinux, version 2019.2. You must not make any changes in this file, and you cannot run it on your system, without the appropriate Vitis setup.
- Files **input.bin**, **output.bin**: The data files that are used by the filter application. They are binary format.

- File **filter.c**: Your C source code.

Branches hardware_precise & hardware_approximate:

- Directory **lib**: Contains the *utils.mk* makefile that provides the necessary function of cross-compiling the source code to ARM architectures (aarch32/aarch64), and a **common** directory, that includes various utility files that help with the compilation process. You must not make any changes to this directory.
- File **.gitlab.ci.yml**: It is a script that is used by the GitLab CI/CD system. It contains all the steps and the commands that run after every commit you make on this branch. You should not make any changes to this file. If needed, please ask your instructor first.
- File **Makefile**: It is the main makefile for building and running the design on the ZCU102 UltraScale+. It includes the makefile that resides in the **lib** directory. It can work only with a suitable set up on the system that will be run, which includes Vitis Core Development Kit and Petalinux, version 2019.2. You must not make any changes in this file, and you cannot run it on your system, without the appropriate Vitis setup.
- Files **design.cfg**, **design_emu.cfg**: They are used as configuration files for the Vitis compiler. They can contain certain flags for the HLS synthesis stage, as well as for the Vitis compiler linking stage. For this lab, we will only need the connectivity flag and the HLS clock option. The file **design_emu.cfg** is being used only for emulation targets, and cannot contain any references to the clock options. You do not need to make any changes, except for the clock options. For more information please refer to Xilinx UG1393.
- Files **input.bin**, **output.bin**: The data files that are used by the filter application.
- File **filterHost.c**: The source code for the host application.
- File **filterHLS.c**: The source code for the hardware kernel function.


GitLab Web Interface

You can access the GitLab Web UI by visiting <http://10.96.12.110/>, while you are connected to UTH VPN.


Details on how to use UTH VPN: <https://it.uth.gr/services/eikoniko-idiotiko-diktyo-vpn>

The first page that you will see is the sign-in page:

ECE340 Embedded Systems



ECE340 Embedded Systems GitLab



Sign in

Username or email

Password

☐ Remember me

[Forgot your password?](#)

Sign in

Welcome to ECE340 Embedded Systems GitLab.

It can be used solely for the purposes of lab requirements, or/and any other activities that are related to ECE340 course. Do not use it for personal or commercial use.

For any questions or assistance, please contact the instructor and/or Mr. Patras Alexandros (patras@uth.gr).

[Home](#) [About](#) [News](#) [Contact](#)

Department of Electrical and Computer Engineering — University Of Thessaly

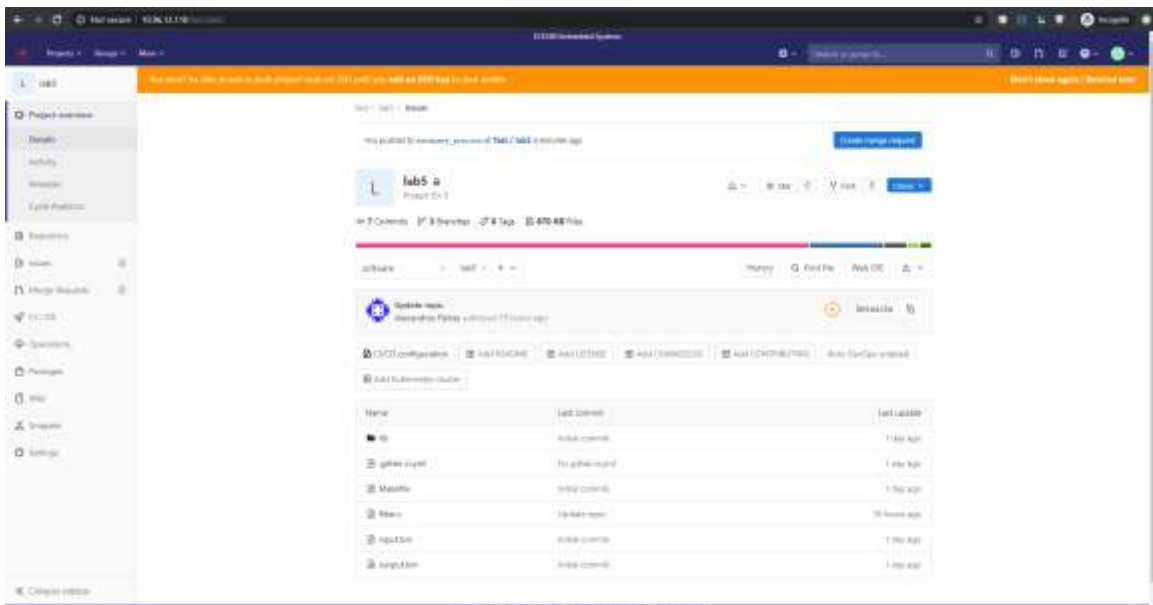
Your username and password have been provided to you with an email. If not, please contact the instructor or teaching assistants. When you sign in for the first time, the system will ask you to set your own password, which must be at least 8 characters long. Please keep in mind that the “forgot password” function is currently not working, so in case you have forgotten your password please contact the instructor or the teaching assistants.

