Installing Linux on a DE10-Nano with support for OpenCL, HDMI, and Webcams

Mackenzie Hauck

Table of Contents

1. Overview

This document will go over all of the steps needed to get embedded Linux with OpenCL drivers, HDMI video out, and webcam drivers running on the Terasic DE10-Nano board.

NOTE	If you do not need a Linux image with both HDMI output, AND OpenCL to be running, you can use an official SD card image provided by Terasic or Intel.

1.1. Prerequisites

Hardware:

- Terasic DE10 Cyclone V development board
- Mini-USB to USB A cable
- 4GB+ MicroSD card
- Ethernet cable
- SD card reader
- A PC that can run Quartus
- A PC/server running Linux (to compile the Linux kernel)
- Optional: webcam, USB OTG to USB A adapter

This document assumes you are at least somewhat familiar with the following topics:

- Linux (navigating via the commandline, and commandline tools)
- Quartus, and Qsys (now known as Platform Designer in 17.1 and up)
- Basics of VHDL or Verilog

No prior knowledge is required in:

- OpenCL
- Compiling the Linux kernel
- Linux drivers

1.2. Hardware

The DE10-Nano includes a single CycloneV SoC (system on chip) composed of a dual core ARM Cortex A9 CPU and a FPGA with 110k logic elements, a MicroSD card slot, USB host via an OTG (on-the-go) port, and ethernet. Intel supports running Linux on the SoC which then requires drivers to communicate with ethernet, USB webcams, FPGA fabric, and OpenCL custom components that *run in* the FPGA fabric. Most of these drivers are available in the Linux kernel source, so including them is as simple as

enabling them in the kernel configuration. The one not supported directly by the kernel is the OpenCL driver. Intel provides the source code of this driver which can be compiled to be used with our kernel.

Our operating system (Linux) is stored on the MicroSD card, and the setup is very similar to that of a Raspberry Pi. The architecture of the A9 cores is `armv7` (the same as Raspbery Pi 2 and up), and includes floating point hardware, so looking at tutorials for those Pi's may be useful to get cross compilers or other software installed. The prefix for this architecture is `arm-linux-gnueabihf` where the `hf` stands for "hard float", i.e., dedicated floating point hardware. The processor also supports the NEON SIMD instructions.

1.3. Embedded Software

In order to more easily take advantage of the FPGA logic from Linux, Intel provides an OpenCL offline compiler allowing us to write "high-level" OpenCL code that then gets compiled down into a bitstream that can be loaded onto the FPGA. The Linux kernel communicates with the custom logic on the FPGA by way of an Intel provided kernel driver and the matching Qsys component. With this driver loaded, we can run what's known as an "OpenCL host program" which uses the standard OpenCL interface to talk to the FPGA (via the kernel driver). This host program is run in userland in Linux. No FPGA specific code is required in the host program, and it can be coded in many languages with OpenCL support like C++, C, Rust, Python, etc. Theoretically the host program could be written in a way to be backend agnostic, where the OpenCL device could be a GPU, FPGA, etc. However, in practice, the workflows are slightly different:

- The FPGA requires pre compiled OpenCL code compared to GPUs using the OpenCL source code
- FPGA code may contain Intel specific OpenCL extensions which will only run on Intel FPGAs

1.4. SD Card layout

Our operating system is stored on the SD card, split into several partitions. The two partitions we are interested in are the root file system partition (EXT3), and the "boot" partition that contains the bootloader, u-boot script, `.rbf` used to configure the FPGA on boot, and the compressed kernel image file. The boot partition is FAT32, so we can access its contents from Windows, or another operating system to modify the files that are used during boot (`zImage` and `opencl.rbf` files).

Due to the difficulty of of compiling everything from scratch and creating an SD card image, we will instead be modifying an existing SD card image from the user thinkoco on github, then updating the kernel with webcam drivers enabled, and installing some software in Linux.

Much of the content in this guide has been adapted from thinkoco's guide which you can view here.

