

Vitis Command Line Flow (Cloud)

2021.2

Abstract

This lab walks you through running a simple vector addition design using the Vitis™ tool command line option using a makefile.

This lab should take approximately 60 minutes.

CloudShare users only: You are provided three attempts to access a lab, and the time allotted to complete each lab is 2X the time expected to complete the lab. Once the timer starts, you cannot pause the timer. Also, each lab attempt will reset the previous attempt—that is, your work from a previous attempt is not saved.

Objectives

After completing this lab, you will be able to:

- Describe the Vitis command line flow
- Create a makefile to compile a project in different modes
- Compile and execute a C application in software emulation, hardware emulation, and hardware deployment modes
- Use the Vitis analyzer tool to view and analyze reports to determine application performance

Introduction

Overview of the vector addition algorithm: The vector addition kernel is described in **K_ALL.cpp** and **kernel.h** as shown below.

```
extern "C" {  
void K_VADD(const unsigned int* A, // Read-Only Vector 1  
            const unsigned int* B, // Read-Only Vector 2  
            unsigned int* R       // Output Result  
            )  
{  
    for (int i=0; i<MAX_Nb_Of_Elements; i++) {  
        R[i] = A[i]+B[i];  
    }  
}
```

Figure 3-1: Vector Addition - K_ALL.cpp File

```
#define MAX_Nb_Of_Elements 1024
```

Figure 3-2: Header file - kernel.h

As you can see the function works with the fixed size (1024 elements) of the A, B and R arrays. The host reads A and B data from the **data_1.txt** and **data_2.txt** files and passes them to the kernel. The host code is located in the **host.cpp** file and uses several sub-functions from **help_functions.cpp (.h)**.

The overall application structure is represented below.

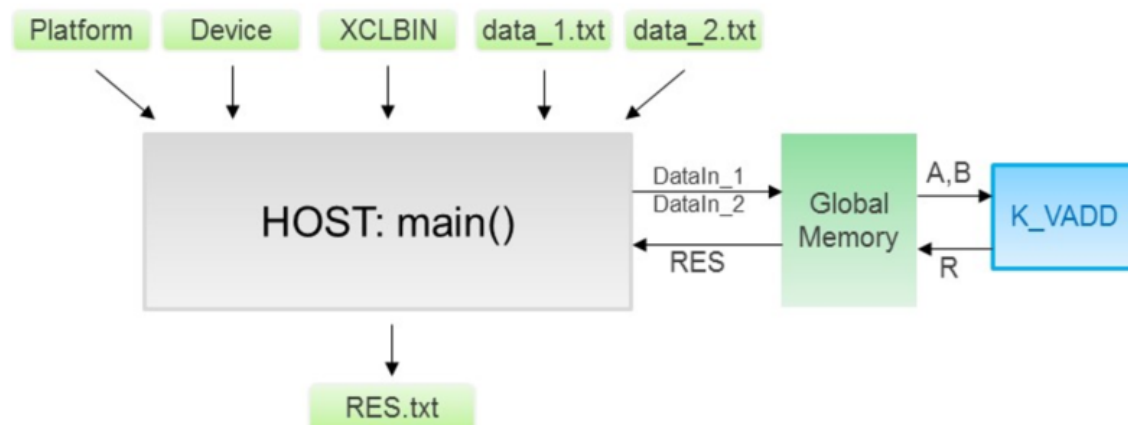


Figure 3-3: Overall Application Structure

The host has five input arguments:

- **Platform:** The name of the platform vendor.
 - In this lab, the vendor is Xilinx.
- **Device:** The target device.
 - In this lab, `xilinx_u50_gen3x16_xdma_201920_3` is used.
- **XCLBIN:** The name of the Xilinx binary container where the `K_ADD` kernel is precompiled.
 - In this lab, the binary container is the `binary_container_1.xclbin` file.
- **data_1.txt:** Input file holding the values of the A array.
- **data_2.txt:** Input file holding the values of the B array.

The application creates the **RES.txt** output file containing the results generated by the `K_VADD` kernel.

Building the Software (Host Program)

The software program in this lab is written in C/C++ and uses OpenCL™ API calls to communicate and control the accelerated kernels. It is built using the standard GCC compiler (or g++ compiler), which is a wrapper around GCC. Each source file is compiled to an object file (.o) and linked with the Xilinx Runtime (XRT) shared library to create the executable.

To compile and link the host application:

Compile: `g++ ... -c <source_file_1> ... <source_file_n> -o <object_file_name> -g`

Linking: `g++ ... -l <object_file_1.0> ... <object_file_n.0> -o <output_file_name>`

Note: Host compilation and linking can be integrated into one step, which does not require the `-c` and `-l` options.

Building the Hardware (Kernel Program - Accelerated Function)

Like building the host application, building the kernels also requires compiling and linking. The hardware kernels can be coded in C/C++, OpenCL C, or RTL. The C/C++ and OpenCL C kernels are compiled using the Vitis compiler, while RTL-coded kernels are compiled using the Xilinx `package_xo` utility.

Hardware compile: `v++ ... -k <kernel_name> <kernel_source_file> ... <kernel_source_file>`

Hardware link: `v++ -t sw_emu ... -l <kernel_xo_file.xo> ... <kernel_xo_file.xo>`

Note: You must specify the target and platform that should match the hardware compile step.

You will use these command lines in the makefile to automate the build and run configuration. Also, there are many libraries and includes to be added for host compilation, and various Vitis compiler options can be set.

Understanding the Lab Environment

Customizable environment variables enable you to tailor your environment for specific machine configurations. The only environment variable (shown below) used in the customer training environment (CustEd_VM) points to the training directory where all the lab files are located.

This environment variable can be customized according to your specific location and can be set for Linux systems in the `/etc/profile` file.

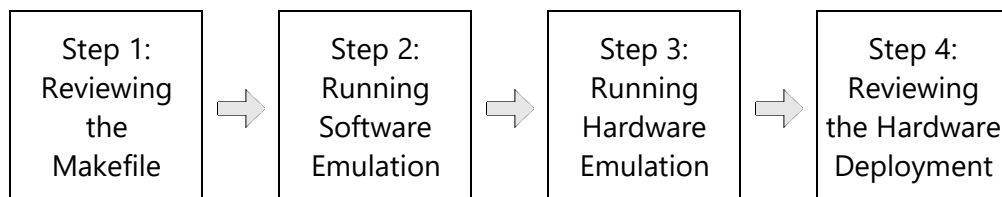
The following is the environment variable used in the customer training VM:

Environment Variable Name	Description
\$TRAINING_PATH	Points to the space allocated for students to work through their labs. This directory includes prebuilt images and starting points for the labs and demos. In the customer training VM, \$TRAINING_PATH sets to the <code>/home/xilinx/training</code> directory.

Note: Environment variables are not supported from the Vitis IDE GUI. When using this tool, you must manually replace `$TRAINING_PATH` with the value of the variable, which in the customer training virtual machine, is `/home/xilinx/training`.

For Cloud development: Similarly, the customer training environment (CustEd_VM) sets the XRT tool install path to `/opt/xilinx/xrt`. Make sure that XRT (2021.2 version) is installed in your environment.

General Flow



Reviewing the Makefile

Step 1

1-1. Set up the operating system's support for the Vitis environment.

1-1-1. If necessary, press **<Ctrl + Alt + T>** to open a new terminal window.

1-1-2. Enter the following commands to source the Xilinx XRT and Vitis tools:

```
[host]$ source /opt/xilinx/xrt/setup.sh
```

```
[host]$ source /opt/Xilinx/Vitis/2021.2/settings64.sh
```

Note: The customer training environment (CustEd_VM) sets the Vitis tool install path to `/opt/Xilinx/Vitis`. If the tool is installed in a different location in your environment, use that install path.

For Cloud development: Similarly, the customer training environment (CustEd_VM) sets the XRT tool install path to `/opt/xilinx/xrt`. Make sure that XRT (2021.2 version) is installed in your environment.

