**Hand on experience - Intel DevCloud**

**Goal:**

In this Lab, you will learn how to connect to DevCloud, and how to access to FPGA hosted in cloud, how to run on FPGA, and how to analyze the performance on FPGA.

First, we will connect to Intel DevCloud using Jupyter. Then you should upload the labs material on Jupyter. The first task is to compile AES dpc++ code for running on the host and validate its functionality. Then, we perform an emulation for FPGA. The AES code for the emulation or running on FPGA has extra features that we explore. After emulation, it is time to synthesize for FPGA but as it is a time-consuming task, we provide the bitstream file for you. After that, we analyze the performance and the interaction between CPU and FPGA using Intel VTune. Finally, we explore some features of Intel Advisor.

**Content of manual:**

1. How to get access to DevCloud
2. How to work with Jupyter notebook
3. Compile with DPCPP compiler and Emulation flow
4. Performance Analysis

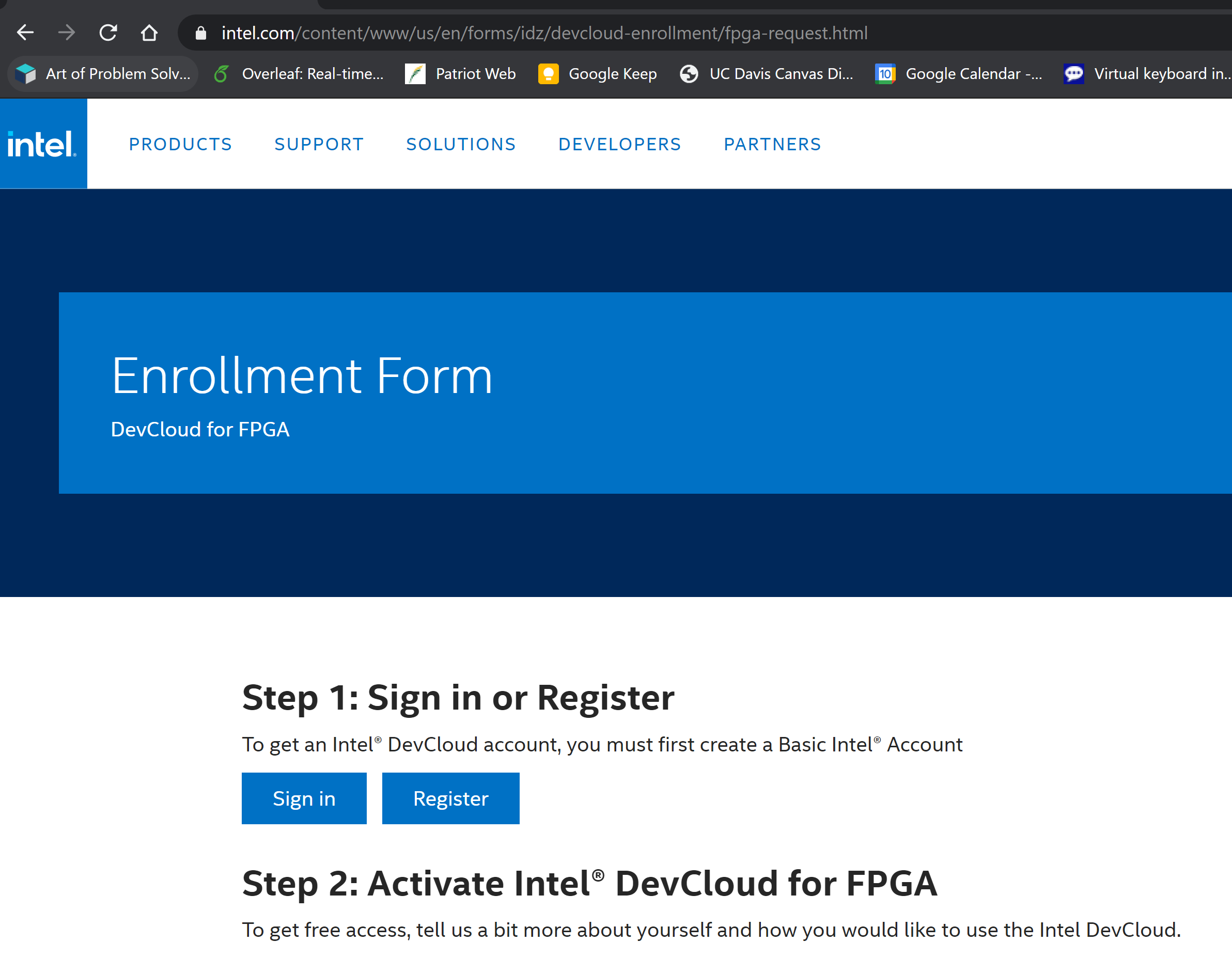
**P.S.: Any sentence with the red color, is the exact task that we are going to run on devcloud. The rest are just general description! Commands are in purple color.**

1. **Who can use Intel DevCloud?**

Developers, data scientists, professors, students, start-ups and others can request access for full access to the latest Intel CPUs, GPUs, and FPGAs, Intel oneAPI Toolkits, and the new programming language, Data Parallel C++ (DPC++). Access is free for 120 days with the possibility of an extension.

