

## AWS FPGA Starter's Guide (Rev 2)

### Introduction

This guide explains how to work with AWS FPGA Hardware/Software Development Kit (HDK/SDK) and deploy your FPGA application on AWS. Developing FPGA application in cloud flows like this.

- 1) Write your custom logic (CL) that goes into FPGA in SystemVerilog.
- 2) RTL Simulate your CL for debugging and validation.
- 3) Create Amazon FPGA image (AFI).
- 4) Write your host program in C/C++.
- 5) Remote into AWS FPGA instance and load AFI into FPGA.
- 6) Compile and run the C/C++ application.



Figure 1. Programming model

### Creating AWS Account

First, sign up for an AWS account with your UW email account at <https://aws.amazon.com>. You will have to enter a valid credit card info, but you will not be charged for anything (Uncle Bezos got you covered this time). Once the sign up is complete, go to <https://aws.amazon.com/education/awseducate> to apply for free AWS credit. You want to choose the option to have the free credit applied to the account you just created (Do not pick the “AWS Starter Account” option). Be sure to do this step first, because this process may take up to 2-3 days at most. You can check whether your free credit has been applied to AWS at Billing Dashboard:

<https://console.aws.amazon.com/billing/home#/>.

The screenshot shows the AWS Billing Dashboard with the 'Credits' tab selected. It includes a security check, a 'Redeem' button, and a table of redeemed credits.

Expiration Date	Credit Name	Credits Used	Credits Remaining	Applicable Products
2019-09-30	EDU_ENG_FY2017_Q3_MAN9_100USD	\$21.52	\$78.48	<a href="#">See complete list</a>

**Total Amount of Credits Remaining: \$78.48**

Custom Logic (CL) is the core of your FPGA application that actually does something inside FPGA. HDK provides the shell template (SH) which encapsulates CL and provides I/O interface to the host and peripherals.

## 1) Setting up HDK

- 1) Log into one of EE Linux machines where Vivado is installed (remotely or in person). Those machines are named 'linux-lab-001.ee.washington.edu' through 'linux-lab-040.ee.washington.edu'.

You need to have EE account. Follow this link, if you don't have one yet.

[https://www2.ee.washington.edu/computing/faq/new\\_account.html](https://www2.ee.washington.edu/computing/faq/new_account.html)

Once you have obtained EE account, follow this link that explains how you can remote into EE machines and setting up X11 forwarding for GUI.

<https://www2.ee.washington.edu/computing/faq/xming.html>

- 2) Run the following line. 'settings64.sh' script will allow you call 'vivado' from command line, and appending this line to '.bashrc' will make sure that this script is called whenever you open a bash terminal.

```
$ "source /homes/lab.apps/xilinx_vivado_v2017.1_sdx/SDx/2017.1.op/settings64.sh" >> ~/.bashrc
```

- 3) Type 'bash' to switch terminal.
- 4) Create a directory where you want to store your project and cd in there.
- 5) Clone git repository from aws-fpga. It should create a directory named "aws-fpga".

```
$ git clone https://github.com/aws/aws-fpga.git
```

- 6) cd into aws-fpga directory, and you will find hdk\_setup.sh file.

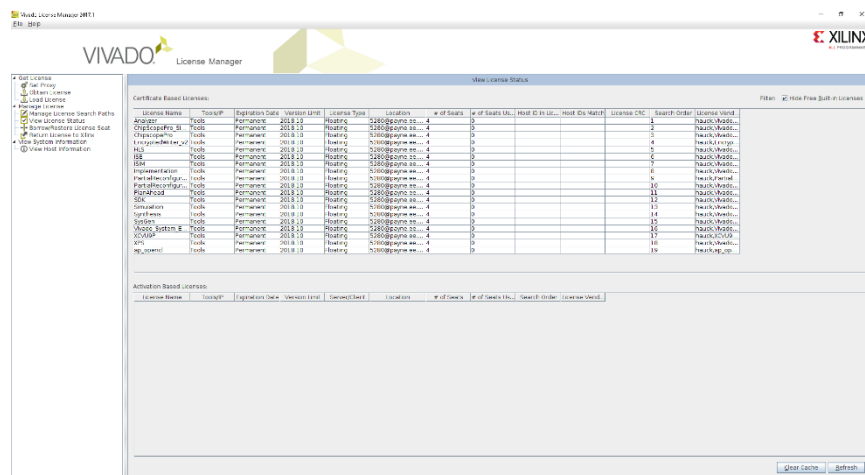
```
ERRATA.md      README.md      hdk      sdk
FAQs.md        RELEASE_NOTES.md  hdk_setup.sh  sdk_setup.sh
LICENSE.txt     SDAccel      sdaccel_setup.sh
[dcjung@linux-lab-019 aws-fpga]$
```

Figure 2. inside aws-fpga directory

- 7) Run `source hdk_setup.sh`.

You should be able to run vivado, and validate that you have required licenses. This will open up Vivado License Manager. Once vivado application is launched, go to Help->Manage License. Make sure you have following licenses.

vivado &



## 2) Hello World (Virtual LED/DIP switch)

Using the virtual LED and DIP switch is the most basic way for the host to communicate with FPGA. The host can pass some configuration or command via virtual DIP, and the FPGA can return the result or raise an alert to the host via virtual LED. It's called 'virtual', because we are remotely programming an FPGA board somewhere inside AWS datacenter, not a local FPGA board.