1-2. Review the files in the lab directory.

1-2-1. Enter the following command to change the path to the lab directory:

```
[host]$ cd $TRAINING_PATH/commandline_flow/lab
```

1-2-2. Enter the following command to see the directories:

```
[host]$ ls
```

```
data  makefiles  src
```

Figure 3-4: Directories in the Lab Directory

The table below provides further details on the directories. Note that you can always have your own preferred directory structure. Here the directory structure has been provided for learning purposes.

Directory	Content
data	Contains <code>data_1.txt</code> and <code>data_2.txt</code> , which are the input data files for the application.
makefiles	Contains: <ul style="list-style-type: none"> <code>Makefile</code>: Host code compilation and linking commands; kernel code compilation and linking commands. <code>design.cfg</code>: Contains the Vitis compiler options. <code>xrt.ini</code>: The Xilinx Runtime (XRT) library uses various control parameters to specify debugging, profiling, and message logging when running the host application and kernel execution. These control parameters are specified in a runtime initialization file (<code>xrt.ini</code>) and used to configure features of XRT at startup.
src	Contains all host and kernel design sources.

1-3. Review the `Makefile` file.

1-3-1. Enter the following command to open the `Makefile` file in the editor:

```
[host]$ gedit makefiles/Makefile
```

1-3-2. Review the `Makefile.mk` file, which opens in the `gedit` editor.

The following describes the various steps that the makefile performs:

- Help on usage of the makefile is provided.

```
.PHONY: help

help::
@echo " Makefile Usage:"
@echo ""
@echo " make build TARGET=<sw_emu/hw_emu/hw>"
@echo " Command to generate the design for specified target"
@echo " Default TARGET is sw_emu"
@echo ""
@echo " make run TARGET=<sw_emu/hw_emu/hw>"
@echo " Command to generate,run and verify the design for specified target"
@echo ""
@echo " make clean TARGET=<sw_emu/hw_emu/hw>"
@echo " Command to remove the generated files for specified target"
@echo ""
@echo " make view_run_summary TARGET=<sw_emu/hw_emu/hw>"
@echo " Command to load run summary report in vitis_analyzer utility"
@echo ""
@echo " make view_timeline_trace TARGET=<sw_emu/hw_emu/hw>"
@echo " Command to view application timeline report in vitis_analyzer utility"
@echo ""
```

Figure 3-5: Help Command

A phony target is one that is not really the name of a file; rather it is just a name for a recipe to be executed when you make an explicit request. That is, explicitly declare the target to be phony by making it a prerequisite of the special target .PHONY as help.

Once Makefile is run, "**make help**" will run the recipe. This will display the makefile usage.

- Selecting the platform vendor and target platform (near line no. 26 to 27):
 - Platform vendor is Xilinx
 - Target platform is the Alveo™ U50 accelerator card

```
# =====
# platform selection
# =====
PLATFORM_VENDOR := Xilinx
PLATFORM := xilinx_u50_gen3x16_xdma_201920_3
```

Figure 3-6: Selecting the Vendor and Platform

- Setting the TARGET build configuration, such as software emulation (sw_emu), hardware emulation (hw_emu), or system run (hw) (near line no. 29 to 36):
 - The TARGET variable defines the build configuration and is set to: sw_emu, hw_emu, or hw.
 - The default value for the TARGET variable is set to sw_emu.
 - The PREBUILT variable defines to build the hardware emulation or use the prebuilt hardware emulation files: Default is NO. If you do not want to build the hardware emulation, use YES.

You can override the default build configuration by entering the following command:

```
[host]$ make build TARGET=hw_emu
```

```
# =====
# TARGET can be set as:
# sw_emu: software emulation
# hw_emu: hardware Emulation
# hw : hardware run
# =====
TARGET := sw_emu
PREBUILT := NO
```

Figure 3-7: Setting the Target Build Configuration

- Naming the host executable, XO, and XCLBIN:
 - The following will be the names after building the project:
 - The `HOST_EXE` variable defines the executable name.
 - The XO filename will be `K_VADD.[sw_emu | hw_emu | hw].xo`.
 - The XCLBIN name will be `kernels.[sw_emu | hw_emu | hw].xclbin`.

```
# =====
# Below are the names for host executable and xclbin.
# Please keep it unchanged.
# =====
HOST_EXE := host.exe
XO_NAME  := K_VADD.${TARGET}
XCLBIN   := kernels.${TARGET}.xclbin
```

Figure 3-8: Naming the Host Executable, Kernel, and XCLBIN

- Setting the source directory, input data files (for validation), and build directory location for each build configuration (near line no. 49 to 56):
 - **Note:** If your lab files are pointing to a different location, set the `ROOT_REPO` variable as appropriate.

```
# =====
# Source directory
# =====
ROOT_REPO := /home/xilinx/training
SRC_REPO  := ${ROOT_REPO}/commandline_flow/lab/src
DATA_REPO := ${ROOT_REPO}/commandline_flow/lab/data

PROJECT_DIR := ${ROOT_REPO}/commandline_flow/lab
BUILD_DIR   := ${PROJECT_DIR}/build/${TARGET}

PREBUILT_DIR := ${ROOT_REPO}/commandline_flow/support/prebuilt_hw_emu
```

Figure 3-9: Setting the Directory Locations

- Defining the host code source files and library directories required for host compilation (near line no. 61 to 71):
 - The `HOST_SRC_CPP` variable defines the host source file(s).
 - The `HOST_SRC_H` variable defines the host header file(s).
 - The `KERNEL_SRC_CPP` variable defines the kernel source file(s).
 - The `KERNEL_SRC_H` variable defines the kernel header file(s).
 - The `KERNEL_SRC_H_DIR` variable defines the top directory of the source files.

```
# =====
# Host Application files repository
# =====
HOST_SRC_CPP := $(SRC_REPO)/host.cpp
HOST_SRC_CPP += $(SRC_REPO)/help_functions.cpp
HOST_SRC_H   += $(SRC_REPO)/help_functions.h
HOST_SRC_H   += $(SRC_REPO)/kernel.h

# =====
# Kernel Source Files repository
# =====
KERNEL_SRC_CPP = $(SRC_REPO)/K_ALL.cpp
KERNEL_SRC_H   = $(SRC_REPO)/kernel.h
KERNEL_SRC_H_DIR := $(SRC_REPO)
```

Figure 3-10: Defining the Directories for Host Application and Kernel Source Files

- Defining the host compiler settings and include libraries (near line no. 79 to 85):
 - `-I`: This option is to specify the include directories, such as
`-I$XILINX_XRT/include -I$XILINX_VIVADO/include`.
 - `-L`: This option is to specify directory searches for `-l` libraries, such as
`-L$XILINX_XRT/lib`.
 - `-l`: This option is to specify libraries used during linking, such as `-lOpenCL`
`-lpthread -lrt -lstdc++`.
 - **Important:** The `-lOpenCL` option indicates that the application is compiled against the OpenCL ICD file.
 - `-O0`: Optimization option (execute the least optimization).
 - `-g`: Generate debug info.
 - `-std=c++1y`: Language standard (define the C++ standard instead of the include directory).
 - `-L$XILINX_XRT/lib`: Look in the XRT library.
 - `-lxilinuxopencl, -lpthread, -lrt, and -lstdc++`: Search the named library during linking.


```
# =====
# Host Compiler Global Settings and Include Libraries
# =====
CXXFLAGS += -I$(XILINX_XRT)/include/
CXXFLAGS += -I$(XILINX_VIVADO)/include/
CXXFLAGS += -I$(SRC_REPO)
CXXFLAGS += -O0 -g -Wall -fmessage-length=0 -std=c++1y

CXXLDFLAGS := -L$(XILINX_XRT)/lib/
CXXLDFLAGS += -lxilinuxopencl -lpthread -lrt
```