Check this link to register and request access:

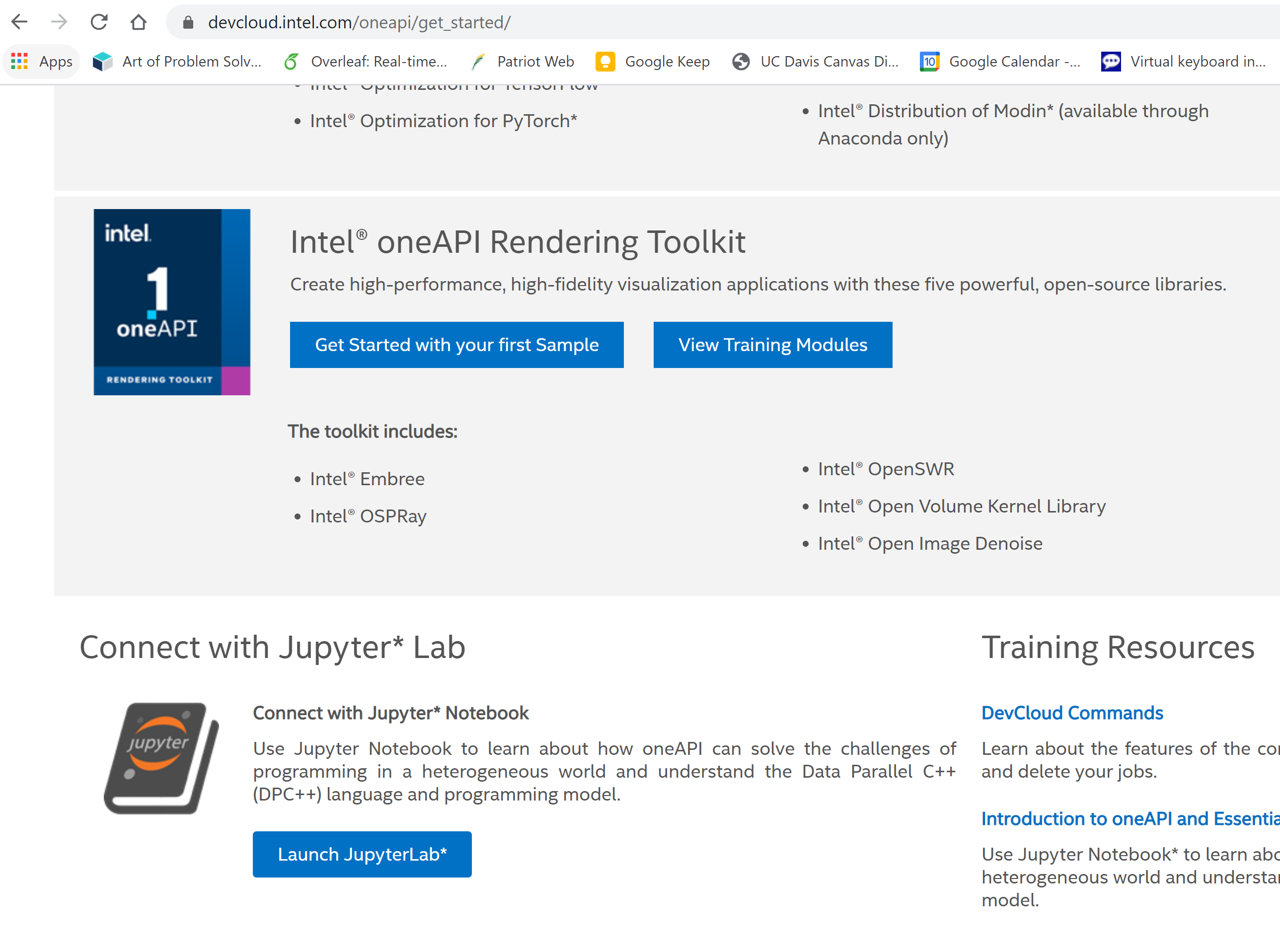
<https://www.intel.com/content/www/us/en/forms/idz/devcloud-enrollment/fpga-request.html>



After sign in, use following link to connect through Jupyter:

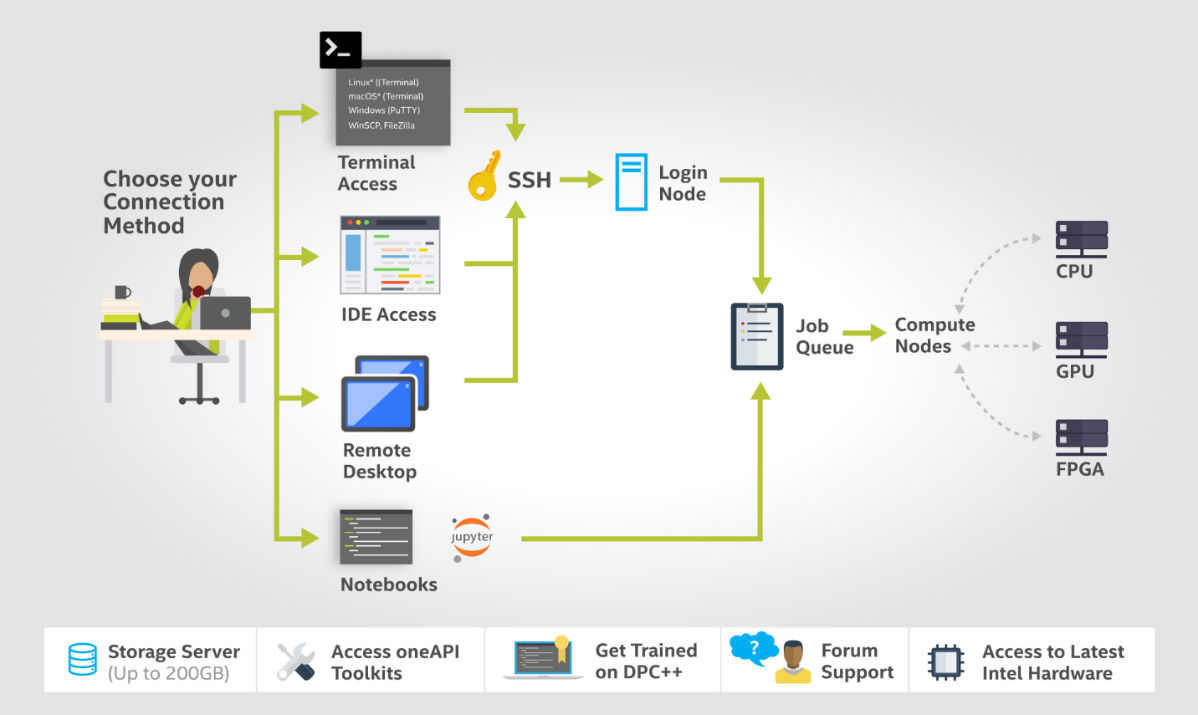
<https://devcloud.intel.com/oneapi/get_started/>

At the end of this page, click on Jupyter lab button. It will redirect you to the devcloud.