The example custom logics are located under 'aws-fpga/hdk/cl/examples/'. Those examples are provided by AWS, and you are free to take a look. Instead, we are going to use the 'cl\_led\_dip' custom logic in aws-fpga-starterpack.

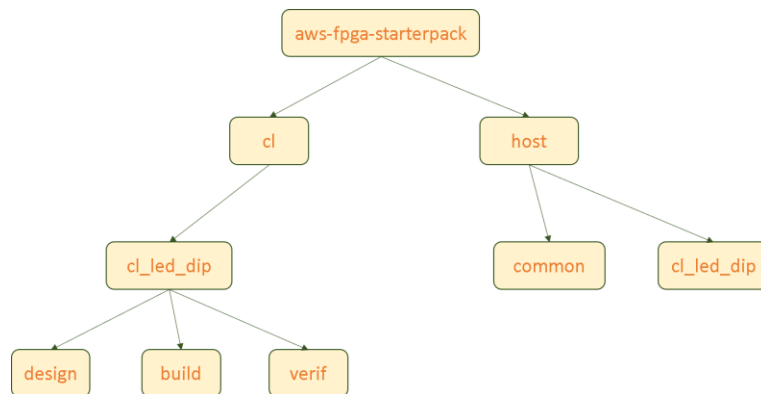


Figure 3. aws-fpga-starterpack directory structure

- 1) Clone aws-fpga-starterpack repository in your project directory.

```
git clone https://github.com/tommydcjung/aws-fpga-starterpack.git
```

- 2) Copy the 'cl/cl\_led\_dip' folder from aws-fpga-starterpack to 'aws-fpga/hdk/cl/examples/'.

```
README.md      cl_examples_list.md  cl_hello_world_ref_hlx  cl_led_dip      hello_world_hlx
cl_dram_dma    cl_hello_world        cl_hello_world_vhdl    cl_uram_example
cl_dram_dma_hlx cl_hello_world_hlx    cl_ipi_cdma_test_hlx   common
```

Figure 4. Inside aws-fpga/hdk/cl/examples/ directory

- 3) Inside the 'cl\_led\_dip' directory, you will find four directories.

```
[dcjung@linux-lab-019 cl_led_dip]$ ls
build design software verify
```

'build' will be used for creating AFI later, and 'verif' will be used for simulating CL. All the custom logic Verilog code should be written in design.

- 4) cd into 'design', and you will find 'cl\_led\_dip.sv' file. This is the top-level Verilog module for your CL. As mentioned before, CL is encapsulated within shell (SH) to abstract I/O for the developer. The input and output ports of this top-level module are defined in the Verilog header file called 'cl\_ports.vh', which is located in 'aws-fpga/hdk/common/shell\_stable/design/interfaces/'.

```
// Hello World virtual DIP/LED
logic[15:0] vdip_q;
always_ff @(posedge clk_main_a0 or negedge rst_main_n_sync)
    if (!rst_main_n_sync) begin
        vdip_q <= 16'b0;
    end
    else begin
        vdip_q <= sh_cl_status_vdip[15:0];
    end
end

assign cl_sh_status_vled = {vdip_q[3:0], vdip_q[7:4], vdip_q[11:8], vdip_q[15:12]};
```

Figure 5. core functionality of cl\_led\_dip

The ports for virtual DIP and LED are called “sh\_cl\_status\_vdip” and “cl\_sh\_status\_vled” respectively, and they are each 16-bit wide. In the code snippet above, the input “sh\_cl\_status\_vdip” flopped into the register called “vdip\_q” every clock cycle, and the “cl\_sh\_status\_vled” is assigned the reversed hex of “vdip\_q”.

### Validating and Debugging CL by RTL simulation

It is a good idea to simulate your custom logic before you build and create the AFI. This provides a chance to validate that your design works as you intended. If there is a bug in your logic, this can be an invaluable tool for debugging. We will write the simulation code in SystemVerilog.

- 1) Go to ‘verif/tests/’ directory. There is a file called ‘test\_led\_dip.sv’.

```
[dcjung@linux-lab-019 cl_led_dip]$ cd verific/
[dcjung@linux-lab-019 verific]$ ls
scripts  sim  tests
[dcjung@linux-lab-019 verific]$ cd tests/
[dcjung@linux-lab-019 tests]$ ls
test_led_dip.sv
```