Figure 3-11: Defining the Compiler Options and Include Libraries

- Defining the Vitis compiler settings and other options (near line no. 90 to 94):
 - `-t`: Set the target build configuration (`sw_emu` | `hw_emu` | `hw`)
 - `--config`: Provide the configuration file, in this case `design.cfg`. In this configuration file, you will set the target platform and connectivity information.
 - `--temp_dir`: Used to specify a location to write the intermediate files.
 - `--log_dir`: Used to specify a directory to store log files.

```
# =====
# Kernel Compiler and Linker Flags
# =====
VPPFLAGS := -t $(TARGET)
VPPFLAGS += --config design.cfg
VPPFLAGS += -I$(KERNEL_SRC_H_DIR)
VPPFLAGS += --temp_dir $(BUILD_DIR)
VPPFLAGS += --log_dir $(BUILD_DIR)
```

Figure 3-12: Defining the Kernel Compiler and Linker Flags

- Generating the host code and kernel code:
 - Host code compilation: (near line no. 99 to 101)
 - `HOST_EXE` provides the name of the `host.exe`. This `host.exe` will be generated using the `$HOST_SRC_CPP` and the `$HOST_SRC_H` files.
 - `g++` is the compiler to be used.

This will create a directory with a name of `$(BUILD_DIR)`. The compiled and linked files will be under the `$(BUILD_DIR)` directory.

```
# =====
# Host Executable File Generation
# =====
$(BUILD_DIR)/$(HOST_EXE): $(HOST_SRC_CPP) $(HOST_SRC_H)
    mkdir -p $(BUILD_DIR)
    g++ $(CXXFLAGS) $(HOST_SRC_CPP) $(CXXLDFLAGS) -o $@
```

Figure 3-13: Building the Host Code Executable

- Kernel code compilation: (near line no. 106 to 112)
 - This is a two-step process: Kernel compilation and linking (v++ compiler command with the -c option and -l option).

This will create a directory with a name of \$(BUILD_DIR). The compiled and linked files will be under the \$(BUILD_DIR) directory.

```
# =====
# Kernel XO and Xclbin File Generation
# =====
$(BUILD_DIR)/$(XO_NAME).xo: $(KERNEL_SRC_CPP) $(KERNEL_SRC_H)
    mkdir -p $(BUILD_DIR)
    v++ $(VPPFLAGS) -c -k K_VADD $(KERNEL_SRC_CPP) -o $$

$(BUILD_DIR)/$(XCLBIN): $(BUILD_DIR)/$(XO_NAME).xo
    mkdir -p $(BUILD_DIR)
    v++ $(VPPFLAGS) -l -o $$ $(BUILD_DIR)/$(XO_NAME).xo
```

Figure 3-14: Building the Kernel Code - XCLBIN

- Generating the emconfig.json file using the emconfig utility.

When software or hardware emulation is run in the command line flow, it is necessary to create an emulation configuration file (emconfig.json) used by the runtime library during emulation. This emulation configuration file defines the device type and quantity of devices to emulate for the specified platform. A single emconfig.json file can be used for both software and hardware emulation.

- --nd: Optional. Specifies number of devices. The default is 1.
- --platform: Defines the target device from the specified platform.
- --od: Optional. Specifies the output directory. When emulation is run, the emconfig.json file must be in the same directory as the host executable. The default is to write the output in the current directory.

```
# =====
# Emulation Files Generation
# =====
EMCONFIG_FILE = emconfig.json

$(BUILD_DIR)/$(EMCONFIG_FILE):
    emconfigutil --nd 1 --platform $(PLATFORM) --od $(BUILD_DIR)
```

Figure 3-15: Generating the emconfig.json File

- Reviewing the primary build target:

```
# =====
# Primary build targets
# ==> build
# ==> run
# ==> view_run_summary
# ==> clean
# =====
.PHONY: all clean

1 # Build the design without running host application
build: $(BUILD_DIR)/$(HOST_EXE) $(BUILD_DIR)/$(XCLBIN) $(BUILD_DIR)/$(EMCONFIG_FILE)

# Build the design and then run host application
2 run:
    pwd
    cp xrt.ini $(BUILD_DIR);
    ifeq ($(TARGET), hw)
        cd $(BUILD_DIR) && unset XCL_EMULATION_MODE; ./$$(HOST_EXE) --kernel_name K_VADD ./$$(XCLBIN);
    else
        ifeq ($(PREBUILT), NO)
            cd $(BUILD_DIR) && XCL_EMULATION_MODE=$(TARGET) ./$$(HOST_EXE) $(PLATFORM_VENDOR) $(PLATFORM) $(XCLBIN)
            $(DATA_REPO)/data_1.txt $(DATA_REPO)/data_2.txt;
        else ifeq ($(PREBUILT), YES)
            cd $(PREBUILT_DIR) && XCL_EMULATION_MODE=$(TARGET) ./$$(HOST_EXE) $(PLATFORM_VENDOR) $(PLATFORM) $(XCLBIN)
            $(DATA_REPO)/data_1.txt $(DATA_REPO)/data_2.txt;
        endif
    endif

# View profile summary report in Vitis Analyzer GUI
3 view_run_summary:
    ifeq ($(TARGET), sw_emu)
        cd $(BUILD_DIR) && vitis_analyzer xclbin.run_summary
    else
        ifeq ($(PREBUILT), NO)
            cd $(BUILD_DIR) && vitis_analyzer kernels.$(TARGET).xclbin.run_summary
        else ifeq ($(PREBUILT), YES)
            cd $(PREBUILT_DIR) && vitis_analyzer kernels.$(TARGET).xclbin.run_summary
        endif
    endif

4 # Clean generated files
clean:
    rm -rf $(BUILD_DIR)/$(XCLBIN) $(BUILD_DIR)/$(HOST_EXE) $(BUILD_DIR)/$(EMCONFIG_FILE) $(BUILD_DIR)/$(XO_NAME).xo
    $(BUILD_DIR)/*.ltx
```

Figure 3-16: Reviewing the Primary Build Targets

1-3-3. Close the Makefile file.

Running Software Emulation

Step 2

Now that you have reviewed the construction of the makefile, it is time to compile the code to run software emulation.

The main goal of software emulation is to validate the functional correctness of the design. In this mode the runtime is very fast because both the host code and the kernel code are compiled to run on an x86 processor.

2-1. Compile for software emulation.

2-1-1. Change the directory to where the compiled files will be generated:

```
[host]$ cd $TRAINING_PATH/commandline_flow/lab/makefiles
```

2-1-2. Enter any one of the following commands to compile the design in software emulation:

```
[host]$ make build
```

OR

```
[host]$ make build TARGET=sw_emu
```

Note: Ignore the warnings.

As noted, by default `TARGET` is set to `sw_emu`. Both of the above commands will have the same effect.

Observe that a new directory `build` has been created under the `$TRAINING_PATH/commandline_flow/lab` directory. Under the `build` directory, you will see one more directory in the name of the target build configuration (`sw_emu`). You will find all the build files under the `sw_emu` directory.

Notice that two files should be generated:

- o `host.exe` (host executable)
- o `kernels.sw_emu.xclbin` (binary container)

The `emconfig.json` file has also been generated. The `emconfigutil` utility generates the `emconfig.json` file, which contains information about the target device. This file is used for the emulation flow.

The `make build` calls the host compilation, kernel compilation, and `emconfig` utility.

```
# Build the design without running host application
build: $(BUILD_DIR)/$(HOST_EXE) $(BUILD_DIR)/$(XCLBIN) $(BUILD_DIR)/$(EMCONFIG_FILE)
```

Figure 3-17: Building the Host Executable, XCLBIN, and emconfig.json File

2-2. Run the software emulation.

2-2-1. Enter the following command to run the application in software emulation mode:

```
[host]$ make run
```

You should see that the application runs successfully.