The first page that you should see is Your Projects page, showing the Lab5 repository:



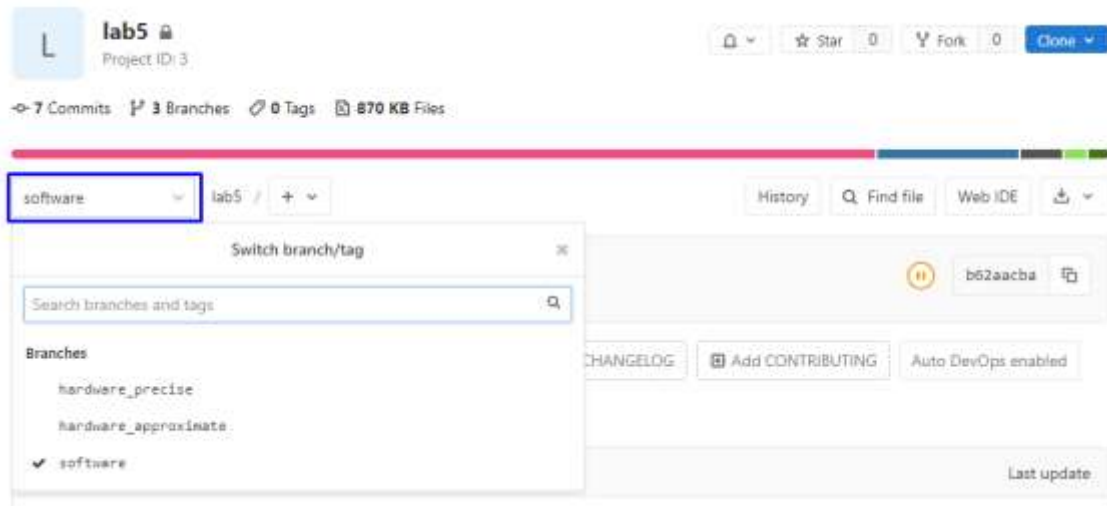
If not, you can see this page by clicking on **Projects** top left button, and then **Your Projects**.

If you click on the Lab5 project/repository, you will be redirected to the default repository page:



You can safely ignore the orange alert message, and click on **Don't show again**. You will be using HTTP credentials, and not SSH keys.

You can navigate between the 3 branches from here:



Make changes

You have two ways to make any changes to your code and commit it to its branch.