- 2) Open ‘test\_led\_dip.sv’, and you will see that the test is setting the DIP switch to “0x0000”, and then to “0xbeef”. When you read the LED value, it should be “0xfeeb”. More details about the test bench can be found here: [https://github.com/aws/aws-fpga/blob/master/hdk/docs/RTL\\_Simulating\\_CL\\_Designs.md](https://github.com/aws/aws-fpga/blob/master/hdk/docs/RTL_Simulating_CL_Designs.md)

```

module test_led_dip();

    import tb_type_defines_pkg::*;

    logic [15:0] vdip_value;
    logic [15:0] vled_value;

    initial begin

        tb.power_up();

        // set vDIP to 0x0000.
        tb.set_virtual_dip_switch(.dip(16'h0000));
        vdip_value = tb.get_virtual_dip_switch();
        $display("[%t] vdip = 0x%x", $realtime, vdip_value);

        vled_value = tb.get_virtual_led();
        $display("[%t] vled = 0x%x", $realtime, vled_value);

        // set vDIP to 0xbeef.
        tb.set_virtual_dip_switch(.dip(16'hbeef));
        vdip_value = tb.get_virtual_dip_switch();
        $display("[%t] vdip = 0x%x", $realtime, vdip_value);
        #10ns; // wait 10ns

        vled_value = tb.get_virtual_led();
        $display("[%t] vled = 0x%x", $realtime, vled_value);

        if (vled_value == 16'hfeeb)
            $display("[%t] Test passed.", $realtime);
        else
            $display("[%t] Test failed.", $realtime);

        tb.kernel_reset();
        tb.power_down();

        $finish;
    end
endmodule
~
~

```

Figure 6. test\_led\_dip.sv

- 3) In order to run the test, cd into “verif/scripts”. There are three files of our interests.
  - a. Makefile – we are using this file to build and run the test.
  - b. top.vivado.f – if you add create a new module or include a new IP, make sure to include it in this file.
  - c. waves.tcl – during the simulation you can capture waveforms of some of the signals in your custom logic. The first line of ‘wave.tcl’ is “add\_wave /card/fpga/CL”. This statement will tell the simulation to capture all the signal in the top-level module (in this case ‘cl\_led\_dip’). Note that it will only signals at the top-level, but not the signals at a lower level. For example, if there was an instance named “MY\_MODULE”, you will have to add ‘add\_wave /card/fpga/CL/MY\_MODULE’ to capture the signals inside “MY\_MODULE” instance. You can also use “add\_wave -recursive /card/fpga/CL” to capture all the signals at lower levels recursively, but it could result in a very large waveform output, so use with discretion.
- 4) While you are in “verif/scripts”, run “make TEST=test\_led\_dip”. The parameter after “TEST=” should match the name of the test file under “verif/tests” without the file extension (.sv). The test script will compile and run the test program, and it should print the output like this in the end.

```

[          5050000] vdip = 0x0000
[          5050000] vled = 0x0000
[          5050000] vdip = 0xbeef
[          5060000] vled = 0xfeeb
[          5060000] Test passed.
$finish called at time : 5160 ns : File "/home/dcjung/tutorial/aws-fpga/hdk/cl/examples/c
l_led_dip/verif/tests/test_led_dip.sv" Line 37
## quit
INFO: [Common 17-206] Exiting xsim at Thu Nov 16 06:57:21 2017...
bash-4.2$

```

Figure 7. test\_led\_dip test result

- Now, let us view the waveform captured during the simulation. If you go back to “verif” directory, you will notice that “verif/sim” directory is created. The simulation results are stored under this directory, with the name of the test you just ran. cd into “verif/sim/test\_led\_dip”. There is a file called “tb.wdb”. That is the waveform database file. Create a text file called “open\_waves.tcl” inside this directory, and add the following two lines and save.

```

current_filesset
open_wave_database tb.wdb

```

Now, run `vivado -source open_waves.tcl`

- Under ‘Scope’ tab, you will see the hierarchy of modules. Click down to “/tb/card/fpga/CL”. In ‘Objects’ panel, you will see the signals defined under each hierarchy. Right-click ‘clk\_main\_a0’, and click “Add to Wave Window”. Also, add ‘rst\_main\_n’, ‘sh\_cl\_status\_vdip[15:0]’ and ‘cl\_sh\_status\_vled[15:0]’. Drag and click the waveform window to zoom into where the interesting events are happening.

You should see that virtual DIP starts out as ‘0x0000’ and becomes ‘0xbeef’. Then, one clock cycle after, virtual LED value becomes ‘0xfeeb’.