Observe the following messages in the terminal:

```
training@xilinx$ pwd
/home/xilinx/training/commandline_flow/lab/makefiles
training@xilinx$ make run
pwd
/home/xilinx/training/commandline_flow/lab/makefiles
cp xrt.ini /home/xilinx/training/commandline_flow/lab/build/sw_emu;
cd /home/xilinx/training/commandline_flow/lab/build/sw_emu &&
XCL_EMULATION_MODE=sw_emu ./host.exe Xilinx xilinx_u50_gen3x16_xdma_201920_3
kernels.sw_emu.xclbin /home/xilinx/training/commandline_flow/lab/data/data_1.txt
/home/xilinx/training/commandline_flow/lab/data/data_2.txt;

HOST-Info: Platform_Vendor : Xilinx
HOST-Info: Device_Name : xilinx_u50_gen3x16_xdma_201920_3
HOST-Info: XCLBIN_file : kernels.sw_emu.xclbin
HOST-Info: DataIn_1_File :
/home/xilinx/training/commandline_flow/lab/data/data_1.txt
HOST-Info: DataIn_2_File :
/home/xilinx/training/commandline_flow/lab/data/data_2.txt

VMware: No 3D enabled (0, Success).
HOST-Info: Reading Input data from the
/home/xilinx/training/commandline_flow/lab/data/data_1.txt file ... Read 1024
values
HOST-Info: Reading Input data from the
/home/xilinx/training/commandline_flow/lab/data/data_2.txt file ... Read 1024
values
HOST-Info: Executing Kernel ...
HOST-Info: The Output Result file: RES.txt

Host-Info: =====
Host-Info: Verifying final results (only failed tests are printed)
Host-Info: =====
Host-Info: Test Successful

HOST-Info: DONE
```

Figure 3-18: Application Output (Software Emulation)

The `make run` performs the following:

- `build` (host compilation, kernel compilation, and `emconfig` utility) if it has not already built.
- Copies the `xrt.ini` file to `$(BUILD_DIR)`, which is required. This file should be placed in the same location where the application is running (`host.exe`).
- Sets the `XCL_EMULATION_MODE` environment to `sw_emu`.
- Executes the application. In this application, there are the following arguments:

```
./host.exe <PLATFORM_VENDOR> <TARGET_PLATFORM> <XCLBIN>
<../data/data_1.txt> <../data/data_2.txt>
```

```
# Build the design and then run host application
run: build
    pwd
    cp xrt.ini $(BUILD_DIR);
ifeq ($(TARGET), hw)
    cd $(BUILD_DIR) && unset XCL_EMULATION_MODE; ./${HOST_EXE} --kernel_name K_VADD ./${XCLBIN};
else
    cd $(BUILD_DIR) && XCL_EMULATION_MODE=$(TARGET) ./${HOST_EXE} $(PLATFORM_VENDOR) $(PLATFORM) $(XCLBIN)
    $(DATA_REPO)/data_1.txt $(DATA_REPO)/data_2.txt;
endif
```

Figure 3-19: Running the Application

2-3. Analyze the reports using the Vitis analyzer.

2-3-1. Enter the following to see the reports generated for the software emulation:

```
[host]$ ls ../build/sw_emu
```

You should see some of the following files:

```
host.exe  kernels.sw_emu.xclbin  emconfig.json  RES.txt
xrt.run_summary
```

As you can see, the application generated the `RES.txt` file. You can open the Run Summary report files using the Vitis analyzer.

2-3-2. Enter the following command to open the `xrt.ini` file:

```
[host]$ gedit xrt.ini
```

2-3-3. Observe the following lines in the `xrt.ini` file:

```
[Debug]
openccl_summary=true
openccl_trace=true
data_transfer_trace=fine
[Emulation]
debug_mode=batch
```

2-3-4. Close the file.

2-3-5. Enter the following command to run the application in software emulation mode:

```
[host]$ make view_run_summary
```

You should now see that the Vitis analyzer opens with the Run Summary reports.

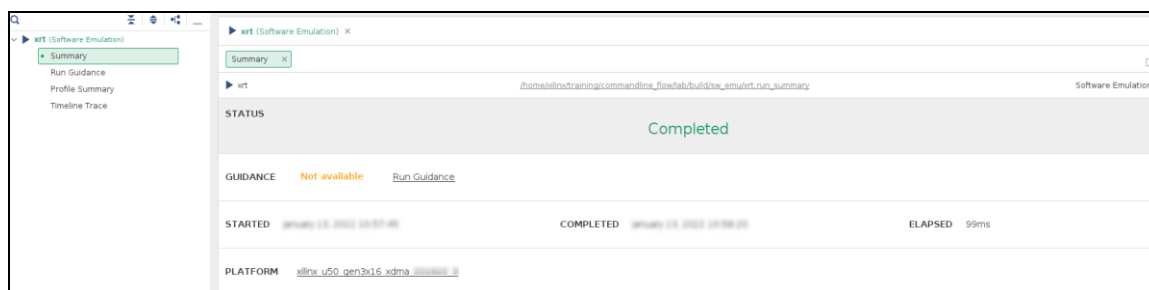


Figure 3-20: Viewing the Run Summary Reports (Software Emulation)

In the Run Summary page, you will find the following details:

- Run Summary file location and the emulation mode
- In the left pane, under xrt (Software Emulation), all the reports generated during the application run are available:
 - Profile Summary: Contains statistical data collected during the application run (host and kernels).
 - Timeline Trace: Provides a graphical view of the execution of the different parts of the application.
 - Run Guidance: Not available for software emulation

2-3-6. In the left pane of the Vitis analyzer, click **Profile Summary** under xrt (Software Emulation) to view the Profile Summary report.

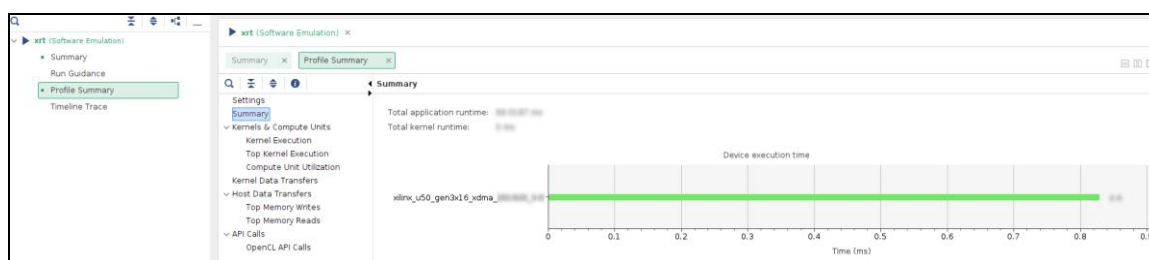


Figure 3-21: Viewing the Summary Section of the Profile Summary

This will show the total application runtime and total kernel runtime.

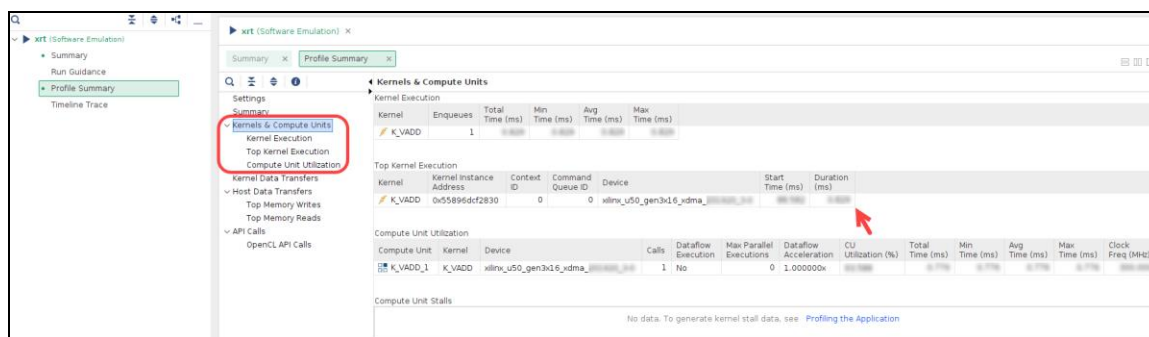


Figure 3-22: Viewing the Kernels and Compute Units

Note: The values may be different for you as they are based on the system configuration.

