

AWS FPGA Starter's Guide (Rev 1)

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#/.](https://console.aws.amazon.com/billing/home#/)

The screenshot shows the AWS Billing Dashboard 'Credits' page. It includes a sidebar with navigation links, a main content area with a security check and 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	See complete list

Total Amount of Credits Remaining: \$78.48

Writing Custom Logic

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

Once you have obtained EE account, follow this link that explains how you can remote into EE machines.

<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`.

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.

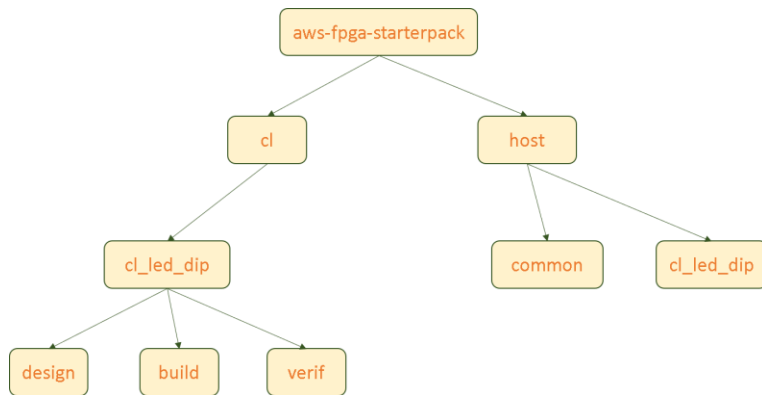


Figure 3. aws-fpga-starterpack directory structure

- 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
[dcjung@linux-lab-019 examples]$

```

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 verif
```

'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 verif/  
[dcjung@linux-lab-019 verif]$ ls  
scripts  sim  tests  
[dcjung@linux-lab-019 verif]$ 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

```
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

- 5) 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`

- 6) 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’.

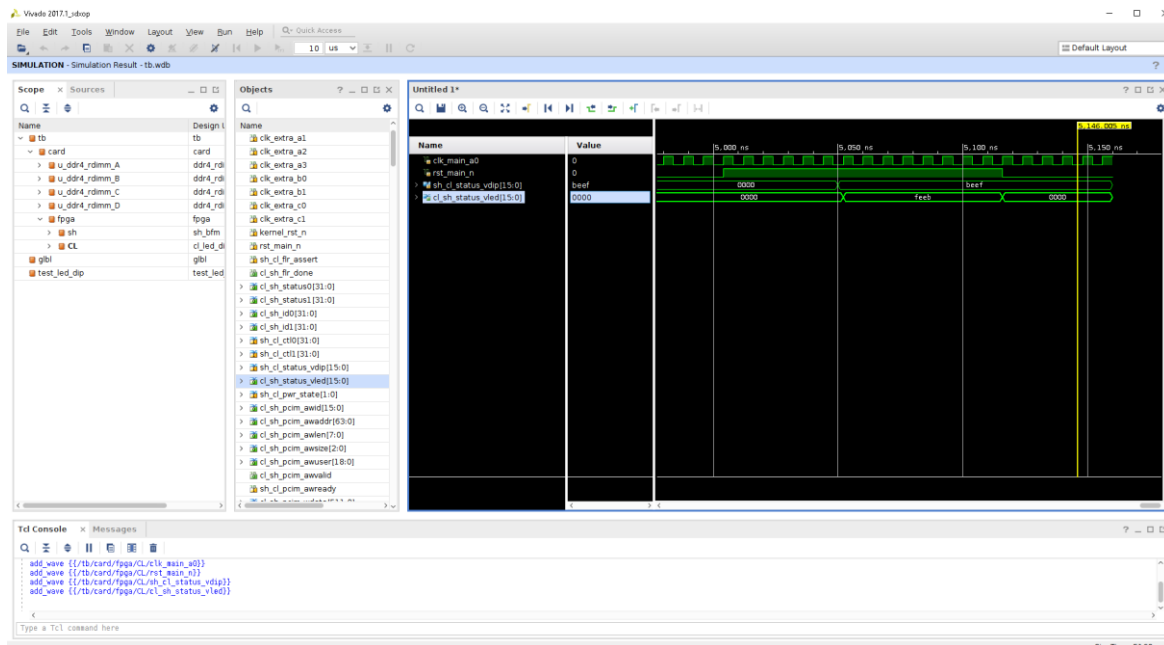


Figure 8. Waveform of `cl_led_dip`.

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_credential 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.

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	Make Inactive Delete
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@linuxserv01 ~]$ aws configure
AWS Access Key ID [None]: [REDACTED]
AWS Secret Access Key [None]: [REDACTED]
Default region name [None]: us-west-2
Default output format [None]:
[dcjung@linuxserv01 ~]$
```

- 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 to_aws]$ 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. 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.