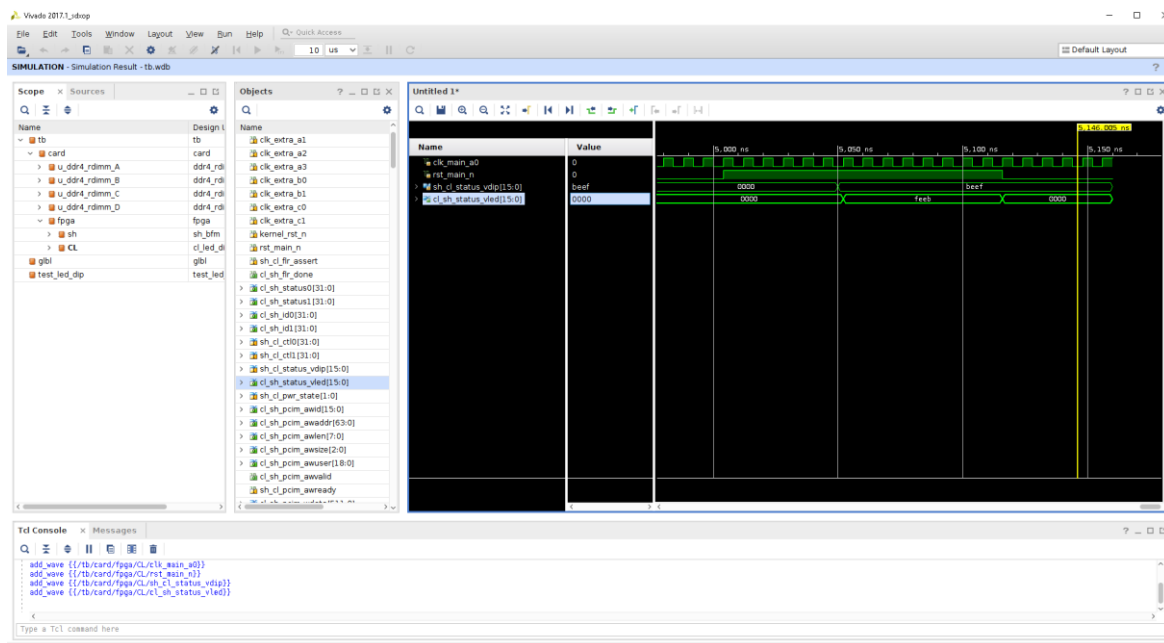


Figure 8. Waveform of cl\_led\_dip.

## Creating Amazon FPGA Image

Now that we have validated the functionality of `cl_led_dip` through RTL simulation, it is time to create Amazon FPGA Image (AFI). First, we have to build a tarball from our CL design. Then, we have to run various AWS CLI commands to create AFI.

### 1. Creating tarball from CL design

- 1) cd into "aws-fpga/hdk/cl/examples/cl\_led\_dip"
- 2) Set the environment variable. When you type `echo $CL_DIR`, it should print out the path to your `cl_led_dip` directory.

```
$ export CL_DIR=$(pwd)
```

- 3) cd into "cl\_led\_dip/build/scripts" and run `./aws_build_dcp_from_cl.tcl -foreground`
- 4) The build process will take a long time even for the simplest application (~2-3 hours), so you can either move onto "Writing C/C++ Host Application" or go out for a walk and grab some coffee.
- 5) If the build was successful, cd into "cl\_led\_dip/build/checkpoints/to\_aws/". You shall find a tarball file generated with timestamp as prefix (e.g. `17_11_16-183421.Developer_CL.tar`).

```
[dcjung@linuxsrv01 to_aws]$ pwd
/home/dcjung/tutorial/aws-fpga/hdk/cl/examples/cl_led_dip/build/checkpoints/to_aws
[dcjung@linuxsrv01 to_aws]$ ls
17_11_16-183421.Developer_CL.tar  17_11_16-183421.SH_CL_routed.dcp  17_11_16-183421.manifest.txt
[dcjung@linuxsrv01 to_aws]$
```

### 2. Configuring AWS CLI / Creating AFI

- 1) We are going to use AWS Command Line Interface (CLI) to create AFI. If you are working from your own computer you will have to install AWS CLI. <https://aws.amazon.com/cli/>  
If you are working with EE dept machines, AWS CLI should have been installed.
- 2) We have to configure the AWS CLI so that it works with your AWS account credential. In order to do that we need to create access key pair. [https://console.aws.amazon.com/iam/home?#/security\\_credentials](https://console.aws.amazon.com/iam/home?#/security_credentials)  
Click Access keys-> Create New Access Key. It will give you "Access Key ID" and "Secret Access Key". Save those two strings somewhere secure. Do not share the secret access key with others.

**aws** Services ▾ Resource Groups ▾ ★ 🔔 dcjung ▾ Global ▾ Support ▾

Search IAM

- Dashboard
- Groups
- Users
- Roles
- Policies
- Identity providers
- Account settings
- Credential report
- Encryption keys

### Your Security Credentials

Use this page to manage the credentials for your AWS account. To manage credentials for AWS Identity and Access Management (IAM) users, use the [IAM Console](#).

To learn more about the types of AWS credentials and how they're used, see [AWS Security Credentials](#) in AWS General Reference.

- + Password
- + Multi-factor authentication (MFA)
- Access keys (access key ID and secret access key)

You use access keys to sign programmatic requests to AWS services. To learn how to sign requests using your access keys, see the [signing documentation](#). For your protection, store your access keys securely and do not share them. In addition, AWS recommends that you rotate your access keys every 90 days.

Note: You can have a maximum of two access keys (active or inactive) at a time.