1. Web IDE
2. Work locally and commit to the repository via git command-line interface

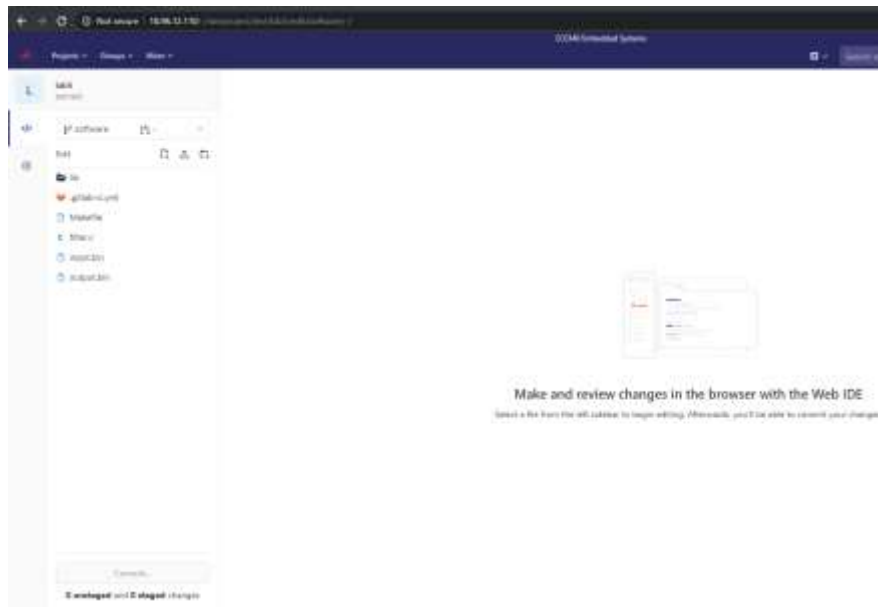
Web IDE

It is the easiest and most convenient way to work in this lab. After you have concluded with your optimizations that you explore locally – software only or hardware kernel using Vivado HLS – you can use the Web IDE to apply those changes to the proper file in any branch.

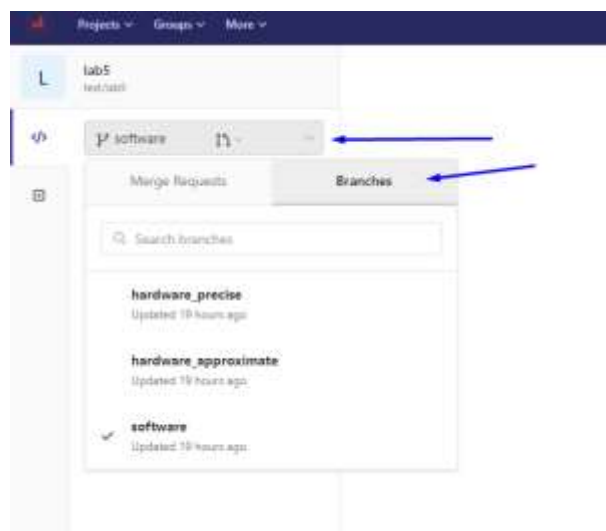
To access the Web IDE, you can click on the project default page on Web IDE:



