ECE6775 High-Level Digital Design Automation, Fall 2024
School of Electrical and Computer Engineering, Cornell University

Vivado HLS Setup Instructions

1 Introduction

High-level synthesis (HLS) is a hardware design method based on high-level programming languages. It automatically handles low-level details, focuses on architectural design, and greatly improves the productivity of hardware development. In this tutorial, you will set up Vivado HLS [1] on the teaching server, and test your setup by optimizing and running a provided example.

2 Setting Up an HLS Project

In this section, you will establish the HLS project for this tutorial on our ECE Linux server: you will login to the server, set up the Vivado 2019 tools, and create necessary files.

2.1 Logging into the Server

After enrolling in the class, you will be granted access to the Cornell ECE Linux server. You can access it from your terminal with the following command:

```
ssh <NetID>@ecelinux.ece.cornell.edu
```

with `<NetID>` replaced with your Cornell account. If you are connected to the campus network, you can directly log in using the password for your Cornell account. For off-campus access, you will need to first connect to the Cornell VPN following these instructions.

You will be automatically redirected to a specific node in the cluster, named *ecelinux-xx*, where *xx* ranges from 01 to 20. If the current node is slow or unstable, you can directly ssh to another node using `ssh <NetID>@ecelinux-xx.ece.cornell.edu`.

In case you find it challenging to do everything through a plain SSH terminal, you can use the free Visual Studio Code (VS Code) editor which integrates all functions needed. This guide provides useful instructions on setting up VS Code with its remote SSH plugin. Once you are logged into the ECE Linux server from VS Code, you should see a user interface: the "EXPLORER" tab on the left allows you to navigate through the project, the "TERMINAL" tab at the bottom works exactly as an SSH terminal, and the top right blank region is where you will be coding.

2.2 Setting up Vivado HLS Tools

You will be using Vivado HLS version 2019 for this tutorial. To use it on the ECE Linux server, you need to source the setup script by running the following command:

```
source /classes/ece6775/setup-ece6775.sh
```

You must do it every time you open a new terminal, or you may consider adding it to your `.bashrc` to automatically set up the environment on every new terminal you open. Please note that the setup script runs a `module load` command, which modifies your environment to make Vivado HLS available. This might override or conflict with other tools that rely on different library or executable versions.

You can also create an alias within your `.bashrc` file so you can easily set up the tools in the future (`v2019` is just for demonstartion, you may use any name you like):

```
echo "alias v2019=\"source /classes/ece6775/setup-ece6775.sh\"" >> ~/.bashrc
source ~/.bashrc
```

Now, you will just need to type the alias in the terminal to enable the environment.

You can check if the environment is correctly set up by running:

```
which vivado_hls
```

and it should output the following:

```
/opt/xilinx/Vivado/2019.2/bin/vivado_hls
```

2.3 Creating HLS Project

You are given a zip file named *mv-tutorial.zip* under */classes/ece6775* on the ECE Linux server. In your working directory on the server, do:

```
    cp /classes/ece6775/mv-tutorial.zip .
    unzip mv-tutorial.zip
    cd mv-tutorial
```

and you will find four files in the project: *mv.h*, *mv.cpp*, *testbench.cpp*, and *run.tcl*.

2.3.1 Header File (mv.h)

The header file contains the declaration of the **top-level function**, the function that drives the design at top level. In this case, the top-level function is `MV`.

```
// mv.h
// constant definition and function declaration

#define N 16

void MV(int A[N][N], int x[N], int y[N]);
```

2.3.2 The Source File (mv.cpp)

The source file implements the top-level function and includes the header file.

```
// mv.cpp
// original, non-optimized version of matrix-vector multiplication

#include "mv.h"

void MV(int A[N][N], int x[N], int y[N]) {
  int i, j;
  int acc;
OUTER:
  for (i = 0; i < N; i++) {
    acc = 0;
  INNER:
    for (j = 0; j < N; j++) {
      acc += A[i][j] * x[j];
    }
    y[i] = acc;
  }
}
```