Created	Deleted	Access Key ID	Last Used	Last Used Region	Last Used Service	Status	Actions
Oct 15th 2017		[REDACTED]	2017-10-15T10:10:10Z	us-west-2	ec2	Active	<a href="#">Make Inactive</a>   <a href="#">Delete</a>
Oct 21st 2017	Nov 16th 2017	[REDACTED]	N/A	N/A	N/A	Deleted	
Oct 6th 2017	Oct 21st 2017	[REDACTED]	N/A	N/A	N/A	Deleted	

[Create New Access Key](#)

- 3) Now back to the terminal. Run the following command. Enter the access key ID and secret access key. Default region name should be 'us-west-2' (Oregon). Just hit enter for default output format.

```
$ aws configure
```

```
[dcjung@linuxsrv01 ~]$ aws configure
AWS Access Key ID [None]: 
AWS Secret Access Key [None]: 
Default region name [None]: us-west-2
Default output format [None]: 
[dcjung@linuxsrv01 ~]$
```

- 4) AWS S3 is a cloud-based file storage service. We have to upload our tarball to S3, which then will be picked up by AFI generation service to be processed.
- 5) Create a S3 bucket (analogous to a file directory)

```
$ aws s3 mb s3://my-fpga-cl --region us-west-2
```

- 6) Upload the tarball to the bucket just created.

```
$ aws s3 cp ./17_11_16-183421.Developer_CL.tar s3://my-fpga-cl/
```

- 7) Create a S3 bucket for AFI generation log.

```
$ aws s3 mb s3://my-fpga-log --region us-west-2
```

You should be able to verify that bucket has been created and the tarball uploaded as expected here on the AWS management portal: <https://s3.console.aws.amazon.com/s3/home?region=us-west-2#>

- 8) Start AFI generation.

```
$ aws ec2 create-fpga-image --region us-west-2 --input-storage-location
Bucket=my-fpga-cl,Key=17_11_16-183421.Developer_CL.tar --logs-storage-
location Bucket=my-fpga-log,Key=
```

Calling this command will return json output like this.

```
[dcjung@linuxsrv01 ~]$ aws ec2 create-fpga-image --region us-west-2 --input-storage-location Bucket=my-fpga-cl,Key=17_11_16-183421.Developer_CL.tar --logs-storage-location Bucket=my-fpga-log,Key=/
{
  "FpgaImageId": "afi-02c0bce69ef7c9902",
  "FpgaImageGlobalId": "agfi-0edfc64f852051323"
}
```

You will need to save these IDs somewhere to load AFI to FPGA later.

- 9) Check if AFI generation has completed.

```
aws ec2 describe-fpga-images --fpga-image-ids <your afi id>
```

Once it's completed `FpgaImages.State.Code` will eventually become "available". Only then you are able to load this AFI to FPGA. It takes about 20~30 minutes.



```
[dcjung@linuxsrv01 to_aws]$ aws ec2 describe-fpga-images --fpga-image-ids afi-02c0bce69ef7c9902
{
  "FpgaImages": [
    {
      "UpdateTime": "2017-11-16T21:50:52.000Z",
      "FpgaImageGlobalId": "agfi-0edfc64f852051323",
      "State": {
        "Code": "pending"
      },
      "FpgaImageId": "afi-02c0bce69ef7c9902",
      "OwnerId": "240095434820",
      "CreateTime": "2017-11-16T21:50:52.000Z"
    }
  ]
}
```

## Writing C/C++ Host Application

Developing host application has to be done on AWS virtual machine that runs FPGA Developer AMI (Amazon Machine Images). AWS provides C library for interacting with FPGA. This interaction is done through PCI Express Bus. Shell (SH) that encapsulates our CL handles the communication. This communication is generally divided into two physical function (PF): MgmtPF and AppPF. MgmtPF deals with the management functionality of FPGA: loading/unloading FPGA image, peeking/poking virtual DIP switch and LED. AppPF deals the functionalities that are more specific to your CL, such as sending and receiving stream of data over DMA.

Host application is separate from CL. In the starterpack repo, there is a directory called '/host'. It contains an example host application for cl\_led\_dip. '/host/common' contains C++ header and class files for wrapping C libraries provided in aws-fpga repo (Refer to aws-fpga/sdk/userspace/include'). Make sure to call 'source sdk\_setup.sh' before you compile the example host applications. Makefile inside host/cl\_led\_dip defines how the host application is compiled.

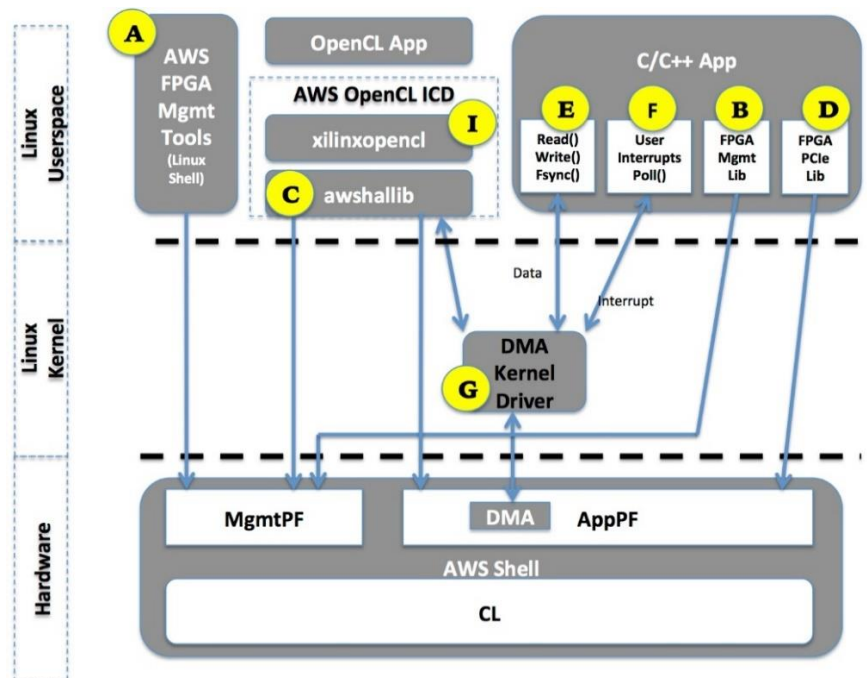
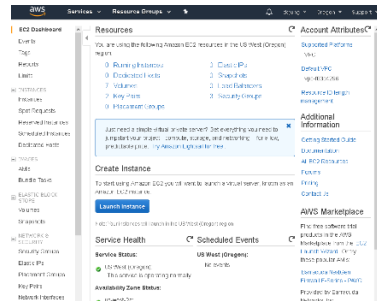