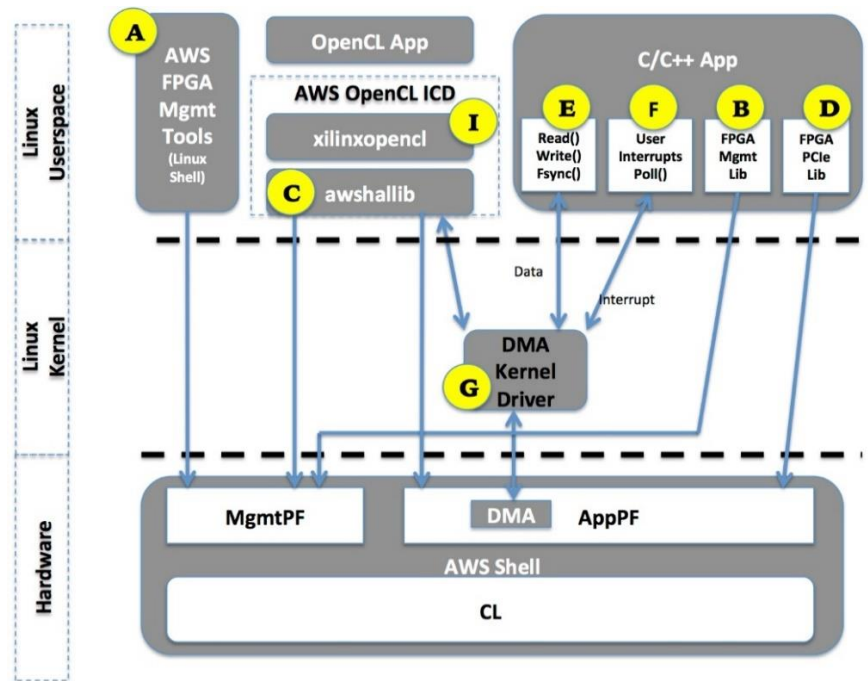
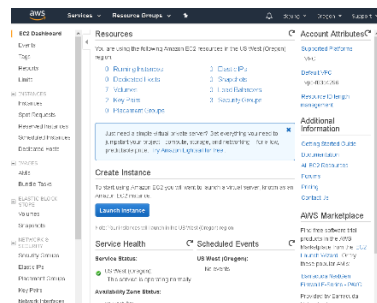


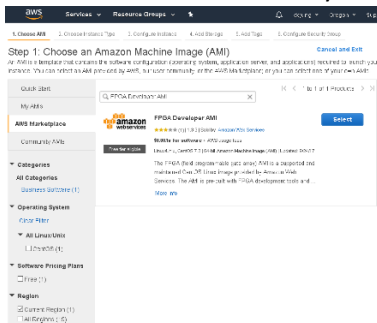
Figure 9. There are many ways to interact with FPGA.

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”.

Step 2: Choose an Instance Type

Compare the instance to help you recommend. You can filter multiple instances from the same AMI, review EC2 instance to take advantage of the lower pricing, design an access management role to the instance, and more.