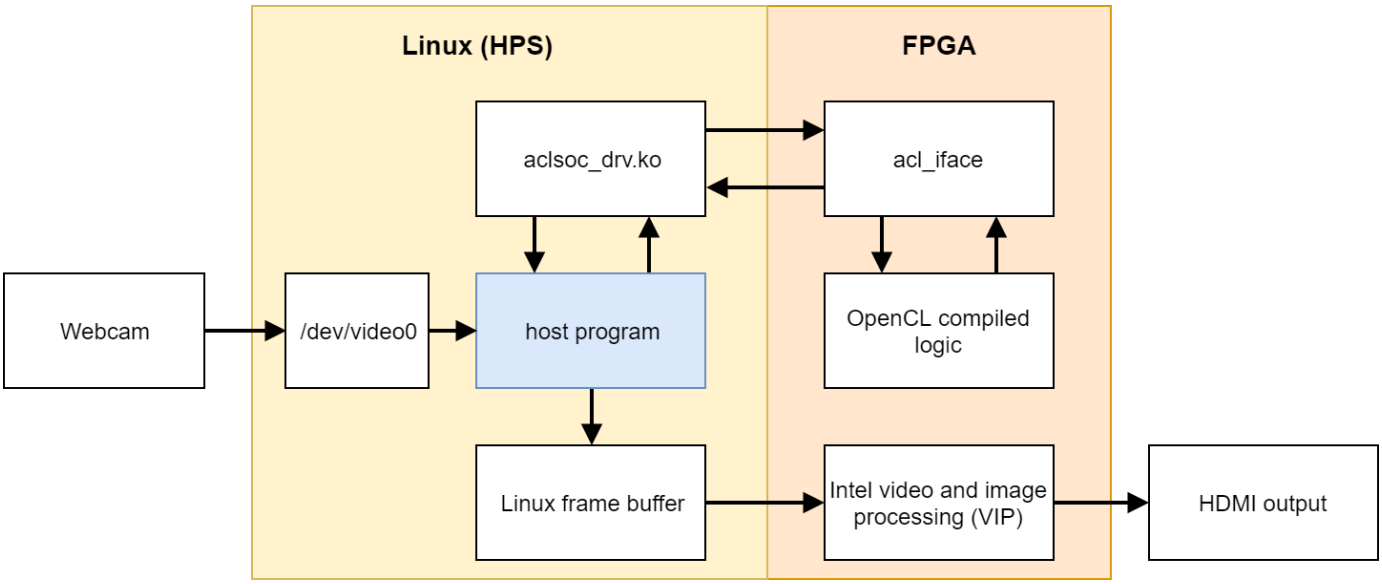
Many thanks to thinkoco (Welon Tank) for their guides, and support!

TIP

The site RocketBoards.org (run by Intel) has what is known as the "Golden System Reference Design (GSRD)" which is a good starting point for working with Intel FPGAs. https://rocketboards.org/foswiki/Documentation/GSRD

Here is the documentation on the SD card layout if you want to learn more.

1.5. Block diagram



2. Preparing our donor SD card image

We will be using the SD card image provied by the user thinkoco on Github which uses kernel version 3.18 with a root file system containing Ubuntu 16.04.

NOTE

The file system also already includes the Intel RTE (Runtime Environment, a subset of the FPGA SDK for OpenCL). The RTE contains diagnostic tools, a compiled kernel driver to communicate with the OpenCL FPGA logic, and the OpenCL dynamic libraries (`.so` files) that the host program is linked against.

2.1. Download the image

The `.img` file can be downloaded from [here]

2.2. Load the image onto a MicroSD card

On Windows use Win32DiskImager, on Mac or Linux use `dd`.

You can follow Intel's guide on writing the image with these tools here.

Connect the DE10-Nano to your development PC with the Mini-USB cable and open a serial connection with baud rate 115200.

NOTE	During boot this serial port will display the Linux boot sequence, and after booting, display a login prompt. The default username is `root` and there is no password.

Insert the SD card into the DE10-Nano, and apply power. If you get a login prompt, you are ready for the next step.

3. Compiling our custom Linux kernel

The Linux kernel provided by thinkoco does not have USB webcams enabled, so we will build our own kernel with the required drivers. In the FAT32 boot partition, the kernel is the `zImage` file.

TIP	A `zImage` is a compressed kernel image, signified by the 'z'. An uncompressed kernel image is named `uImage`.

Our built kernel (`zImage`) will also disable strict checking for loadable modules. This means that even if the version magic of a module does not match exactly, it will still be loaded by the kernel.

NOTE	Our OpenCL Linux kernel driver is loaded as module into the kernel, hence the need for loadable module support.

WARNING	Loading kernel modules that weren't compiled against the *exact* kernel version (3.18) may lead to kernel panics with this setting disabled. Be sure to re-enable for non-development builds.

The following steps were done on a server running Ubuntu 16.04.