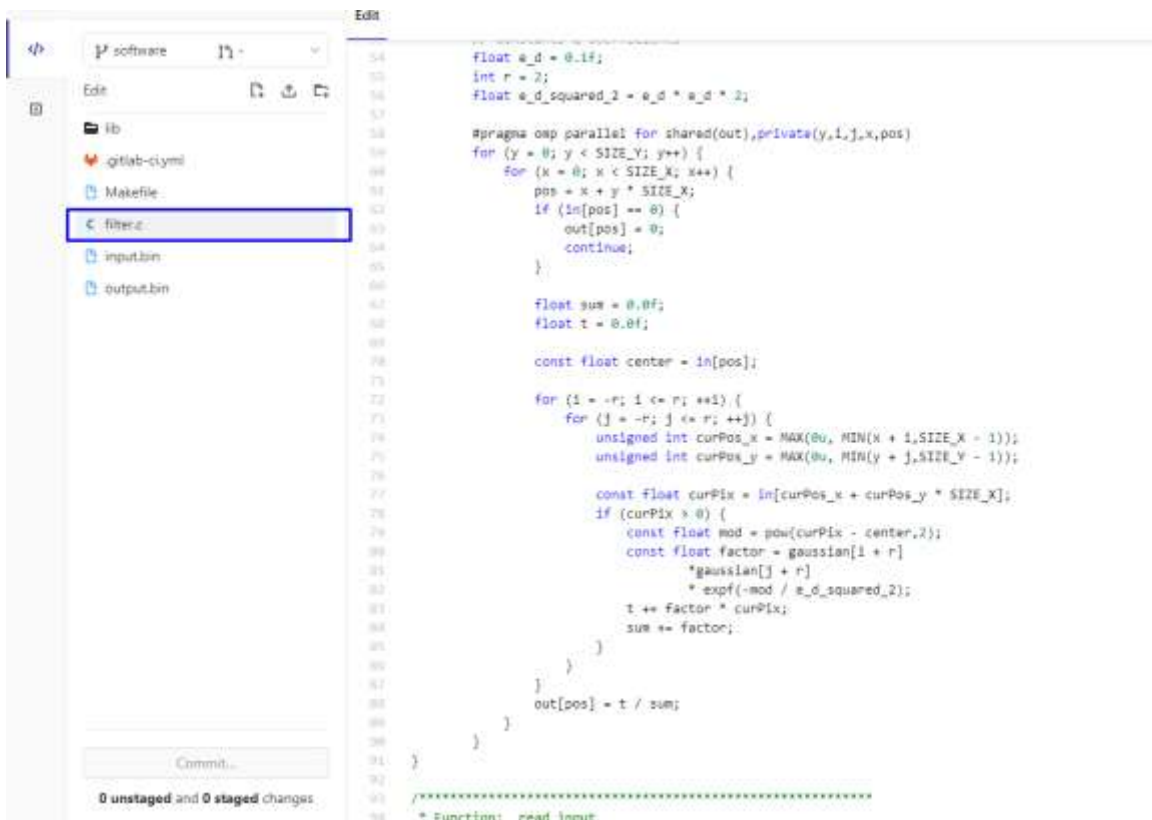
It will redirect you to the Web IDE interface.



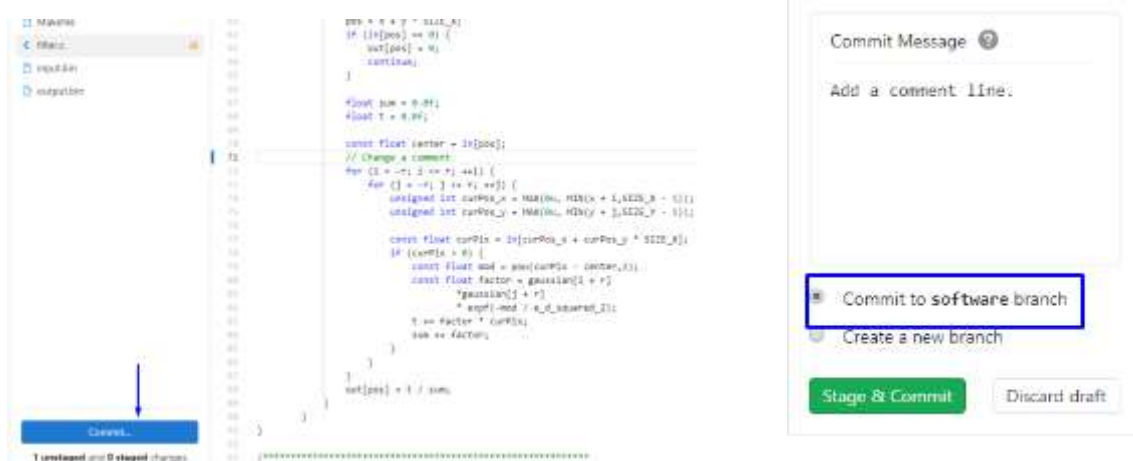
You can choose from the branch button, on which branch you would like to work on.