Figure 9. There are many ways to interact with FPGA.

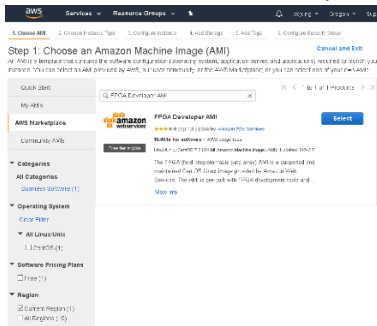
## 1. Setting up FPGA Development AMI VM

- 1) Go to <https://us-west-2.console.aws.amazon.com/ec2/v2/home?region=us-west-2>

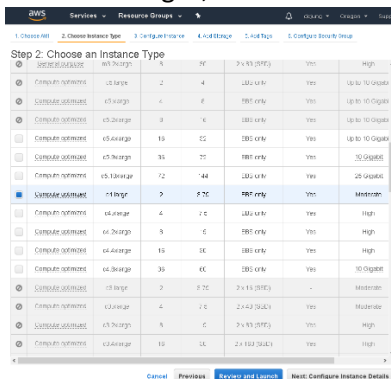
Click “Launch Instance”.



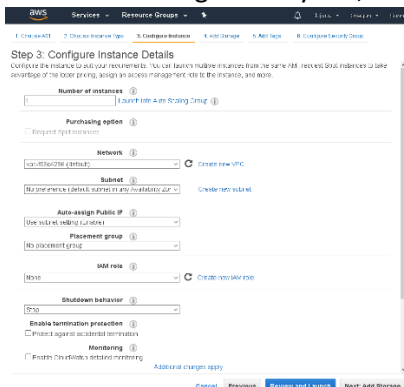
- 2) Click “AWS Marketplace” on the left tab, and search for “FPGA Developer AMI”. Click “Select”. Click “Continue” when it shows you a menu.



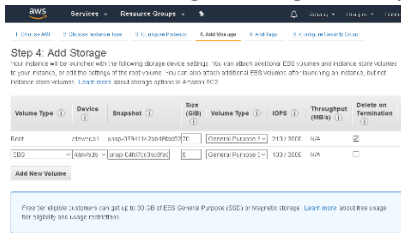
- 3) Select “c4.large”, and click “Next”.



- 4) Leave the settings as they are, and click “Next”.



## 5) Leave the storage settings as they are, and click “Next”



Volume Type	Device	Snapshot	Size (GB)	Volume Type	IOPS	Throughput (MB/s)	Delete on Termination
gp2	m1.xlarge	snap-7f81162a:rootfs01-17	100	General Purpose (SSD)	313 / 3000	N/A	<input checked="" type="checkbox"/>
gp2	ephemeral	snap-6407c3a0:eph	8	General Purpose (SSD)	100 / 3000	N/A	<input type="checkbox"/>

[Add New Volume](#)

Free tier eligible customers can get up to 30 GB of EBS General Purpose (SSD) or Magnetic storage. [Learn more](#) about the usage for eligibility and usage restrictions.

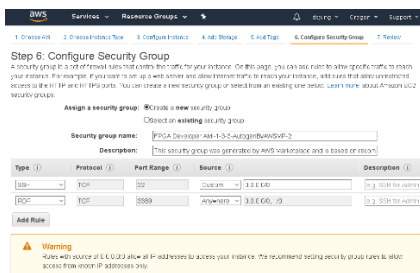
## 6) Click “Next”



Key	Value
Name	m1.xlarge

[Add Tag](#)

## 7) Click “Add Rule”. Select type “RDP” for remote desktop. Select source “Anywhere”. Click “Review and Launch”

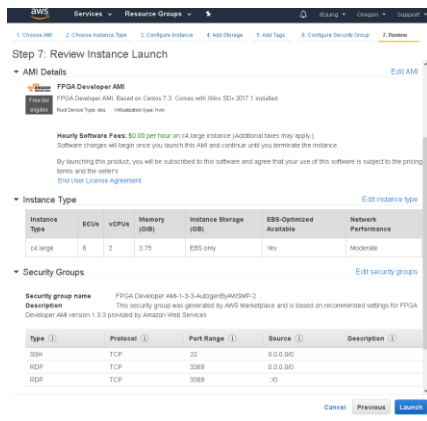


Type	Protocol	Port Range	Source	Description
RDP	TCP	3389	Anywhere	3.3.3.3/0
RDP	TCP	3389	Anywhere	3.3.3.3/0