3.1. Installing dependencies

```
$ sudo apt update
$ sudo apt install u-boot-tools gcc-arm-linux-gnueabihf g++-arm-linux-gnueabihf
libncurses5-dev make lsb uml-utilities git
```

3.2. Downloading kernel source

Apparently Intel does not like keeping source code public that is vulnerable to some exploits, so they have removed their 3.18 branch from github possibly due to the Meltdown / Spectre exploits. Instead, we will download the kernel source from thinkoco's github repository.

We can either clone the whole repository (~1.4GB), or just a single branch as shown below. If you clone

the entire repo, be sure to checkout the 3.18 branch.

```
$ cd ~
$ git clone --single-branch -b socfpga-opencl_3.18 https://github.com/thinkoco/linux-
socfpga.git
$ cd linux-socfpga
```

3.3. Building with a modified config

We will use a config already modified to include support for webcams.

NOTE	The `.config` file holds the configuration for building the kernel and should only be modified by certain tools. See the appendix for how to modify it with a `ncurses` frontend.

```
$ cp 3.18_usbcam_config .config
# setup compiler options
$ export ARCH=arm
$ export CROSS_COMPILE=arm-linux-gnueabihf-
$ export LOADADDR=0x8000

# setting LOCALVERSION blank means that kernel modules will
# not have to match their version exactly to the kernel
$ export LOCALVERSION=

# build with 24 threads. replace 24 with the number of threads on your machine.
# with 24 cores this took ~90 seconds
$ make -j24 zImage

# copy the compiled kernel
$ cp arch/arm/boot/zImage ~/output/
```

Now that we have our modified `zImage`, we can insert the MicroSD card into our computer and overwrite the original `zImage` in the FAT32 partition. Once again, boot the DE10 with the USB serial connected and verify you get to the login prompt.

4. Installing and running OpenCL diagnostic tools

First, we need to expand the root filesystem to give us the full space on the SD card.

```
$ cd ~
$ ./expand_rootfs.sh
# follow the scripts instructions after it finishes.
```

4.1. Installing `clinfo`

We need to copy some files around so the diagnostic tools know what OpenCL devices are available.

```
$ cd ~/aocl-rte-17.1.0-590.arm32
$ mkdir -p /etc/OpenCL/vendors
$ cp Altera.icd /etc/OpenCL/vendors/
$ mkdir -p /opt/Intel/OpenCL/Boards
$ echo /home/root/aocl-rte-17.1.0-590.arm32/board/c5soc/arm32/lib/libintel_soc32_mmd.so >
/opt/Intel/OpenCL/Boards/c5.fcd

# install the `clinfo` tool which prints information about available OpenCL devices
$ sudo apt install clinfo

$ clinfo
```

4.2. Intel's `aocl` tool

After initializing the environment variables for use with the RTE, we can run `aocl diagnose` to check that the kernel module loaded correctly.

```
$ cd ~
# set the environment variables, and load the OpenCL driver
$ source ./init_opencl_17.1.sh
$ aocl diagnose
```

5. Quartus setup

5.1. Installation

Download and install Intel FPGA for OpenCL 17.1 edition of Quartus.