And then you can simply choose what file you need to edit. For example, let's edit the file *filter.c*.



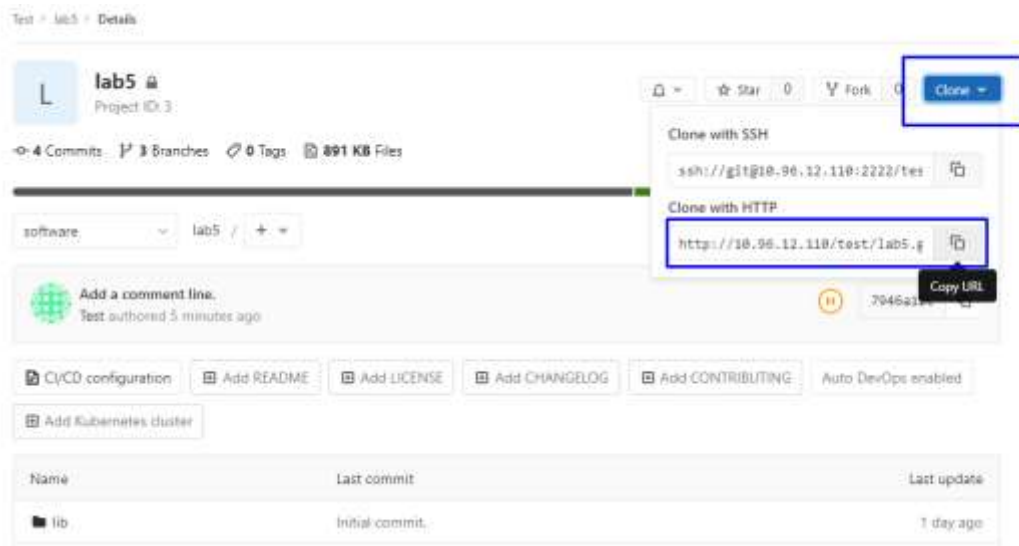
After you have made all your changes, you can simply click the *Commit* button, enter the commit message, and click on the option to *Commit to software branch*. Do not create a new branch. Then click *Stage & Commit*. Your changes will be committed to your branch.



Work locally using Git CLI

If you prefer to work locally, you can clone the repository to your computer, and make any changes there.

First, make a new local directory in which you will clone the remote repository. Then, visit the Gitlab web repository page and copy the HTTP clone URL:



Then, using your Git CLI interface use this commands to clone the repository:

\$ git clone <copied HTTP url>

You will be prompted to enter your username and your password. Enter them accordingly and you should see something similar to the following:

```
~/ece340$ git clone http://10.96.12.110/test/lab5.git
Cloning into 'lab5'...
Username for 'http://10.96.12.110': test
Password for 'http://test@10.96.12.110':
remote: Enumerating objects: 104, done.
remote: Counting objects: 100% (104/104), done.
remote: Compressing objects: 100% (100/100), done.
remote: Total 104 (delta 27), reused 0 (delta 0)
Receiving objects: 100% (104/104), 381.55 KiB | 4.60 MiB/s, done.
Resolving deltas: 100% (27/27), done.
~/ece340$ ls
lab5
~/ece340$ cd lab5
~/ece340/lab5 (software)$ ls
filter.c input.bin lib Makefile output.bin
~/ece340/lab5 (software)$
```

The process is similar if you use Windows.