Observe the kernel execution duration.

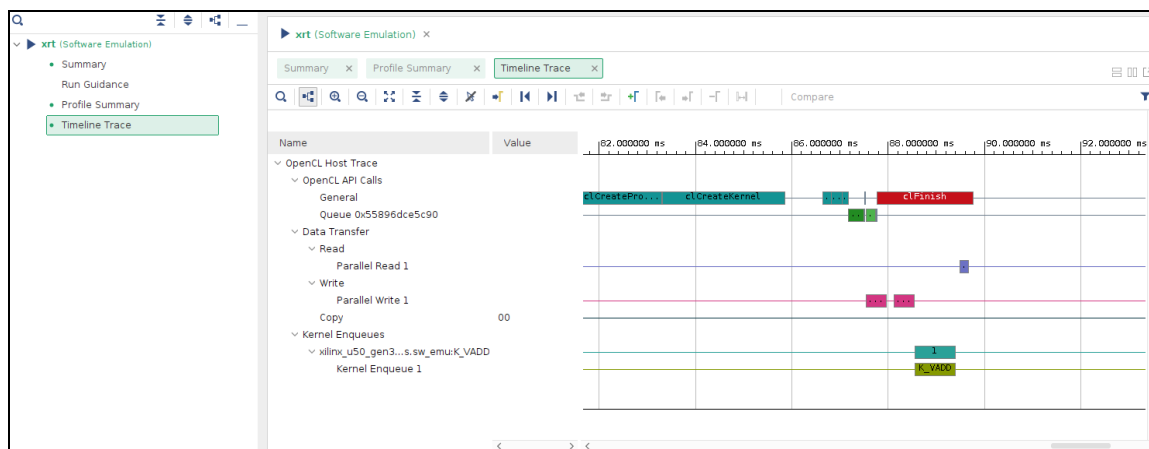
2-3-7. Select the **Kernel Data Transfers** section.

Notice that not all data is available in the report. In the software emulation flow, some of the information is not available in the kernel data transfer area (Top KernelTransfer: Kernels to Global Memory). This information will be available in the hardware emulation and system run flows.

The Profile Summary report has multiple sections that can be selected.

2-3-8. Review each section.

Tab	Description
Settings	Shows the XRT configuration details.
Summary	Displays the total application runtime and total kernel runtime. Kernels and global memory. This tab shows a summary of the top operations. It displays the profile data for top data transfers between the device and device memory.
Kernels & Compute Units	Displays the profile data for all kernels, compute units, and kernel data transfers.
Host Data Transfers	Displays the profile data for all read and write transfers between the host and device memory through the PCIe® core link, if enabled.
API Calls	Displays the profile data for all OpenCL C host API function/XRT API calls executed in the host application.

2-3-9. Click **Timeline Trace** in the left pane of the Vitis analyzer to view the Timeline Trace report.**Figure 3-23: Reviewing the Timeline Trace Report (Software Emulation)**

In addition to the Profile Summary and Timeline Trace reports, you have the option to generate additional reports such as the Estimate and HLS Synthesis reports. This can be done by setting the Report Type option. However these reports will be generated by default in the hardware emulation flow.

2-3-10. After you have reviewed the report, close the Vitis analyzer.

Running Hardware Emulation

Step 3

While the software emulation flow is a good measure of functional correctness, it does not guarantee the correctness of the device execution target.

The hardware emulation flow does enable you to check the correctness of the generated logic. This emulation flow invokes the hardware simulator in the Vitis environment to test the logic functionality. As a consequence, the runtime in the hardware emulation flow is longer than in the software emulation flow.

In addition, the hardware emulation flow provides more complete and precise profiling information, allowing you to analyze the application performance and identify the bottlenecks.

Building the hardware emulation in the VirtualBox environment takes approximately 20-22 minutes (based on your system configuration), whereas in a native Linux machine it will take 7-10 minutes to build the hardware emulation.

If you have time, you can follow step 3-1; otherwise, skip to step 3-2 to use a prebuilt project.

3-1. Compile for hardware emulation.

3-1-1. If you are not already in the `makefiles` directory, change the directory to `makefiles`:

```
[host]$ cd $TRAINING_PATH/commandline_flow/lab/makefiles
```

3-1-2. Enter the following command to compile the design in hardware emulation:

```
[host]$ make build TARGET=hw_emu
```

Note: Ignore the warnings. It may take approximately 20-22 minutes (based on your system configuration) to complete the compilation in the VirtualBox environment. In a native Linux environment, it may take approximately 7-10 minutes.

Under the `build` directory, you will see one more directory in the name of the target build configuration (`hw_emu`). You will find all the build files under the `hw_emu` directory.

Notice that two files should be generated:

- o `host.exe` (host executable)
- o `kernels.hw_emu.xclbin` (binary container)

The `emconfig.json` file has also been generated. The `emconfigutil` utility generates the `emconfig.json` file, which contains information about the target device. This file is used for the emulation flow.

The `make build` calls the host compilation, kernel compilation, and `emconfig` utility.

```
# Build the design without running host application  
build: $(BUILD_DIR)/$(HOST_EXE) $(BUILD_DIR)/$(XCLBIN) $(BUILD_DIR)/$(EMCONFIG_FILE)
```

Figure 3-24: Building the Host Executable, XCLBIN, and emconfig.json File

3-2. Run the hardware emulation.

3-2-1. Enter the following command to run the application in hardware emulation mode:

[Own project]: [host]\$ `make run TARGET=hw_emu`

[Prebuilt project]: [host]\$ `make run TARGET=hw_emu PREBUILT=YES`

If you receive the "Permission denied" message when you run the prebuilt project, change the path to the `support` directory and change the permission settings by using the `chmod` command as shown below:

[host]\$ `cd $TRAINING_PATH/commandline_flow/support/prebuilt_hw_emu`

[host]\$ `chmod 777 host.exe`

Change the path back to the `makefiles` directory and rerun the application for the prebuilt project:

[host]\$ `cd $TRAINING_PATH/commandline_flow/lab/makefiles`

You should see that the application runs successfully.

Observe the following messages in the terminal:

```
training@xilinx$ make run TARGET=hw_emu
pwd
/home/xilinx/training/commandline_flow/lab/makefiles
cp xrt.ini /home/xilinx/training/commandline_flow/lab/build/hw_emu;
cd /home/xilinx/training/commandline_flow/lab/build/hw_emu &&
XCL_EMULATION_MODE=hw_emu ./host.exe Xilinx xilinx_u50_gen3x16_xdma_201920_3
kernels.hw_emu.xclbin /home/xilinx/training/commandline_flow/lab/data/data_1.txt
/home/xilinx/training/commandline_flow/lab/data/data_2.txt;

HOST-Info: Platform_Vendor : Xilinx
HOST-Info: Device_Name      : xilinx_u50_gen3x16_xdma_201920_3
HOST-Info: XCLBIN_file      : kernels.hw_emu.xclbin
HOST-Info: DataIn_1_File    :
/home/xilinx/training/commandline_flow/lab/data/data_1.txt
HOST-Info: DataIn_2_File    :
/home/xilinx/training/commandline_flow/lab/data/data_2.txt

INFO: [HW-EM 01] Hardware emulation runs simulation underneath. Using a large data
set will result in long simulation times. It is recommended that a small dataset is
used for faster execution. The flow uses approximate models for DDR memory and
interconnect and hence the performance data generated is approximate.
HOST-Info: Reading Input data from the
/home/xilinx/training/commandline_flow/lab/data/data_1.txt file ... Read 1024
values
HOST-Info: Reading Input data from the
/home/xilinx/training/commandline_flow/lab/data/data_2.txt file ... Read 1024
values
HOST-Info: Executing Kernel ...
HOST-Info: The Output Result file: RES.txt

Host-Info: =====
Host-Info: Verifying final results (only failed tests are printed)
Host-Info: =====
Host-Info: Test Successful
INFO::[ Vitis-EM 22 ] [Time elapsed: 0 minute(s) 39 seconds, Emulation time:
0.023303 ms]
Data transfer between kernel(s) and global memory(s)
K_VADD_1:m_axi_gmem-HBM[0]      RD = 8.000 KB      WR = 4.000 KB
INFO: [HW-EMU 06-0] Waiting for the simulator process to exit
INFO: [HW-EMU 06-1] All the simulator processes exited successfully

HOST-Info: DONE
```