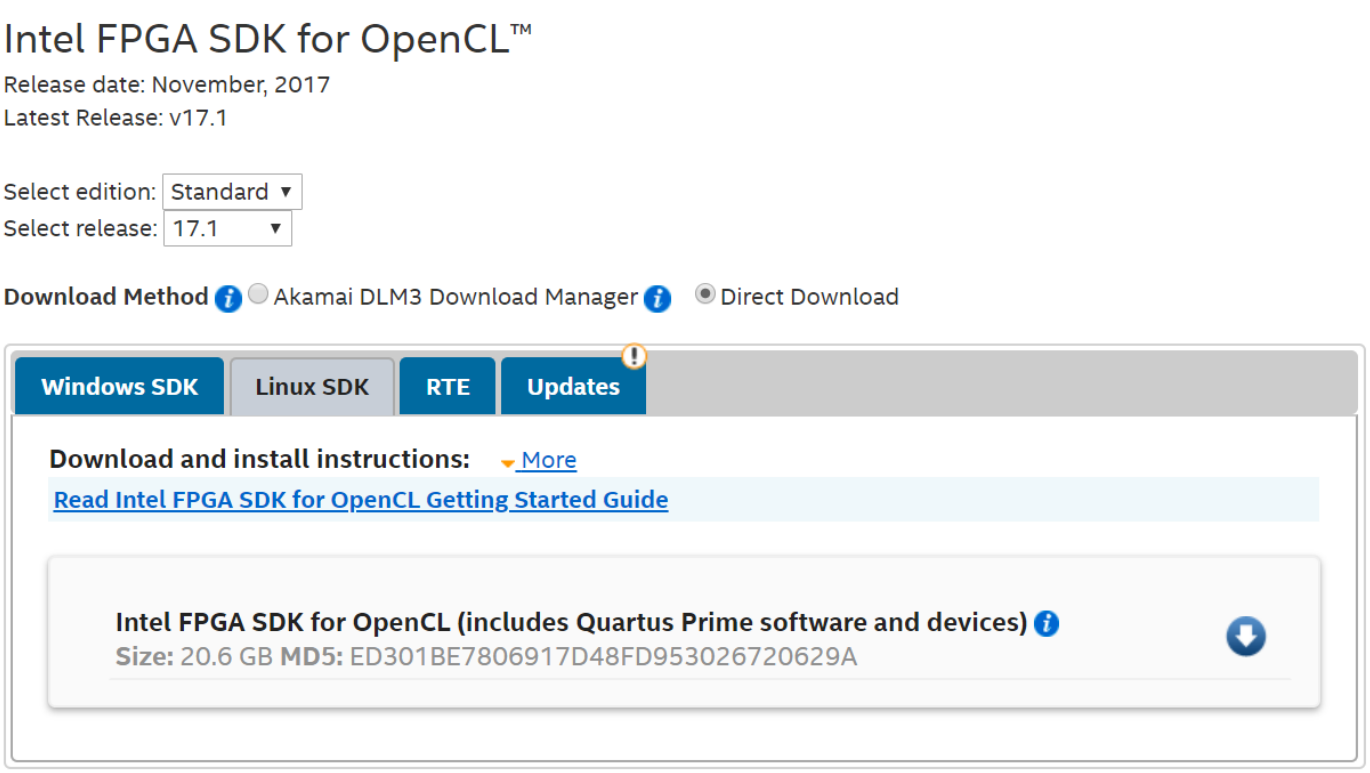


Figure 1. Screenshot of download page

When installing, be sure to enable support for Cyclone V devices. We will need at least a Standard license in order to compile everything.

TIP	See the appendix for working from home with a VPN and using the University of Alberta license server.

5.1.1. Environment Variable setup

Follow Intel's instructions from their getting started guide here.

Table 1. Table Summary of Variables (on Windows)

Variable	Value
`INTELFPGAOCLSDKROOT`	`C:\intelFGPA\17.1\hld`
`AOCL_BOARD_PACKAGE_ROOT`	`C:\intelFPGA\17.1\hld\board\c5soc`
`PATH`	append these to `PATH`: • `%INTELFPGAOCLSDKROOT%\bin` • `%INTELFPGAOCLSDKROOT%\windows64\bin` • `%INTELFPGAOCLSDKROOT%\host\windows64\bin`

5.2. Board Support Package (BSP)

To go with the Linux image provided by thinkoco, we will also use their board support package (BSP).

NOTE

What is a BSP?

A BSP allows us to include non OpenCL code in our FPGA fabric. This is where we add custom Qsys components, custom VHDL or Verilog code, etc.

The BSP we use contains:

- Qsys component `acl_iface` "OpenCL interface" which is what our Linux kernel driver will use to communicate with the compiled OpenCL code.
- Qsys component for HDMI output
- Qsys components for onboard switches, buttons, and LEDs

5.2.1. Installation

Download or clone thinkoco's repository.

Copy the folder `de10_nano_sharedonly_hdmi` to the following folder: `C:\intelFPGA\17.1\hld\board\c5soc\hardware` or equivalent if on Linux.

NOTE

If you do not have admin access to this folder on Windows, there are ways to setup the environment variables from earlier to work around it, but they will not be covered here.

TIP

You can view the contents of the BSP by opening the above folder's `top.qpf` file in Quartus, and opening `system.qsys` and `iface/acl_iface_system.qsys` in Platform Designer.

6. Compiling and running our own host program, and OpenCL code

The offline OpenCL compiler is invoked with the `aoc` command at the commandline.

6.1. Compiling `.cl` file into `.aocx` and `.rbf` files with `aoc`

Intel's offline OpenCL compiler is invoked via the `aoc` command from the commandline. We can compile our `trivial.cl` OpenCL source file into the required `trivial.aocx` with:

```
$ aoc -v -report -board=de10_nano_sharedonly_hdmi trivial.cl -o trivial.aocx
```

This compiles with verbose message printing, produces a report, and selects the BSP.