2.3.3 Test Bench (testbench.cpp)

The test bench file tests the functionality of your top-level function. It includes the header files for the functions or directories used within the code and a `main()` function that bootstraps the testing of your code. The `main()` function should return non-zero if it finds an error.

```
// matrix-vector multiply test bench

#include <stdio.h>
#include <cstdlib>
#include "mv.h"

int main() {

  int A[N][N]; // matrix
  int x[N];    // vector
  int sw_y[N]; // software output, used for testing
  int y[N];    // hardware output

  int correct; // number of correct outputs

  // temporary variables
  int i, j;
  int sw_acc;

  // initialize input
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      A[i][j] = rand() % 10;
    }
    x[i] = rand() % 10;
    sw_y[i] = 0;
    y[i] = 0;
  }

  // call MV function
  MV(A, x, y);

  // software result
  for (i = 0; i < N; i++) {
    sw_acc = 0;
    for (j = 0; j < N; j++) {
      sw_acc += A[i][j] * x[j];
    }
    sw_y[i] = sw_acc;
  }

  // compare and check results
  correct = 0;
  for (i = 0; i < N; i++) {
    if (y[i] == sw_y[i])
      correct++;
    else
      printf("[ %4d] hw = %6d, sw = %6d\n", i, y[i], sw_y[i]);
  }

  printf("\n\n\t\t%d/%d correct values!\n\n\n", correct, N);

  if (correct == N)
    return 0;
  else
    return 1;
}
```

2.3.4 Tcl Script (run.tcl)

Tcl stands for tool commanding language, which is widely used by CAD tools including Vivado HLS. The following code is an example script for the matrix-vector multiplication design in this tutorial:

```
#====================================================================
 2 # run.tcl
 3 #=====================================================================
 4 # @brief: A Tcl script for synthesizing the matrix-vector multiply design.
 5 #
 6 # @desc: This script launches a simulation & synthesis run
 7 #
 8 #--------------------------------------------------------
 9 
10 # open the HLS project
11 open_project -reset mv.prj
12 
13 # set the top-level function of the design to be fir
14 set_top MV
15 
16 # add design files
17 add_files mv.cpp
18 
19 # always add testbench file for cosim
20 add_files -tb testbench.cpp
21 
22 open_solution "solution1"
23 
24 # use Zynq device
25 set_part xc7z020clg484-1
26 
27 # target clock period is 10 ns
28 create_clock -period 10 -name default
29 
30 # do a c simulation
31 csim_design
32 
33 # synthesize the design
34 csynth_design
35 
36 # do a co-simulation
37 cosim_design
38 
39 # exit vivado HLS
40 exit
```

Note that it includes all building and testing commands in lines 30 - 37; you may not need all of them in a specific step in this tutorial.

3 Synthesizing and Testing the Project

3.1 Testing the Design as a C++ Application

3.1.1 Direct Compilation and Linking with GCC

Any Vivado HLS project can be compiled as a regular C++ application. On the ECE Linux server, you can compile and link the project with GNU Compiler Collection (GCC), by running the command below:

```
gcc mv.cpp testbench.cpp -o mv
```

Once the C++ code is compiled and linked properly, a binary output file named *matvec_gcc* will occur. You may run the binary output file to launch the test bench and obtain results. This is done by running:

```
./mv
```

If everything works well, you should see the following output:

```
16/16 correct values!
```

It is important to note that this tutorial project only involves one source file, one test bench file, and does not use any HLS-specific libraries. As the project gets more complicated in the future, you may need to link more files, and include the HLS libraries accordingly.

3.1.2 Running HLS C Simulation (csim)

A better way to deal with complicated projects, especially for those using HLS-specific libraries (you will learn about these libraries in upcoming labs), is to utilize an HLS feature called csim. Csim also treats your design as an C++ application, but it figures out the libraries on its own as long as you correctly add the source and test bench files in the Tcl script, as shown in lines 4 - 6 in *run.tcl* from Section 2.3.4.