Instance Type	On-Demand	Reserved	Spot	Memory	Storage
Compute optimized	2	4	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	4	8	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	8	16	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	16	32	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	32	64	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	64	128	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	128	256	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	256	512	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	512	1024	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	1024	2048	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	2048	4096	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	4096	8192	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	8192	16384	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	16384	32768	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	32768	65536	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	65536	131072	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	131072	262144	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	262144	524288	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	524288	1048576	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	1048576	2097152	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	2097152	4194304	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	4194304	8388608	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	8388608	16777216	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	16777216	33554432	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	33554432	67108864	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	67108864	134217728	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	134217728	268435456	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	268435456	536870912	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	536870912	1073741824	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	1073741824	2147483648	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	2147483648	4294967296	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	4294967296	8589934592	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	8589934592	17179869184	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	17179869184	34359738368	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	34359738368	68719476736	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	68719476736	137438953472	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	137438953472	274877906944	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	274877906944	549755813888	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	549755813888	1099511627776	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	1099511627776	2199023255552	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	2199023255552	4398046511104	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	4398046511104	8796093022208	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	8796093022208	17592186044416	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	17592186044416	35184372088832	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	35184372088832	70368744177664	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	70368744177664	140737488355328	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	140737488355328	281474976710656	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	281474976710656	562949953421312	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	562949953421312	1125899906842624	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	1125899906842624	2251799813685248	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	2251799813685248	4503599627370496	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	4503599627370496	9007199254740992	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	9007199254740992	18014398509481984	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	18014398509481984	36028797018963968	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	36028797018963968	72057594037927936	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	72057594037927936	144115188075855872	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	144115188075855872	288230376151711744	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	288230376151711744	576460752303423488	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	576460752303423488	1152921504606846976	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	1152921504606846976	2305843009213693952	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	2305843009213693952	4611686018427387904	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	4611686018427387904	9223372036854775808	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	9223372036854775808	18446744073709551616	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	18446744073709551616	36893488147419103232	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	36893488147419103232	73786976294838206464	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	73786976294838206464	147573952589676412928	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	147573952589676412928	295147905179352825856	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	295147905179352825856	590295810358705651712	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	590295810358705651712	1180591620717411303424	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	1180591620717411303424	2361183241434822606848	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	2361183241434822606848	4722366482869645213696	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	4722366482869645213696	9444732965739290427392	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	9444732965739290427392	18889465931478580854784	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	18889465931478580854784	37778931862957161709568	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	37778931862957161709568	75557863725914323419136	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	75557863725914323419136	151115727451828646838272	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	151115727451828646838272	302231454903657293676544	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	302231454903657293676544	604462909807314587353088	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	604462909807314587353088	1208925819614629174706176	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	1208925819614629174706176	2417851639229258349412352	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	2417851639229258349412352	4835703278458516698824704	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	4835703278458516698824704	9671406556917033397649408	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	9671406556917033397649408	19342813113834066795298816	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	19342813113834066795298816	38685626227668133590597632	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	38685626227668133590597632	77371252455336267181195264	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	77371252455336267181195264	154742504910672534362390528	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	154742504910672534362390528	309485009821345068724781056	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	309485009821345068724781056	618970019642690137449562112	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	618970019642690137449562112	1237940039285380274899124224	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	1237940039285380274899124224	2475880078570760549798248448	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	2475880078570760549798248448	4951760157141521099596496896	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	4951760157141521099596496896	9903520314283042199192993792	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	9903520314283042199192993792	19807040628566084398385987584	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	19807040628566084398385987584	39614081257132168796771975168	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	39614081257132168796771975168	79228162514264337593543950336	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	79228162514264337593543950336	158456325028528675187087900672	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	158456325028528675187087900672	316912650057057350374175801344	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	316912650057057350374175801344	633825300114114700748351602688	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	633825300114114700748351602688	1267650600228229401496703205376	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	1267650600228229401496703205376	2535301200456458802993406410752	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	2535301200456458802993406410752	5070602400912917605986812821504	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	5070602400912917605986812821504	10141204801825835211973625643008	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	10141204801825835211973625643008	20282409603651670423947251286016	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	20282409603651670423947251286016	40564819207303340847894502572032	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	40564819207303340847894502572032	81129638414606681695789005144064	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	81129638414606681695789005144064	162259276829213363391578010288128	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	162259276829213363391578010288128	324518553658426726783156020576256	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	324518553658426726783156020576256	649037107316853453566312041152512	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	649037107316853453566312041152512	1298074214633706907132624082305024	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	1298074214633706907132624082305024	2596148429267413814265248164610048	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	2596148429267413814265248164610048	5192296858534827628530496329220096	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	5192296858534827628530496329220096	10384593717069655257060992658440192	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	10384593717069655257060992658440192	20769187434139310514121985316880384	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	20769187434139310514121985316880384	41538374868278621028243970633760768	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	41538374868278621028243970633760768	83076749736557242056487941267521536	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	83076749736557242056487941267521536	166153499473114484112975882535043072	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	166153499473114484112975882535043072	332306998946228968225951765070086144	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	332306998946228968225951765070086144	664613997892457936451903530140172288	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	664613997892457936451903530140172288	1329227995784915872903807060280344576	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	1329227995784915872903807060280344576	2658455991569831745807614120560689152	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	2658455991569831745807614120560689152	5316911983139663491615228241121378304	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	5316911983139663491615228241121378304	10633823966279326983230456482242756608	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	10633823966279326983230456482242756608	21267647932558653966460912964485513216	Up to 10 Cpus	Yes	Up to 10 Gbps
Compute optimized	21267647932558653966460912964485513216	425352			

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

Type	Protocol	Port Range	Source	Description
SSH	TCP	22	Custom	SSH for admins
RDP	TCP	3389	Anywhere	RDP for admins

- 8) Click “Launch”

Instance Type	EC2	vCPUs	Memory (GB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance
c4.large	8	2	3.75	EBS only	Yes	Moderate

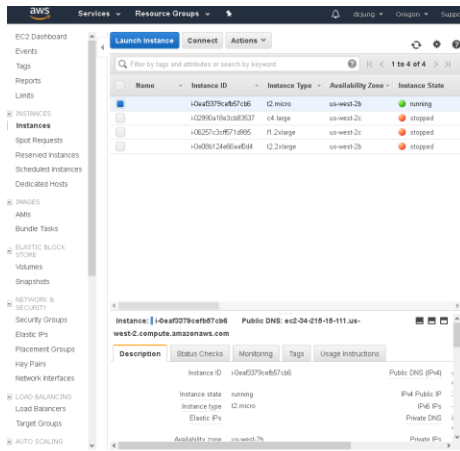
- 9) Select “Create a new key pair”. Download your key pair (.pem). Click “Launch Instances”.

Create a new key pair

Key pair name

Download Key Pair

- 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
- 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 cl_led_dip]$ ls
app.cpp Makefile
[centos@ip-172-31-0-242 cl_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 cl_led_dip]$ ls
app app.cpp Makefile
[centos@ip-172-31-0-242 cl_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/awseducate/>

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