Figure 3-25: Application Output (Hardware Emulation)

The `make run TARGET=hw_emu` performs the following:

- **build** (host compilation, kernel compilation, and emconfig utility) if it has not already built.
- Copies the `xrt.ini` file to `$(BUILD_DIR)`, which is required if you have built the project yourself. This file should be placed in the same location where the application is running (`host.exe`).
- Sets the `XCL_EMULATION_MODE` environment to `hw_emu`.
- Execute the application. In this application, there are the following arguments:
 - `./host.exe <PLATFORM_VENDOR> Online Portal <XCLBIN>`
`<../data/data_1.txt> <../data/data_2.txt>`

```
# Build the design and then run host application
run: build
    pwd
    cp xrt.ini $(BUILD_DIR);
ifeq ($(TARGET), hw)
    cd $(BUILD_DIR) && unset XCL_EMULATION_MODE; ./$(HOST_EXE) --kernel_name K_VADD ./$(XCLBIN);
else
    cd $(BUILD_DIR) && XCL_EMULATION_MODE=$(TARGET) ./$(HOST_EXE) $(PLATFORM_VENDOR) $(PLATFORM) $(XCLBIN)
    $(DATA_REPO)/data_1.txt $(DATA_REPO)/data_2.txt;
endif
```

Figure 3-26: Running the Application

3-3. Analyze the reports using the Vitis analyzer.

3-3-1. Enter the following to see the reports generated for the hardware emulation if you have built the project yourself; otherwise, skip to the next instruction:

```
[host]$ ls ../build/hw_emu
```

You should see some of the following files:

```
host.exe  kernels.hw_emu.xclbin  emconfig.json  RES.txt
xrt.run_summary
```

As you can see, the application generated the `RES.txt` file. You can open the Run Summary report files using the Vitis analyzer.

The Run Summary report is created by the XRT library during the application execution and provides a summary of the run process. When the Run Summary report is viewed, the tool references the following reports generated during the application run:

- System Diagram
- Platform Diagram
- Run Guidance
- Profile Summary
- Waveform
- Timeline Trace

3-3-2. Enter the following command to view the Run Summary report of the application in hardware emulation mode:

[Own project]: `[host]$ make view_run_summary TARGET=hw_emu`

[Prebuilt project]: `[host]$ make view_run_summary TARGET=hw_emu
PREBUILT=YES`

You should now see that the Vitis analyzer opens with the Run Summary reports.

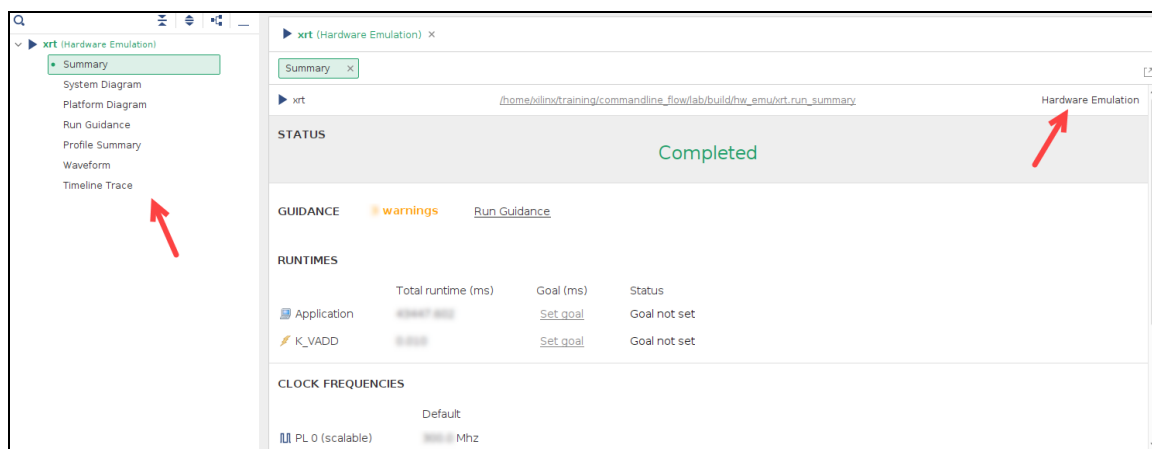


Figure 3-27: Viewing the Run Summary Reports (Hardware Emulation)

In the Run Summary page, you will find the following details:

- Run Summary file location and the emulation mode
- In the left pane, under xrt, all the reports generated during the application run are available:
 - Profile Summary: Contains statistical data collected during the application run (host and kernels).
 - Timeline Trace: Provides a graphical view of the execution of the different parts of the application.

3-3-3. In the left pane of the Vitis analyzer, click **Profile Summary** under xrt (Hardware Emulation) to view the Profile Summary report.

3-3-4. Click the **Kernels & Compute Units** section as shown below.

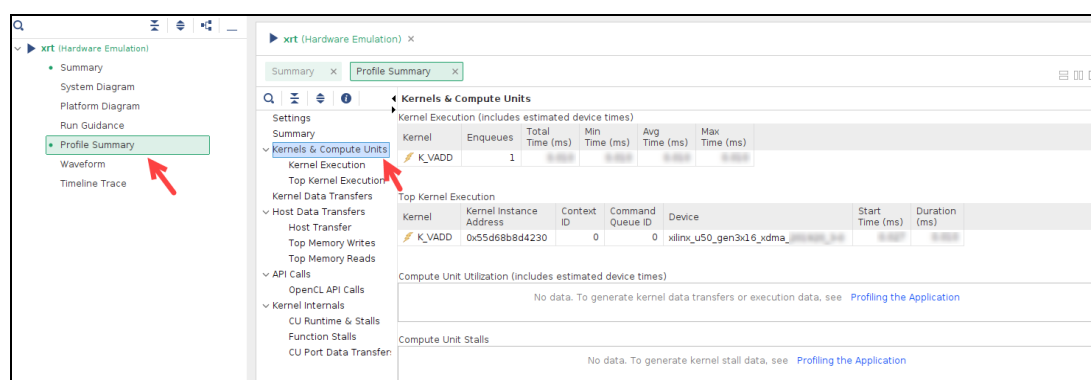


Figure 3-28: Reviewing the Profile Summary Report (Hardware Emulation)

Observe the following information:

- Kernel duration: 0.010 ms (estimated kernel duration for the hardware emulation flow).
 - The same information can be found in the Kernel Execution and Top Kernel Execution sections under Kernels & Compute Units.

3-3-5. In the left pane of the Vitis analyzer, click **Timeline Trace** under xrt (Hardware Emulation) to view the Timeline Trace report.