The directory *lab5* corresponds to the repository directory and the default branch – here is branch *software*. You can now navigate between branches using the command:

\$ git checkout <branch_name>

And you will a message that informs you that you have changed branch and the directory contains the set of files that are in that branch.

```
~/ece340/lab5 (software)$ git checkout hardware_precise
Branch 'hardware_precise' set up to track remote branch 'hardware_precise' from 'origin'.
Switched to a new branch 'hardware_precise'
~/ece340/lab5 (hardware_precise)$ ls
design.cfg design_emu.cfg filterHLS.cpp filterHost.c input.bin lib Makefile output.bin
~/ece340/lab5 (hardware_precise)$
```

You can now make any changes that you may want, and after you are done, you can commit those changes and push them to the GitLab, using those commands:

\$ git add <filename>

\$ git commit -m “<a message>”

\$ git push

You should see the following output:

```

~/ece340/lab5 (hardware_precise *)$ git add filterHLS.cpp
~/ece340/lab5 (hardware_precise +)$ git commit -m "Add a comment"
[hardware_precise 78f9235] Add a comment
 1 file changed, 2 insertions(+), 2 deletions(-)
~/ece340/lab5 (hardware_precise)$ git push
Username for 'http://10.96.12.110': test
Password for 'http://test@10.96.12.110':
Counting objects: 3, done.
Delta compression using up to 16 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 300 bytes | 300.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote:
remote: To create a merge request for hardware_precise, visit:
remote:   http://10.96.12.110/test/lab5/merge_requests/new?merge_request%5Bsource_branch%5D=hardware_precise
remote:
To http://10.96.12.110/test/lab5.git
   038ddc4..78f9235 hardware_precise -> hardware_precise
~/ece340/lab5 (hardware_precise)$

```

Note: If you do not want to trigger the CI/CD system, you can add at the end of the commit message the keywords [ci skip]. With that message, no pipeline will run. (e.g. `$git commit -m "message [ci skip]"`).

GitLab CI/CD

After every commit you make on the remote repository, the CI/CD system will start to run the corresponding steps for each branch. The steps are designed to build the necessary software and hardware binaries, and then upload them on the ZCU102 UltraScale+ FPGA, run the application and print the results.

Branch *software* has two CI/CD pipeline steps (Estimated run time: 20 seconds):

1. *build*: cross-compiles the *filter.c* file and generates the *filter* executable.
2. *run*: uploads the **filter** executable binary to the FPGA, alongside with input.bin and output.bin. Then it executes a running script, prints out the results, and then deletes any files on the FPGA, so it can be clean for the next run.

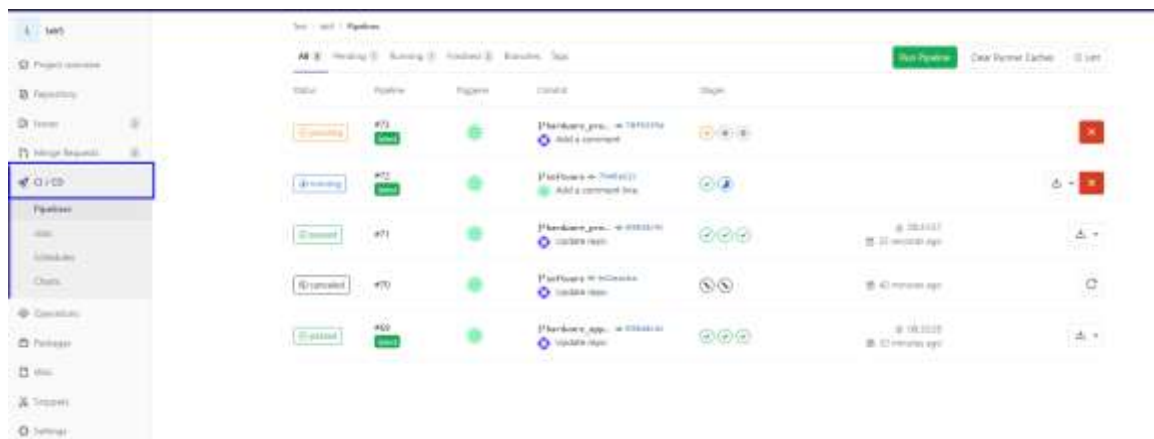
Branches *hardware_precise*, *hardware_approximate* have three steps (Estimated run time: 30 – 92 minutes):

1. *emulation* (1 minute): runs a software emulation, using the Vitis development tools. You can view the results of the application, and check the correctness of the error result. If the error is not in the expected range, you can cancel the rest of the stages. You should ignore the timing results, as they do not represent the real execution time that will be run on FPGA.