Since we need a `.rbf` file we can convert the generated `top.sof` to `opencl.rbf` with:

```
$ quartus_cpf -c -o bitstream_compression=on top.sof opencl.rbf
```

Now copy the generated `opencl.rbf` to the FAT32 partition of the SD card. The FPGA will be programmed with this file on every boot.

6.2. Install Rust on the DE10-Nano

I am most familiar with Rust, so the example will be written in it, but many languages support the OpenCL interface (C, C++, Python, Java, etc.).

See the included files (`example-rs/src/main.rs`) to see the Rust code. Cross compiling is a pain since we have to link to the OpenCL headers, so we will take the easy way out and install Rust on the DE10.

```
$ cd ~
$ mkdir rust
$ wget https://static.rust-lang.org/rustup/dist/armv7-unknown-linux-gnueabihf/rustup-init
$ chmod +x rustup-init
$ ./rustup-init
$ [enter]
# after finished installing, follow instructions to add the rust executables to `PATH`

# install the OpenCL headers
$ sudo apt install ocl-icd-opencl-dev
```

6.3. Run the example on the DE10

Copy the example code to the DE10 into `~/example-rs`

Load the OpenCL kernel module, and set the required environment variables:

```
$ cd ~
$ source ./init_opencl_17.1.sh
```

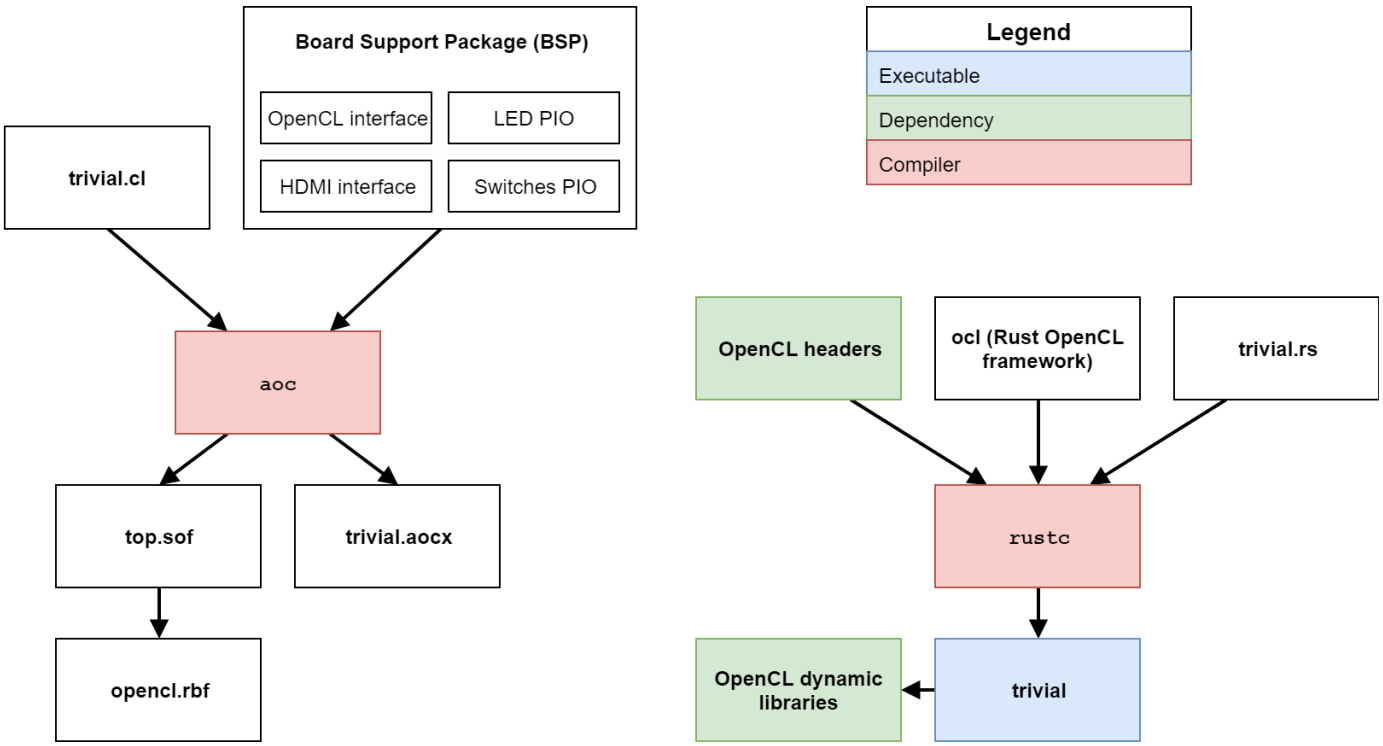
NOTE	Rust's package manager `cargo` needs an internet connection to download the required dependencies.

```
$ cd ~/example-rs
$ cargo run --release trivial.aocx
```