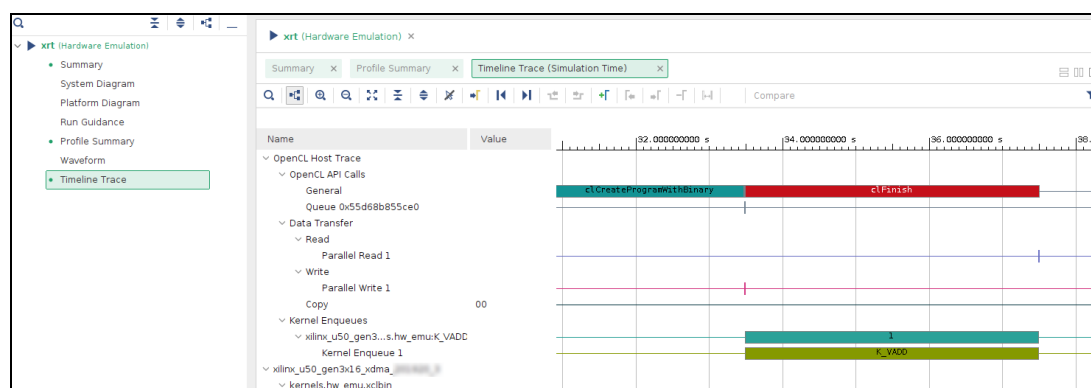


Figure 3-29: Reviewing the Timeline Trace Report (Hardware Emulation)

The Timeline Trace view contains the data transfer information that was missing in software emulation.

- 3-3-6.** In the left pane of the Vitis analyzer, click **Waveform** under xrt (Hardware Emulation) to view the waveform.

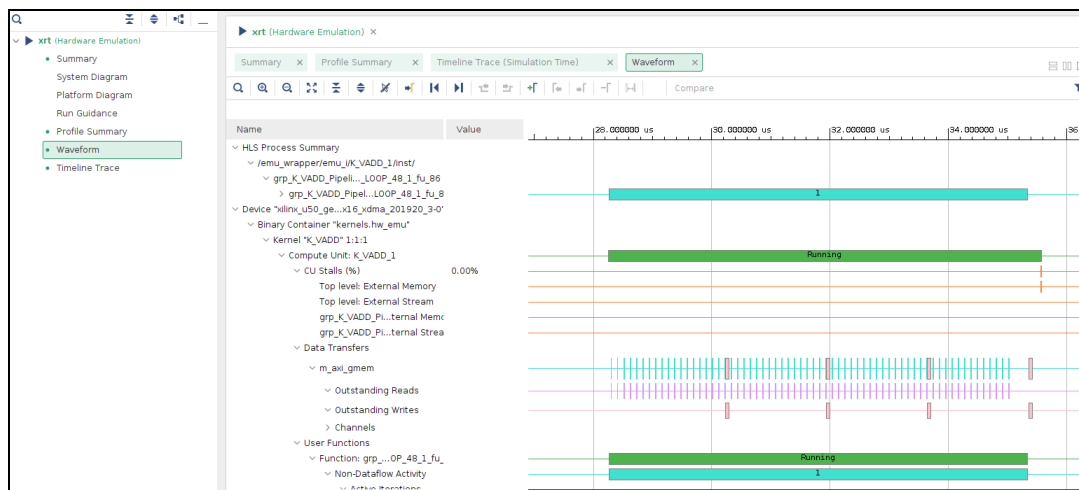


Figure 3-30: Viewing the Waveform

This will show the kernel RTL waveform, which helps with debugging the kernel.

[Prebuilt project]: Skip to step 3-3-10.

- 3-3-7.** Select **File > Open Summary**.

[Own project]: Browse to

`$TRAINING_PATH/commandline_flow/lab/build/hw_emu` and select **kernels.hw_emu.xclbin.link_summary**.

[Prebuilt project]: Skip to step 3-3-10.

- 3-3-8.** Click **Open**.

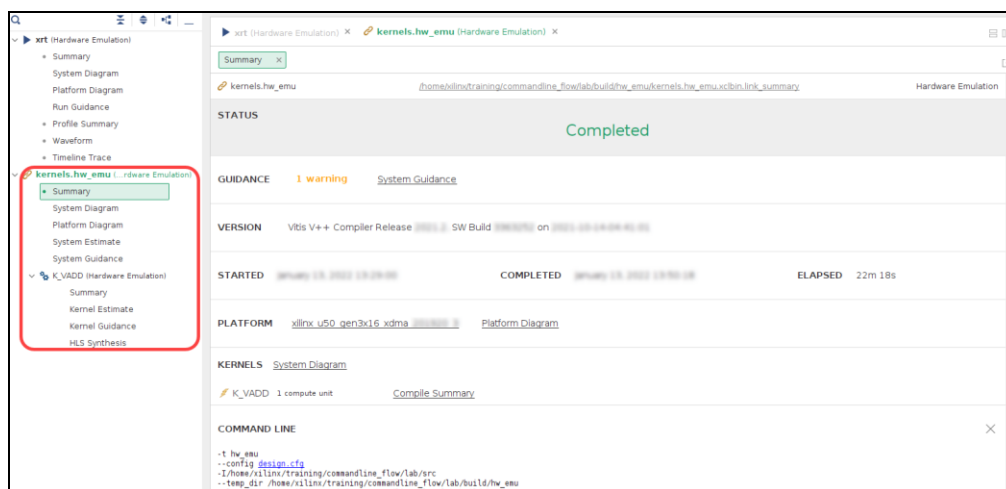


Figure 3-31: Viewing the Link Summary Reports

- 3-3-9.** In the left pane of the Vitis analyzer, click **System Estimate** under **kernels.hw_emu** (Hardware Emulation) to view the System Estimate report.

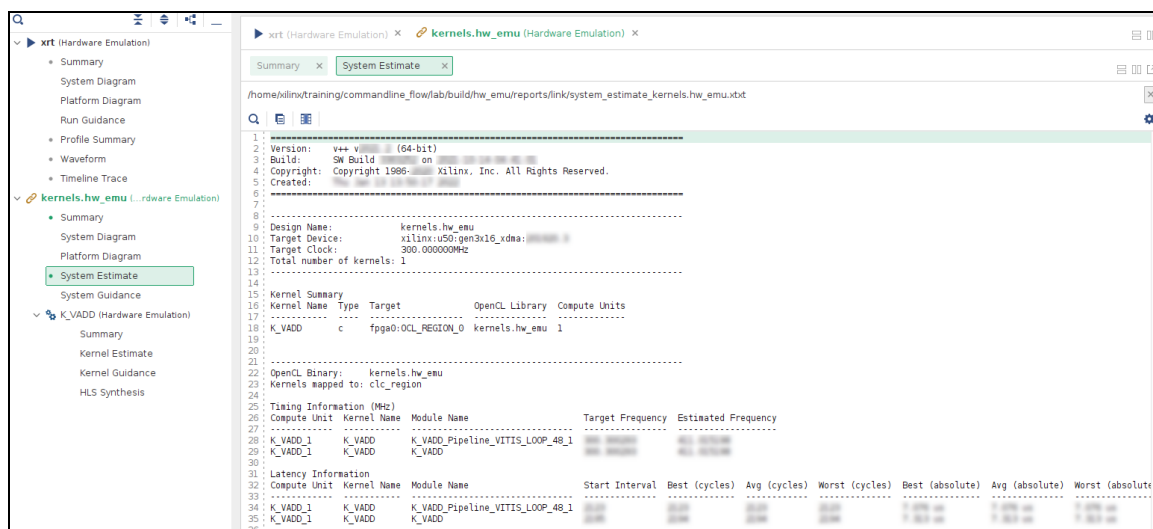


Figure 3-32: Reviewing the System Estimate Report

The System Estimate report provides information on every binary container in the application and every compute unit in the design, such as timing information, latency information, and the area to be used by a compute unit on the target device.

- 3-3-10.** Close the Vitis analyzer.

- 3-4.** For more detailed analysis of the kernel, view the RTL waveform for the kernel in Vivado simulator.

In the `xrt.ini` file add the emulation group `debug_mode` option to enable the waveform generation for the kernel.

The Vitis IDE provides waveform-based HDL debugging in the hardware emulation mode. The waveform is opened in the Vivado waveform viewer, which should be familiar to Vivado logic simulation users.