2. *build* (30 – 90 minutes) : cross-compiles the *filterHost.c* file and generates the *filter* executable. It also compiles *filterHLS.cpp* and produces the hardware binary file with the name *bilateralFilterKernel.hw.xclbin*.
3. *run* (40 seconds): It creates the bitstream from *bilateralFilterKernel.hw.xclbin*, and then uploads the *filter*, *bilateralFilterKernel.hw.xclbin*, *bilateralFilterKernel.hw.bit* files to the FPGA, alongside with *input.bin* and *output.bin*. Then it executes a running script, prints out the results, and then deletes any files on the FPGA, so it can be clean for the next run.

GitLab CI/CD Web UI

You can view the CI/CD system on GitLab web interface, by clicking the *CI / CD* button:



The steps for every commit make a *Pipeline*, and each step corresponds to a *Job*. A pipeline will fail and stop if any of the jobs in that pipeline produces an error. You can also cancel and resume (*Retry*) any pipeline or job, using the corresponding buttons (Red X for cancel, a closed-loop arrow for *Retry*).


```
2000 [jvs 00-00] Total elapsed time: 00:00:00
2001 #0000 - p acb000
2002
2003 *** v: acb000 - jobs 10 - platform: xilinx_v6b_20000.1 - config: design_emu.cfg - user: temp - g - selected: gpus: kernel: hlsinternalPfilterkernel, kernel_flags: -le - -temp_dir:
2004 /hls340b014_dir/acb000_cx000_custom/hlsinternalPfilterkernel - i - v: acb000/hlsinternalPfilterkernel/acb000_cx000
2005
2006 Option Map File: /hls340b014_dir/acb000_cx000_2/data/v6b014/app/emulab.xml
2007
2008 ***** v: acb000.1 (04-01)
2009
2010 **** hls build 2000070 on host: 0 21:04:14 001 0014
2011
2012 ** Copyright 1996-2019 Xilinx, Inc. All Rights Reserved.
2013
2014 [jvs 00-0000] Additional information associated with this vss link can be found at:
2015 https://www.github.com/xilinx/gaifu/pw/v6b014/hls340b014_dir/acb000_cx000_custom/hlsinternalPfilterkernel/reports/link
2016
2017 log files: /hls340b014_dir/acb000_cx000/gaifu/v6b014/hls340b014_dir/acb000_cx000_custom/hlsinternalPfilterkernel/logs/link
2018
2019 Running Elapsed Server on port 42574
2020
2021 [jvs 00-0000] Creating build summary section with primary output: /hls340b014_dir/acb000_cx000_custom/hlsinternalPfilterkernel/reports/link
2022 at Sat Apr 20 13:53:00 2008
2023
2024 [jvs 00-0100] Initiating connection to refresh server, at Sat Apr 20 13:53:00 2008
2025
2026 Running Rule Check Server on port 42574
2027
2028 [jvs 00-0100] Creating rulecheck section with output: /hls340b014_dir/acb000_cx000_custom/hlsinternalPfilterkernel/reports/link
2029 /hls340b014_dir/acb000_cx000_custom/hlsinternalPfilterkernel/reports/link - Sat Apr 20 13:53:00 2008
2030
2031 [jvs 00-0000] Target platform: /opt/xilinx/platform/xilinx_v6b_20000.1/v6b014_dir/acb000_cx000
2032
2033 [jvs 00-0500] This platform contains Device Support Archive: /opt/xilinx/platform/xilinx_v6b_20000.1/v6b014_dir/acb000_cx000_20000.1.vss
2034
2035 [jvs 00-0000] Loading for software emulation target
2036
2037 [jvs 00-0100] Target device: xilinx_v6b_20000.1
2038
2039 [jvs 00-0400] kernel flags are: -g -i /hls340b014_dir/acb000_cx000/gaifu/v6b014/hls340b014_dir/acb000_cx000_custom/hlsinternalPfilterkernel -le -g
2040
2041 [jvs 00-0000] Created xilinx/hlsinternalPfilterkernel/acb000_cx000
2042
2043 [jvs 00-0300] Run completed. Additional information can be found at:
2044 https://www.github.com/xilinx/gaifu/pw/v6b014/hls340b014_dir/acb000_cx000_custom/hlsinternalPfilterkernel/reports/link/vss_link_hlsinternalPfilterkernel/acb000_cx000
2045
2046 xilinx.vss
2047
2048 Steps log files: /hls340b014_dir/acb000_cx000/gaifu/pw/v6b014/hls340b014_dir/acb000_cx000_custom/hlsinternalPfilterkernel/logs/link/link_steps.log
2049
2050 [jvs 00-0000] Total elapsed time: 00:00:00
2051
2052 /filter: xilinx/hlsinternalPfilterkernel/acb000_cx000
2053
2054 CRITICAL WARNING: [hls 00-0000] Unable to find config file: using default device "xilinx_v6b014_dir/acb000_cx000/v6b014"
2055
2056 [jvs 00-0000] Found 1 platform
2057
2058 [jvs 00-0000] Selected platform 0 from Xilinx
2059
2060 [jvs 00-0000] Found 1 device
2061
2062 load time: 0.00077 milliseconds
2063 filter time: 70.00078 milliseconds
2064
2065 rts: 0.00000
2066
2067 compile time: 1.00200 milliseconds
2068
2069 hls: 0.00000
```