**Architecture of DevCloud:**

The cluster consists of multiple compute servers, which we call compute nodes, storage servers, and the login node.



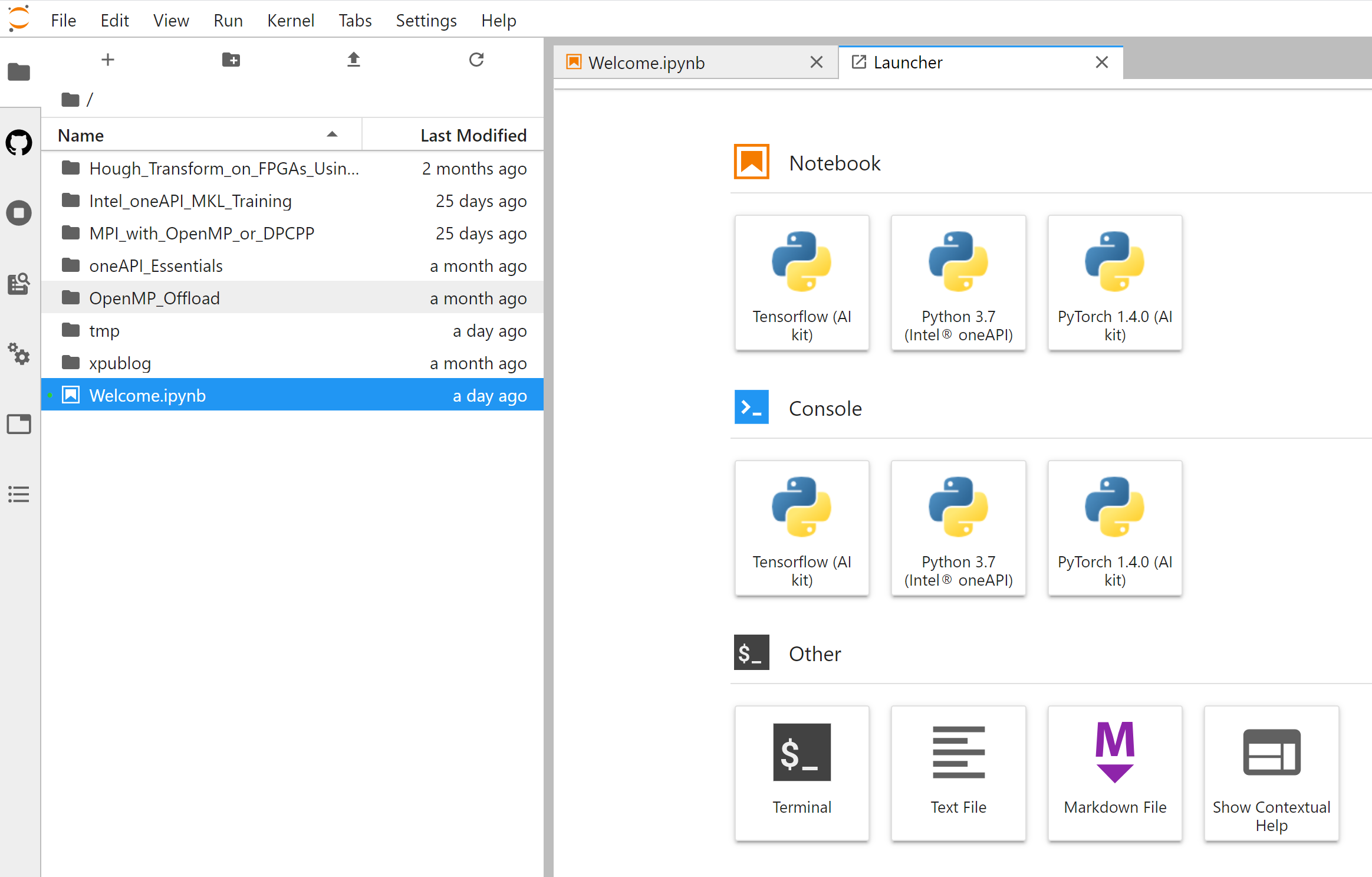
Your Jupyter Notebook instance runs on one of the compute nodes. When you execute the Notebook cells, they run on an Intel Xeon Scalable processor. There may be other people connecting to your compute node, which reduces the available compute power. However, you can reserve a full node for your job and even use multiple nodes at once by submitting scripts to the job queue.

<https://devcloud.intel.com/datacenter/learn/getting-started/>

1. **Jupyter Notebook Basics:**

**How to open terminal:**

Open the launcher (red circle) and click on terminal icon:



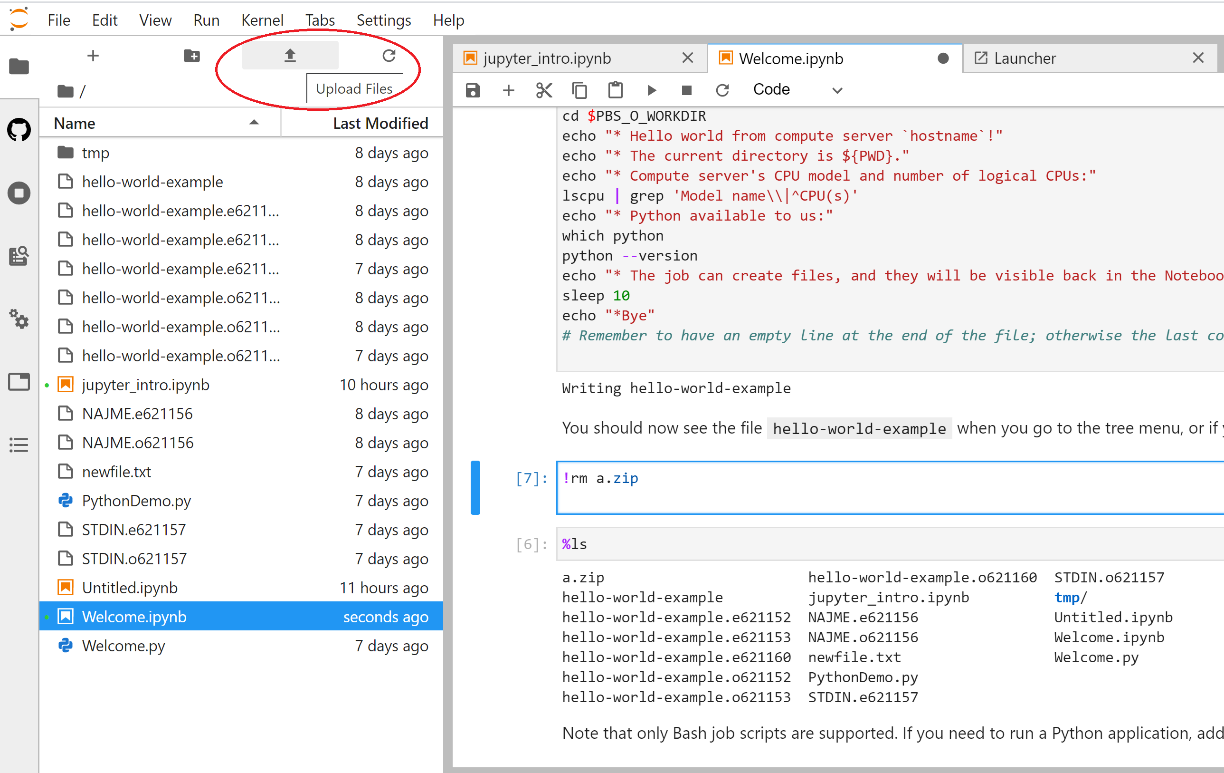
It will open a terminal for you.

**Data Management:**

When you connect through the Jupyter notebook, you will see folder in your home directory named /*tmp*. Do not use this folder. Be aware that Home folder is NFS-shared between the login node and the compute nodes.

* Transferring Data between your local machine and DevCloud:

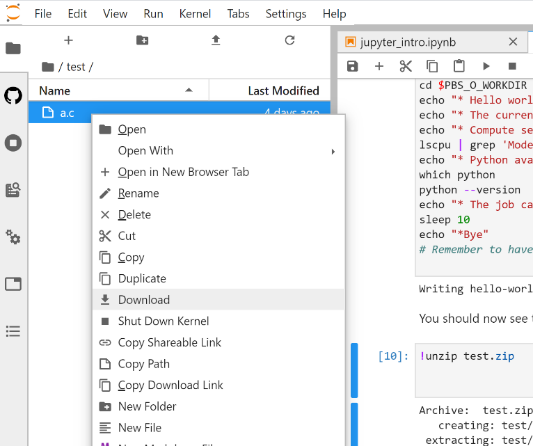
You may ask, how you can upload your data and source codes from your local system to the DevCloud environment or how to download your outputs and results from DevCloud to your local machine. In Jupyter, this is an easy task. You can zip your working directory in your local machine and upload it to Jupyter environment as follow (the red circle):



Now, please upload a Workshop\_HandsOn.zip to the DevCloud and then unzip it with following command:

$ unzip Workshop\_HandsOn.zip

Now, if you want to download the content of a directory (such as a file), you can right click on the file and select download. It will download the file to your local machine.



1. **DPCPP Compiler:**

In this part, you will learn how to compile the DPC++ designs using the compiler available on devcloud.

Please go to the directory and explore the content of Workshop\_HandsOn. The available files are a binary file for decryption a message, a key file for AES, different versions (host, emulation, FPGA) of DPC++ code for AES, and job scripts required for this lab.

**Compilation with Command Line**:

$ dpcpp <source\_files> -<options>

This command runs the compiler. The source files can be dpc++ codes. Options specify what the compilers does with the source file.

**Options**:

* -fintelfpga

This option lets you perform ahead-of-time compilation with FPGA. It compiles with dependencies and debug enabled.

* -fsycl-link

Tells the compiler to perform a partial link of device binaries to be used with FPGA. If it used with -fintelfpga, then it tells compiler to generate an html report when the partial link is created. Then an FPGA image will be generated.

* -Xs

This will passes the options to the backend tool. Such as -Xshardware that will be used for generating hardware and -Xsprofile that will be used to enable vtune to profile the hardware.

* -c

This option prevents linking and compiler will stop after generating object files.

* -D*name*

Defines a macro name that can be associated with an optional value. It is equivalent to #define preprocessor directives.

**Compile the software version of AES**:

Open a terminal and compile the code for the host using the following command:

$ dpcpp aes\_dpcpp\_sw.cpp -o aes\_sw -DCPU\_HOST

$ ./aes\_sw

Check the success of encryption by running the decrypt binary file. Before that, change the mode of file using the command below:

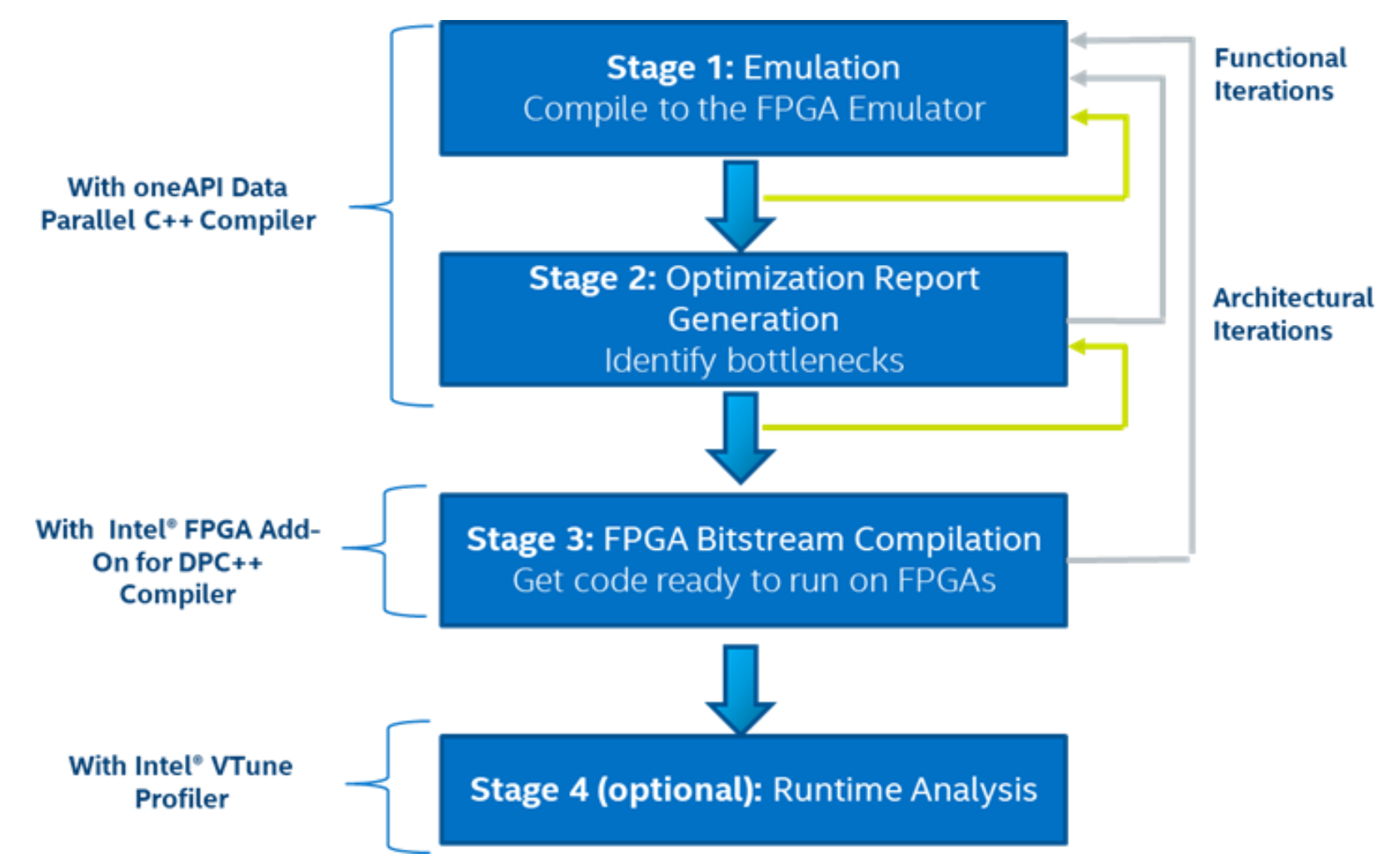
$ chmod +x decrypt

$ ./decrypt

By running the command, you will see the encrypted message was “abcdefjhij”

**Emulate AES design**:

The hardware generation flow is as follow:



*Emulation*: Validates code functionality by compiling on the CPU to simulate computation.

*Optimization Report Generation*: Generates an optimization report that describes the structures generated on the FPGA, identifies performance bottlenecks, and estimates resource utilization.

*Bitstream Compilation*: Produces the real FPGA bitstream/image to execute on the target FPGA platform.

*Runtime Analysis*: Generates output files containing the metrics and performance data.

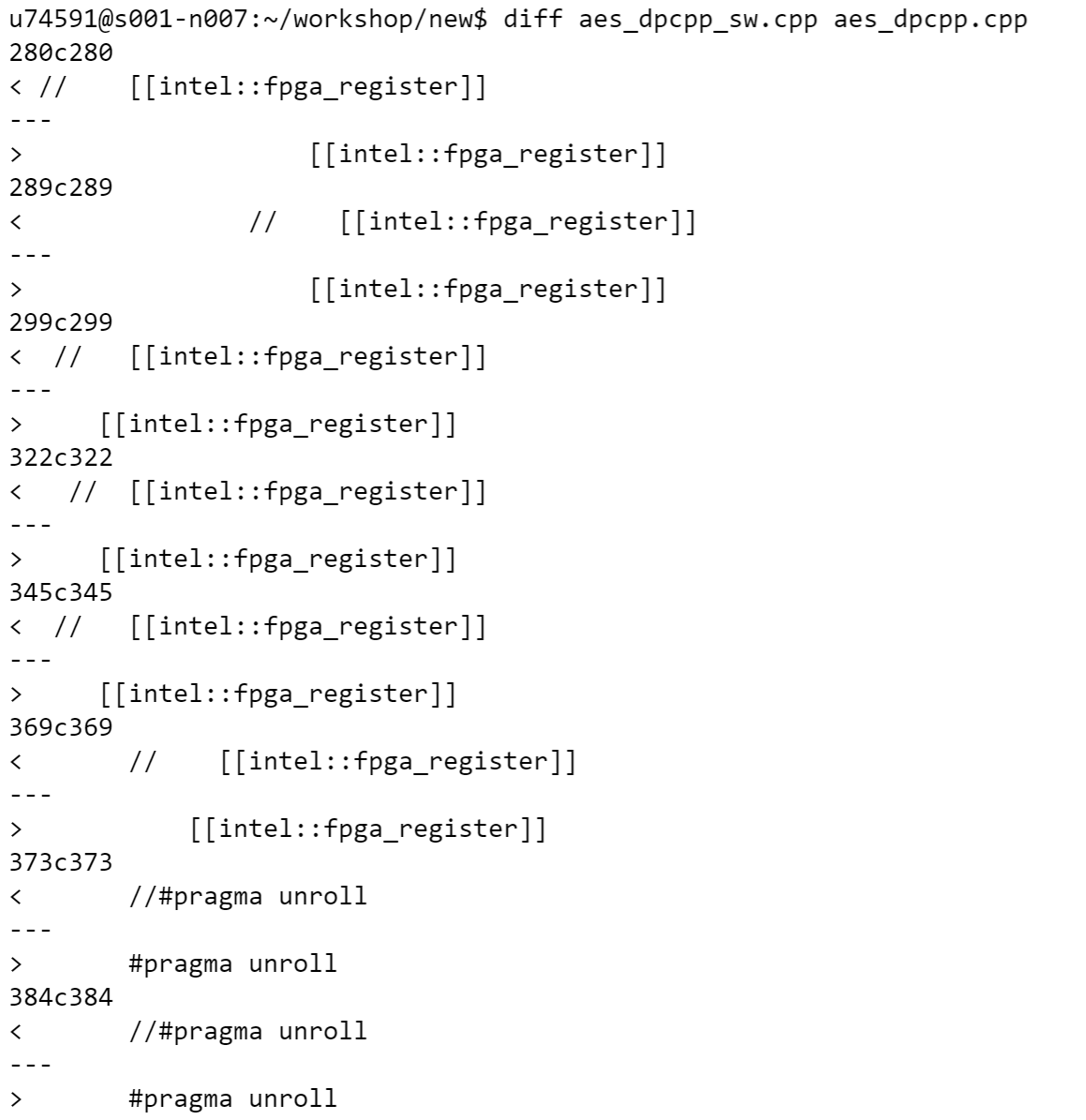
In this lab, we focus on Runtime Analysis, stage 4. Stages 1, 2, and 3 will be covered on the other workshop ([**Using Intel® oneAPI Toolkits with FPGAs**](https://www.fccm.org/workshops-tutorials~/#Using-Intel-oneAPI-Toolkits-with-FPGAs)).