Where `trivial.aocx` is the compiled output of `trivial.cl`. For more info see Compilation flow

7. Appendix

7.1. Compilation flow



7.2. The device source tree

In the FAT32 partition of the SD card there is a file named `socfpga.dtb` which is a "device tree blob". The device tree is a way to compile a single Linux kernel that can run on different hardware, as it defines where all of the hardware components are located.

It includes things like the kind of processor, amount of RAM, the HDMI Qsys component's address, FPGA manager address, button addresses, onboard LED addresses, etc. You can view Intel's documentation on how to modify a `.dts` file (device tree *source*) which is then compiled into the above `.dtb` file.

Intel documentation

7.3. Modifying Linux kernel config

The config can be modified on the command line by running the following in the kernel source directory:

CAUTION	The `ARCH` and `CROSS_COMPILE` environment variables should be set before running this command

```
$ make menuconfig
```

This will give you an ncurses interface to modify the config. Be sure to backup your config before running `make clean` as I believe it deletes it.

7.4. Running graphics programs on the HDMI output, from the USB serial port

It is convenient to use the shell provided on the USB serial port, but usually it does not let us open programs on the HDMI output.

To open these programs on the HDMI output, export the `DISPLAY` environment variable.

```
$ export DISPLAY=":0"
```

7.5. Working remotely with VPN

We need a license to compile the OpenCL code with Quartus, so to work remotely we can use the university's VPN to contact the U of A license server.

Install the university VPN client from here.

Login with your CCID, and password.

NOTE	The VPN will only tunnel traffic to *.ualberta.ca domains, so your external IP won't change.

Close Quartus if it's open.

Set the `LM_LICENSE_FILE` environment variable to: 12000@lic.ece.ualberta.ca

When opening Quartus, set the license to use the environment variable.

7.6. Compiling `.cl` for software emulation

Since `aoc` takes around 1 hour to finish when compiling for hardware, we can instead use its emulator to quickly check the functionality of the `.cl` code before committing to the full build time.

To compile for emulator add `-march=emulator` to the `aoc` command. The OpenCL host program can then be compiled for the development PC, and run with this `.aocx` file, even if the dev PC doesn't have an FPGA installed.

TIP	If on Windows, you can download Intel's RTE for Windows, and in one of the folders is `OpenCL.lib`. Copy this library beside the source (beside Cargo.toml in above examples) when compiling so it links to the OpenCL headers.

7.7. OpenCL quickstart

http://www.drdobbs.com/parallel/a-gentle-introduction-to-opencl/231002854

7.8. Documentation for Intel FPGA SDK for OpenCL

Intel's official documentation, with more detailed instructions for installing Quartus, and running `aoc`.

Getting Started

Programming Guide

Custom Platform Toolkit

7.9. Useful Linux commands

Check if dynamic libraries are available for a given executable

```
$ ldd [executable]
```

List loaded kernel modules:

```
$ lsmod
```

8. Troubleshooting

Problem	Possible Cause	Possible Solution
No login prompt	• corrupt root file system • corrupt `zImage` • incorrect `socfpga.dtb`	remake the SD card with the provided `.img` and verify you get a login screen without any modifications
`aocl` command not found	Environment variables not set	run: `source ./init_opencl_17.1.sh` from the home directory
`quartus_cpf` command not found	Development PC environment variables not set correctly	See Environment Variable setup
`aoc` command not found	Development PC environment variables not set correctly	See Environment Variable setup
OpenCL host program freezes when run on DE10	`opencl.rbf` is not the correct version	Shutdown DE10, plug MicroSD card into PC and copy the `.rbf` file that corresponds to the OpenCL host program. It should be named `opencl.rbf` in the FAT32 partition. Alternatively, there could be something wrong with the BSP creating an invalid `.rbf` file. Double check the BSP has no errors in Quartus/Qsys.