ECE340 Embedded Systems

Multiple compute units

In order to use multiple compute units of the same kernel function, you will need to proceed with the following steps:

1. Change files **design.cfg**, **design_emu.cfg**, by commenting on the connectivity option that is being used, and comment out the second line.
2. Change the **fileHost.c** file:
 - a. Comment in the declaration in line 49.
 - b. Use step 3 for 2 compute units.
 - c. Use step 6 for 2 compute units.
 - d. Create suitable buffers for each compute unit (CU is the accelerator in Vitis terminology). The data buffers must be different for each kernel.
 - e. Split the input and output arrays suitably and map them to the corresponding buffer.
 - f. Set the arguments for each CU instance. First set the arguments for instance 1, enqueue the buffer migration for that instance and queue the task for that instance. It will then start executing, while you can proceed with the next instance.
 - g. Wait for all kernel instances to finish, and then migrate results back to the host.
 - h. Do not forget to clean up the FPGA by releasing every object.

Clock Frequency

The clock frequency that a kernel function will use is configurable. First of all, you must ensure that your design is synthesizable on a target frequency. You can explore this constrain during Vivado HLS development stage, by setting the Clock Period at Synthesis Setting, and checking the estimated clock frequency in HLS synthesis report.

Normally, Vivado designs can freely use any clock frequency in specific ranges (e.g. 10 MHz – 600 MHz), and this option can be configured through Vivado configuration options and constraints. However, in a Vitis development environment, the available clock frequencies must be set at the platform creation stage, and to be hard-coded into the platform. That means that the developer cannot pick any frequency, but only the ones that the platform provides. If no frequency is provided

to Vitis tools, it chooses a default clock frequency. The default frequency is configured at platform creation stage.

For the purposes of this lab, the ZCU102 Board has been setup to provide 4 clock frequencies:

- 100 MHz
- 200 MHz
- 300 MHz (default)
- 400 MHz

You can choose the frequency before compilation, by modifying the **design.cfg** file. There are lines for each clock frequency for HLS, paired with a clock ID. You can comment in the active configuration, and comment out the one that you would like to use.

Note: When your target is software emulation there is no need to configure the clock frequency. You can confirm that, if you view **design_emu.cfg** file that is used for emulation target.