Compared to CPU or GPU, generating a device image for FPGA hardware is a computationally intensive and time-consuming process. It is usual for an FPGA compile to take several hours to complete. Therefore, the FPGA emulator is the fastest method to verify the correctness of your code. The FPGA emulator executes the DPC++ device code on the CPU. The emulator is similar to the SYCL\* host device, but unlike the host device, the FPGA emulator device supports FPGA extensions such as FPGA pipes.

Before emulating the AES dpc++ code for hardware implementation, let’s see what the differences are between the software version and the hardware version. Run the following command to see the differences:

$ diff aes\_dpcpp\_sw.cpp aes\_dpcpp.cpp

You can see that some pragmas and special FPAG extensions has been added to the code:



**Cheat sheet:**

Here is a cheat sheet of the DPC++ compiler commands to compile for the FPGA emulator, generate the FPGA early image optimization reports, and compile for FPGA hardware.

# FPGA emulator

dpcpp -fintelfpga -DFPGA\_EMULATOR fpga\_compile.cpp -o fpga\_compile.fpga\_emu

# Optimization report (default board)

dpcpp -fintelfpga -Xshardware -fsycl-link=early fpga\_compile.cpp -o fpga\_compile\_report.a

# FPGA hardware (default board)

dpcpp -fintelfpga -Xshardware -Xsprofile fpga\_compile.cpp -o fpga\_compile.fpga

Make sure to include -Xsprofile that you can run intel vtune for CPU and FPGA interaction analysis.

Now let’s emulate AES using existed jobscripts to ensure that the dpc++ code works correctly. Before that, let see what the job is?

**What is a Job:**

Anything that we want to run on DevCloud is a job such as training a neural network, compiling C++ code, and running an application. The job queue is the only method for accessing the full capacity of the computing resources available on the DevCloud.

**Submitting a Job to the Queue**:

You can submit your job script using the *qsub* command as follow:

!qsub <name of your job script>

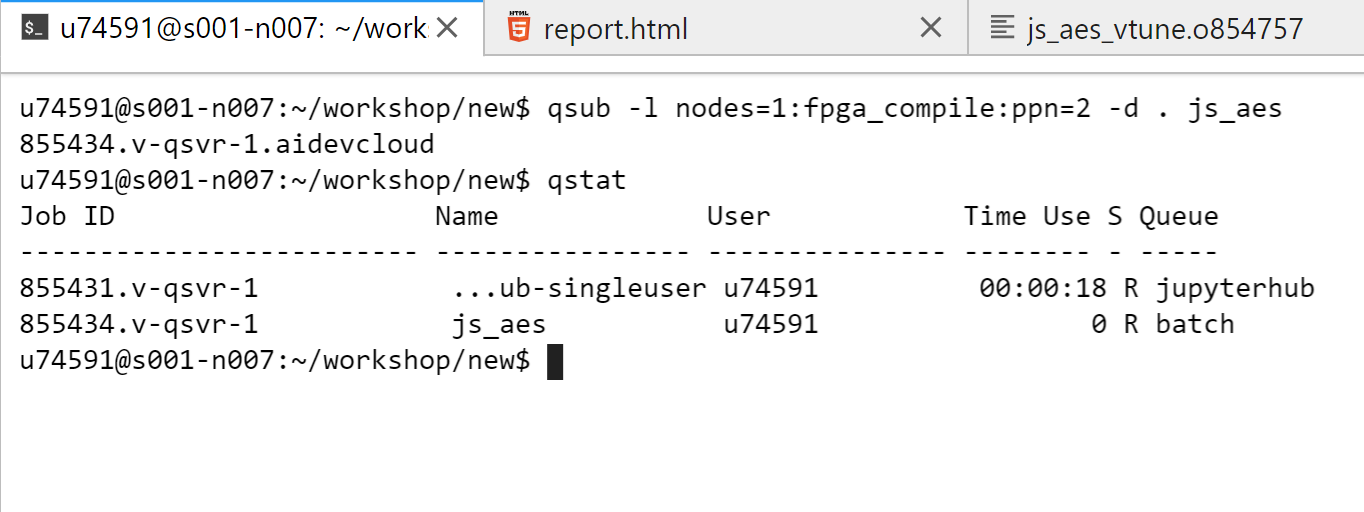
*!qsub myjob*

Now let’s emulate AES using existed jobscripts to ensure that the dpc++ code works correctly. For this purpose, submit the js\_aes as follow:

$ qsub -l nodes=1:fpga\_compile:ppn=2 -d . js\_aes

Use following command to see the status of the job:

$ qstat



In the column S you will see a letter indicating its status: "Q" is for "queued", "R" is for "running", and "E" is either an error, or a transition to a normal job completion. The result will be in the output .o file. After emulation is done, fpga.emu file will be generated. You should constantly check the status until see that the job (js\_aes) is no longer showing in the list.