The Vivado simulator lets you display kernel interfaces, internal signals, and includes debug controls such as restart, HDL breakpoints, as well as HDL code lookup and waveform markers. In addition, it provides top-level DDR data transfers (per bank) along with kernel-specific details including compute unit stalls, loop pipeline activity, and data transfers.

- 3-4-1.** Enter the following command to open the `xrt.ini` file:

[Own project]: [host]\$ `gedit $TRAINING_PATH/commandline_flow/lab/makefiles/xrt.ini`

[Prebuilt project]: [host]\$ `gedit $TRAINING_PATH/commandline_flow/support/prebuilt_hw_emu/xrt.ini`

- 3-4-2.** Near line no. 6, for the `debug_mode` option, replace **batch** with **gui** to enable the waveform generation at the end of the file:

```
[Emulation]
```

```
debug_mode=gui
```

With the `debug_mode=gui` option set, a live waveform viewer will be launched when the application is run.

If the live waveform viewer is activated, the waveform viewer automatically opens when the executable is run. By default, the waveform viewer shows all interface signals and the debug hierarchy.


3-5. Run the hardware emulation and view the waveform in the Vivado Design Suite GUI.

- 3-5-1.** Enter the following command to run the application in hardware emulation mode:

```
[Own project]:[host]$ make run TARGET=hw_emu
```

```
[Prebuilt project]:[host]$ make run TARGET=hw_emu PREBUILT=YES
```

It may take a few minutes to launch the Vivado simulator. You should see the waveform as shown below.

During the execution, the Vivado Design Suite GUI will automatically open with a kernel RTL waveform. You may have to click the **Zoom Fit** icon () in the horizontal toolbar to view the waveform.

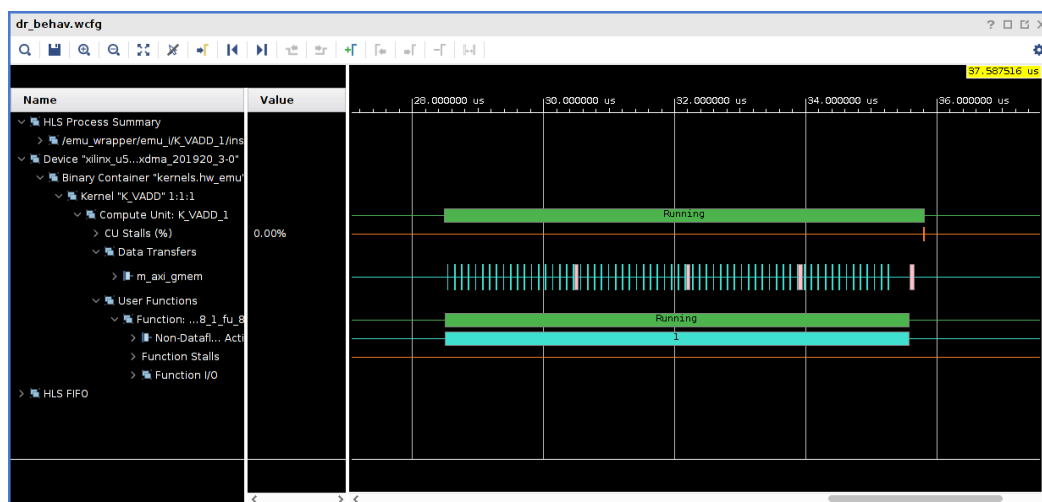


Figure 3-33: Reviewing the Waveform for Kernel Analysis

This is same as you viewed the Waveform in the step 3-3-6.

- 3-5-2.** After you are finished reviewing the waveform, select **File > Exit** to close the Vivado Design Suite.
- 3-5-3.** Click **OK** to exit.

3-6. Analyze the HLS report to understand the kernel performance and area usage.

The generated HLS report will be available under the `reports/<KERNAL_NAME>.hw.emu` directory.

3-6-1. Enter the following command to open the HLS synthesis report:

[Own project]: [host]\$ `gedit ../build/hw_emu/reports/K_VADD.hw_emu/hls_reports/K_VADD_csynth.rpt`

[Prebuilt project]: [host]\$ `gedit $TRAINING_PATH/commandline_flow/support/prebuilt_hw_emu/K_VADD_csynth.rpt`

3-6-2. Review the Vitis HLS Synthesis report, which helps for detailed kernel performance/area analysis.

3-6-3. After you are finished reviewing the report, close the editor.

Note that you can also view this report by using `kernels.hw_emu.xclbin.link_summary`, which is available under the TARGET build directory.

Known issue: From the Vitis analyzer, this report does not open.

Reviewing the Hardware Deployment

Step 4

Building the design for a system run will take about three hours. Do not execute any of the steps below. Just review the steps on how to build the system and run it on a board.

4-1. Only review the steps below on how to build the design in a system configuration and run it on a board.

4-1-1. Enter the following command to compile the design in hardware emulation:

[host]\$ `make build TARGET=hw`

Note: Ignore the warnings. It may take approximately 4-5 hrs (based on your system configuration) to complete the compilation in the VirtualBox environment.

Under the `build` directory, you will see one more directory in the name of the target build configuration (`hw`). You will find all the build files under the `hw` directory.

Notice that two files should be generated:

- o `host.exe` (host executable)
- o `kernels.hw.xclbin` (binary container)

The `emconfig.json` file has also been generated. The `emconfigutil` utility generates the `emconfig.json` file, which contains information about the target device. This file is used for the emulation flow.

The `make build` calls the host compilation, kernel compilation and `emconfig` utility.

```
# Build the design without running host application
build: $(BUILD_DIR)/$(HOST_EXE) $(BUILD_DIR)/$(XCLBIN) $(BUILD_DIR)/$(EMCONFIG_FILE)
```

Figure 3-34: Building the Host Executable, XCLBIN, and `emconfig.json` File

4-2. Run the application on the board.

4-2-1. Enter the following command to run the application on hardware:

```
[host]$ make run TARGET=hw
```

```
# Build the design and then run host application
run: build
    pwd
    cp xrt.ini $(BUILD_DIR);
    ifeq ($(TARGET), hw)
        cd $(BUILD_DIR) && unset XCL_EMULATION_MODE; ./${HOST_EXE} --kernel_name K_VADD ./${XCLBIN};
    else
        cd $(BUILD_DIR) && XCL_EMULATION_MODE=$(TARGET) ./${HOST_EXE} $(PLATFORM_VENDOR) $(PLATFORM) $(XCLBIN)
        $(DATA_REPO)/data_1.txt $(DATA_REPO)/data_2.txt;
    endif
```

hw RUN

Figure 3-35: Running in Hardware Emulation

While running the application in hardware mode, make sure the `XCL_EMULATION_MODE` environment variable should be unset.

You should see that the application runs successfully.

4-3. Analyze the reports using the Vitis analyzer.

4-3-1. Enter the following command to see the reports generated for the hardware run:

```
[host]$ ls ../build/hw
```

You should see some of the following files:

```
host.exe  kernels.hw.xclbin  emconfig.json  RES.txt
xrt.run_summary
```

4-3-2. Enter the following command to run the application in hardware mode:

```
[host]$ make view_run_summary TARGET=hw
```

You should now see that Vitis analyzer opens with the Run Summary reports.

To see the system estimate reports, run the Vitis analyzer with the Link Summary report. This will give the actual resource usage and the performance of the system.

For CloudShare users, you can skip the "Clean up the VirtualBox file system" instructions below.

4-4. Clean up the VirtualBox file system.

4-4-1. Enter the following command to delete the contents of the workspace:

```
[host]$ rm -rf $TRAINING_PATH/commandline_flow
```

This will recursively delete all of the files in the `$TRAINING_PATH/commandline_flow` directory.

Summary

In this lab you have learned how to run and analyze design from the Vitis command line with a makefile.