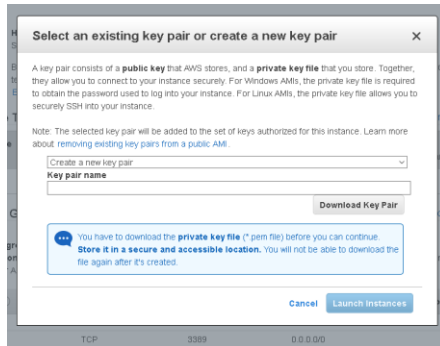
[Add Rule](#)

**Warning**  
Rules with source of 0.0.0.0 allow all IP addresses to access your instance. The recommended security group rules to allow access from most IP addresses only.

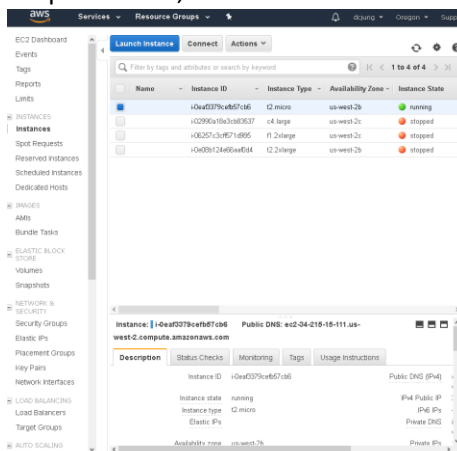
## 8) Click “Launch”



- 9) Select “Create a new key pair”. Download your key pair (.pem). Click “Launch Instances”.



- 10) You can see that your instance is up and running. Make sure that the instance state is “Stopped”, when you are not using. You will get billed for just leaving the machine running idly. You can click “Actions” dropdown bar, and select instance state.



## 2. Connecting to VM

- 1) You can follow the instruction here to connect to VM on Windows using PuTTY here: [https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/putty.html?icmpid=docs\\_ec2\\_console](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/putty.html?icmpid=docs_ec2_console)
- 2) You can also connect using ssh: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AccessingInstancesLinux.html>

Note: your login will be ‘centos’

## 3. Setting up SDK

- 1) Once you are logged into the VM, we need to set up the SDK. You want to store all your files in /home/centos/src/project\_data, because the files stored there will be persist even after the session ends.

```
cd ~/src/project_data
```

- 2) Clone the aws-fpga repository.

```
git clone https://github.com/aws/aws-fpga.git
```

- 3) cd into aws-fpga, and run

```
source sdk_setup.sh
```

- 4) You should see message like this.

```
AWS FPGA: Copying Amazon FPGA Image (AFI) Management Tools to /usr/bin
AWS FPGA: Installing shared library to /usr/local/lib64
          libfpga_mgmt.so.1 (libc6,x86_64) => /usr/local/lib64/libfpga_mgmt.so.1
AWS FPGA: Done with Amazon FPGA Image (AFI) Management Tools install.
Done with SDK install.
Done with AWS SDK setup.
[centos@ip-172-31-0-242 aws-fpga]$
```

#### 4. Cloning aws-fpga-starterpack repository

I have created C++ wrapper around C library in the SDK.

- 1) cd back into ~/src/project\_data
- 2) 

```
git clone https://github.com/tommydcjung/aws-fpga-starterpack.git
```
- 3) Inside the repository, there is aws-fpga-starterpack/host/common directory where C++ wrapper headers/classes are located.
- 4) You can go into aws-fpga-starterpack/host/cl\_led\_dip directory, where the source code for the host application for cl\_led\_dip is located. It also contains Makefile that builds the app.

```

#include <iostream>
#include "fabricmanager.h"
#include "fpga_mgmt.h"
using namespace std;

int main(int argc, char ** argv)
{
    auto fabricManager = new FabricManager();
    fpga_mgmt_image_info_t* info = 0;

    try
    {
        // get fpga image info.
        info = fabricManager->getImageInfo(0);
        cout << "FPGA Image Info:" << endl;
        printf("Vendor ID: 0x%x\r\n", info->spec.map[FPGA_APP_PF].vendor_id);
        printf("Device ID: 0x%x\r\n", info->spec.map[FPGA_APP_PF].device_id);

        // set virtual DIP to 0x0000
        fabricManager->setvDIP(0, 0x0000);
        cout << "Setting vDIP to 0x0000" << endl;

        // read virtual LED value
        uint16_t vLED = fabricManager->getvLED(0);
        printf("vLED = 0x%x\r\n", vLED);

        // set virtual DIP to 0xbeef
        fabricManager->setvDIP(0, 0xbeef);
        cout << "Setting vDIP to 0xbeef" << endl;

        // read virtual LED value
        vLED = fabricManager->getvLED(0);
        printf("vLED = 0x%x\r\n", vLED);
    }
    catch(exception& e)
    {
        cout << e.what() << endl;
    }

    // make sure to free dynamically allocated memory.
    free(info);
    delete fabricManager;

    return 0;
}