* Getting the result:

Once the job is completed, the resulting output and error streams (stdout and stderr) are placed in two separate text files. These output files have the following naming convention:

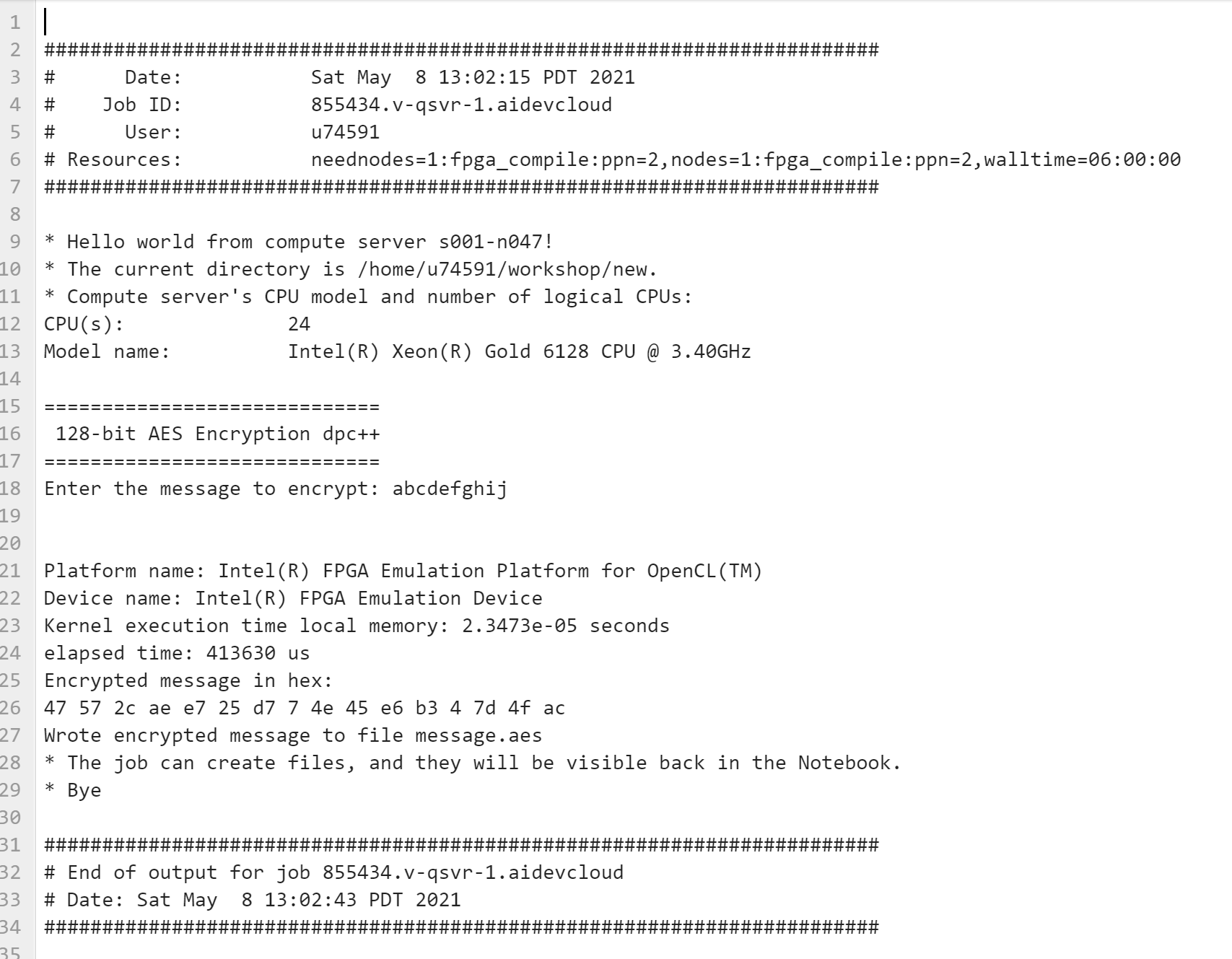
* stdout: [Job Name].o[Job ID]. Example: myjob.o12345
* stderr: [Job Name].e[Job ID]. Example: myjob.e12345

[Job Name] is either the script name, or a custom name — for example, the name specified by the -N parameter of *qsub*.

[Job ID] is the number you got from the output of the *qsub* command.

Using *%cat* command, you can see the content of a file:

$ cat js\_aes.oXXX



When the emulation is done, you can run the emulated file as follow**:**

$ ./fpga.emu

You can see that it works similar to software version. You can verify it by the decrypt file.

After emulation, you need to compile for FPGA and generate the hardware. As it is a time consuming task, we skip this part. But we provided the bitstream file for you to use, “a.out” file. So, you need to give it run permissionas follow:

$ chmod +x a.out

1. **Performance analysis**

## Run Predefined Analysis

The predefined analysis configurations already have most of the *knobs*  (configuration options) set by default for your convenience. To run a predefined performance analysis, use the - collect action:

$ vtune -collect <analysis\_type> [-knob <knobName=knobValue>] [--] <target>

where:

* *<analysis\_type>*  is the type of analysis to run. To see the list of available analysis types, enter: vtune  -help collect
* -target-system  is an option targeted for remote analysis and specifies a remote Linux\* system or a Android\* device
* -knob  is a configuration option that modifies the analysis [*knobName=knobValue* ]  is the name of the specified knob and its value
* *<target>*  is the path and name of the application to analyze. If you need to analyze a process, use the  [-target-process](https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/command-line-interface/command-line-interface-reference/target-process.html)  or  [-target-pid](https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/command-line-interface/command-line-interface-reference/target-pid.html) option to specify the process name or ID. For a system-wide analysis, no target specification is required.