The Tcl command to launch csim is `csim_design`, as shown in line 31 in *run.tcl*. You may comment out the commands in lines 34 and 37 if you only want to run csim. In the terminal, run:

```
vivado_hls -f run.tcl
```

and you will see the following output among many other info messages from HLS:

```
...
INFO: [SIM 211-2] *************** CSIM start ***************
INFO: [SIM 211-4] CSIM will launch GCC as the compiler.
   Compiling ../../../../testbench.cpp in debug mode
   Compiling ../../../../mv.cpp in debug mode
   Generating csim.exe


                16/16 correct values!
```

```
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
...
```

3.2 Building the Output RTL by C Synthesis (csynth)

3.2.1 Launching Synthesis

Csynth refers to the process where the C++ source code is turned into RTL, which is the main job of HLS. The Tcl command to synthesize your design into RTL is `csynth_design`, as shown in line 34 in *run.tcl* from Section 2.3.4. You may comment out the commands in lines 31 and 37 if you only want to run csynth. In the terminal, run:

```
vivado_hls -f run.tcl
```

and you will see the synthesis information printed to the terminal as the tool is running. The synthesis information is also logged into a file named *vivado_hls.log*.

3.2.2 Checking the Synthesis Report

Once csynth is finished, you can find a *MV_csynth.rpt* file within the report folder of your project, which should have the following path:
./mv.prj/solution1/syn/report/MV_csynth.rpt

The synthesis report shows the estimates of the latency and resource utilization of your deisgn. It is highly recommended to review the contents of this file to understand the quality of your design and identify optimization opportunities.

Please save the report elsewhere; you use it as the baseline design when trying out optimizations later in Section 4.

3.3 Testing the Hardware Design by C/RTL Co-simulation (cosim)

3.3.1 Running Cosim

After having the output RTL as the hardware design, you need to check if it has the correct functionality, since improper optimizations may cause the hardware to malfunction. To verify the output RTL, you will utilize the cosim feature of Vivado HLS, which replaces the call to the top-level function in the test bench with RTL simulation.

To launch cosim, you need to make sure csynth has finished so that there is an RTL to be vierfied. The Tcl command to run cosim is `cosim_design`, as shown in line 37 in *run.tcl* from Section 2.3.4. You may comment out the command in line 31 if you only want to run csynth and cosim. In the terminal, run:

```
vivado_hls -f run.tcl
```

and you will see the simulation activity among the output messages:

```
...
// RTL Simulation : 0 / 1 [0.00%] @ "125000"
// RTL Simulation : 1 / 1 [100.00%] @ "10715000"
...
```

if the hardware works well, you will also see a success message:

```
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
```

3.3.2 Checking the Cosim Report

Vivado HLS will measure the actual latency of each call to your top-level function, and generate a cosim report. For our example, you can find *MV_cosim.rpt* at:
./mv.prj/solution1/sim/report/MV_cosim.rpt

It is highly recommended to check the cosim report and compare it with the synthesis report.

4 Trying Out HLS Optimization Directives

Here you will apply optimizations on the provided project and see their effects. The detailed explanation of these optimizations will be covered in later sessions of this course. Please follow the instructions to update the source file, and compare the synthesis results against the baseline design from Section 3.2.2.

Applying pragmas is the main method of optimization in Vivado HLS. Pragmas are compiler directives used by developers to control the behavior of the compiler. In Vivado HLS they always start with `#pragma HLS`. You will learn more about frequently used HLS pragmas later in this course.

Please add two pragmas so that the implementation of `MV` in *mv.cpp* looks like the following (see the pragmas in lines 14 and 15):

```
1  // mv.cpp
2  // original, non-optimized version of matrix-vector multiplication
3
4  #include "mv.h"
5
6  void MV(int A[N][N], int x[N], int y[N]) {
7    int i, j;
8    int acc;
9  OUTER:
10   for (i = 0; i < N; i++) {
11     acc = 0;
```

```
  INNER:
    for (j = 0; j < N; j++) {
      #pragma HLS unroll factor=2
      #pragma HLS pipeline II=1
      acc += A[i][j] * x[j];
    }
    y[i] = acc;
  }
}
```

Please run csynth on the updated project, and compare the report with the baseline. You should be able to see a decrease in latency and an increase in resource utilization.

Note that it is also possible to add optimization directives in the Tcl script instead of adding pragmas in the source code. You may also add `set_directive_unroll -factor 2 MV/INNER` and `set_directive_pipeline -II 1 MV/INNER` before `csynth_design` in *run.tcl* to achieve the same effect as adding pragmas. This technique can be useful when you want to generate different designs from the same source code.

5 Acknowledgement

This tutorial was created by Yixiao Du, Andrew Butt, George Maidhof, and Matthew Hoffman.

References

[1] (2023) Vitis hls. [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html