```

Figure 10. app.cpp

- 5) Run **make**. That should compile the program and create a file called **app**.

```

[centos@ip-172-31-0-242 c1_led_dip]$ ls
app.cpp Makefile
[centos@ip-172-31-0-242 c1_led_dip]$ make
g++ -o app ./app.cpp ../common/*.h ../common/*.hpp ../common/*.cpp -lfpga_mgmt -I/home/centos/src/project_data/aws-fpga/sdk/userspace/include/ -I../common/ -lrt -lpthread -std=c++0x
[centos@ip-172-31-0-242 c1_led_dip]$ ls
app app.cpp Makefile
[centos@ip-172-31-0-242 c1_led_dip]$

```

- 6) From here, you can develop your own FPGA host application in C++. You should create your own github repository and start from there.

### Loading AFI in AWS FPGA instance

Finally, we have CL and host application written. We are ready to run our application on the real FPGA instance.

1. Setting up F1 instance

Amazon Elastic Compute Cloud (EC2) now provides F1 instances, which are equipped with Xilinx FPGA (VU9P). The details of this FPGA can be found under 'doc/' in aws-fpga-starterpack repository. (ds890-ultrascale-overview.pdf).

## F1 Instance Details

Instance Type	FPGA Cards	vCPUs	Instance Memory (GiB)	SSD Storage (GB)	Enhanced Networking	EBS Optimized
f1.2xlarge	1	8	122	470	Yes	Yes
f1.16xlarge	8	64	976	4 x 940	Yes	Yes

For F1.16xlarge instances, the dedicated PCI-e fabric lets the FPGAs share the same memory space and communicate with each other across the fabric at up to 12 GBps in each direction.

1. The steps for setting up f1 instance is nearly identical to setting up FPGA Development AMI VM as above except we are picking f1.2xlarge as instance type. We will still be using FPGA Development AMI as the machine image.

## Running C/C++ host application

1. Setting up aws-fpga and aws-fpga-starterpack repo
  - 1) Once you are logged into f1.2xlarge instance, do the same as done above. Go to ~/src/projectdata. Git clone both aws-fpga, and aws-fpga-starterpack repositories.
  - 2) Run `source sdk_setup.sh` from aws-fpga directory.
  - 3) Run `make` to build the host application from aws-fpga-starterpack/host/cl\_led\_dip.
  - 4) Clear the AFI slot 0.

```
$ sudo fpga-clear-local-image -S 0
```

```
[centos@ip-172-31-0-242 cl_led_dip]$ sudo fpga-clear-local-image -S 0
AFI          0          none          cleared          1          ok          0          0x071417d3
AFIDEVICE    0          0x1d0f          0x1042          0000:00:1d.0
[centos@ip-172-31-0-242 cl_led_dip]$
```

- 5) Load the AFI you created in the section “[Creating Amazon FPGA Image](#)” section above. Use the FPGA image global ID that started with ‘agfi-’ that you saved somewhere .

```
$ sudo fpga-load-local-image -S 0 -I {{ Your AFGI ID }}
```

```
[centos@ip-172-31-0-242 cl_led_dip]$ sudo fpga-load-local-image -S 0 -I agfi-0edfc64f852051323
AFI          0          agfi-0edfc64f852051323 loaded          0          ok          0          0x071417d3
AFIDEVICE    0          0x1d0f          0xf000          0000:00:1d.0
[centos@ip-172-31-0-242 cl_led_dip]$
```

- 6) Run the host application.

```
$ sudo ./app
```

```

[centos@ip-172-31-0-242 ci_led_dip]$ pwd
/home/centos/src/project_data/aws-fpga-starterpack/host/ci_led_dip
[centos@ip-172-31-0-242 ci_led_dip]$ ls
app.cpp  Makefile
[centos@ip-172-31-0-242 ci_led_dip]$ make
g++ -o app ./app.cpp ../common/*.h ../common/*.hpp ../common/*.cpp -lfpga_mgmt -I/home/centos/src/project_data/aws-fp
ga/sdk/userspace/include/ -I../common/ -lrt -lpthread -std=c++0x
[centos@ip-172-31-0-242 ci_led_dip]$ ls
app  app.cpp  Makefile
[centos@ip-172-31-0-242 ci_led_dip]$ sudo ./app
FPGA Image Info:
Vendor ID: 0x1d0f
Device ID: 0xf000
setting vDIP to 0x0000
vLED = 0x0
setting vDIP to 0xbeef
vLED = 0xfeeb
[centos@ip-172-31-0-242 ci_led_dip]$ █

```

Figure 11. It works.

- 7) Make sure that your instances are turned off when you are done.

### Useful Links

<https://github.com/aws/aws-fpga>

<https://forums.aws.amazon.com/forum.jspa?forumID=243&start=0>

<https://github.com/tommydcjung/aws-fpga-starterpack>

<https://github.com/>

<https://aws.amazon.com/console/>

<https://aws.amazon.com/education/awsseduate/>

<https://www2.ee.washington.edu/computing/faq/index.html>