| **Analysis Type** | **Description** |
| --- | --- |
| [performance-snapshot](https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/command-line-interface/running-command-line-analysis/running-performance-snapshot-from-the-command-line.html" \o "" \t "_self) | Get an overview of issues that affect application performance on your target system. |
| [hotspots](https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/command-line-interface/running-command-line-analysis/running-basic-hotspots-analysis-from-the-command-line.html" \o "" \t "_self) | Analyze application flow and identify sections of code that take a long time to execute (hotspots). |
| [hpc-performance](https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/command-line-interface/running-command-line-analysis/running-hpc-performance-analysis-from-the-command-line.html" \o "" \t "_self) | Identify opportunities to optimize CPU, memory, and FPU utilization for compute-intensive or throughput applications. The HPC Performance Characterization analysis type is a starting point for understanding the performance landscape of your application. Use this analysis type to improve application performance by increasing the number of floating-point operations per second (GFLOPS) and reducing the overall application run time. The analysis collects data related to CPU, memory, and FPU utilization. Additional scalability metrics are available for applications that use OpenMP\* or MPI runtime libraries. |
| [memory-consumption](https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/command-line-interface/running-command-line-analysis/running-memory-consumption-analysis-from-the-command-line.html) | Analyze memory consumption by your Linux application, its distinct memory objects and their allocation stacks. |
| [uarch-exploration](https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/command-line-interface/running-command-line-analysis/running-general-exploration-analysis-from-the-command-line.html" \o "" \t "_self)   (former  general-exploration  ) | Collect hardware events for analyzing a typical client application. This analysis calculates a set of predefined ratios used for the metrics and facilitates identifying hardware-level performance problems. |
| [fpga-interaction](https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top/command-line-interface/running-command-line-analysis/fpga-interaction-command-line-analysis.html) | Analyze the CPU/FPGA interaction issues via exploring OpenCL kernels running on FPGA, identify the most time-consuming FPGA kernels. |

In this lab, we focus on fpga-interaction analysis. We can use the CPU/FPGA Interaction analysis to assess the balance between CPU and FPGA in systems with FPGA hardware that run Data Parallel C++ (DPC++) or OpenCL™ applications. Also we can review FPGA time spent executing kernels, overall time for memory transfers between the CPU and FPGA, and wait time impact on CPU and FPGA workloads.

**Syntax**

$ vtune -collect fpga-interaction [-knob <knobName=knobValue>] [--] <target>

Knobs: sampling-interval, enable-stack-collection.

**Example**:

This example runs the CPU/FPGA Interaction analysis on an application with stack collection enabled:

$ vtune -collect fpga-interaction -knob enable-stack-collection=true -- /home/test/myApplication

This command must run on a system that has an FPGA. We provided the binary file that contain the hardware of AES and can be run on fpga. The file name is “a.out”.

To run Vtue for AES, submit js\_aes\_vtune as follow: It may take few minutes (6 to 7 minutes) to finish.

$ qsub -l nodes=1:fpga\_runtime:ppn=2 -d . js\_aes\_vtune

Check the status to make sure the jon is done! Then open the output file (js\_aes\_vtune.oXXXXXX) and explore the results..

**CPU/FPGA Interaction View**

To interpret the performance data provided in the CPU/FPGA Interaction viewpoint, you may follow the steps below:

1. Define a Performance Baseline: Use the Elapsed Time value as a baseline for comparison of versions before and after optimization.
2. Assess FPGA Utilization: Look at the FPGA Top Compute Tasks list on the Summary window for a list of kernels running on the FPGA. Review the FPGA Utilization timeline, which shows how many kernels and transfers are executing at the same time on the FPGA.
3. Review Memory Transfers: Look at the Data Transferred column on the Bottom-up window or the Computing Queue rows on the Platform window to view the FPGA kernels and memory transfers.
4. Determine Workload Impact: The Context Switch Time metric on the Summary window shows the amount of time the CPU spent in context switches. Switch to the Platform window and hover over the timeline to view the reason for the context switch. In some cases, CPU context switches may represent CPU waits for the FPGA. Look at the FPGA Utilization line to identify times when the CPU may have been waiting on the FPGA and vice versa. For instance, when there is no FPGA activity, but CPU activity is high, it is likely that the FPGA is waiting for the CPU to complete a preparation step.
5. Review FPGA device metrics: Switch to the Bottom-up window to analyze Stalls, Global Bandwidth and Occupancy metrics and see how efficiently your kernels run on the FPGA device. Analyze the Idle % metrics values to understand the percentage of cycles when there were no valid work-items executing or stalling the memory or channel instruction. The Activity % metric shows the percentage of cycles a predicated channel or memory instruction is enabled.
6. Analyze channel depth: In the Bottom-up window, locate the Average and Maximum Channel Depth information for selected instances. If required, adjust the channel depth for your needs. If the channel is full all the time, the write side of the channel is working faster than the read side, and the channel will be stalling in the write kernel. If the channel is mostly empty, the read side is likely to be stalling, and if the channel is bigger than 32 bits deep, you can reduce it in size without a performance hit.
7. Analyze loops: Analyze the occupancy for profiled loops
8. Analyze Source of the host application part: Double-click the function you want to optimize to view its related source code file in the Source/Assembly window. You can open the code editor directly from the Intel® VTune™ Profiler and edit your code (for example, minimizing the number of calls to the hotspot function).
9. Analyze Source of the kernel running on FPGA device: Double-click the kernel to see FPGA device metrics per the kernel source lines. Use the Source view to see what channels and memories cause most stalls and how much data they transfer.

**Intel Advisor:**

1. Run Command Line Analysis:

To run the intel advisor, we use two scripts. The first script run the application and collects the data. It may takes 3 to 4 minutes. Run the following command to collect the data:

$ advisor-python $APM/collect.py advisor -- ./aes\_sw

The second script, analyze the data gathered by the first script. It may takes 30 secons. Now, run this command:

$ advisor-python $APM/analyze.py advisor

This command, perform a quick fast analysis. At the end, results will be stored at the following directory:

/advisor/e000/pp000/data.0

Go to that directory and open the html report to explore the results! If you cannot see anything on the browser, click on trust html!

At the end, you can zip and download the profiling folder to open it on your local system using a GUI based Intel Advisor for more details.