



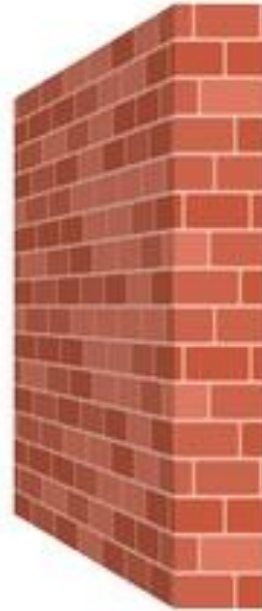
Instant FPGA deployment and scaling on the cloud

*Chris Kachris
Elias Koromilas
Ioannis Stamelos*

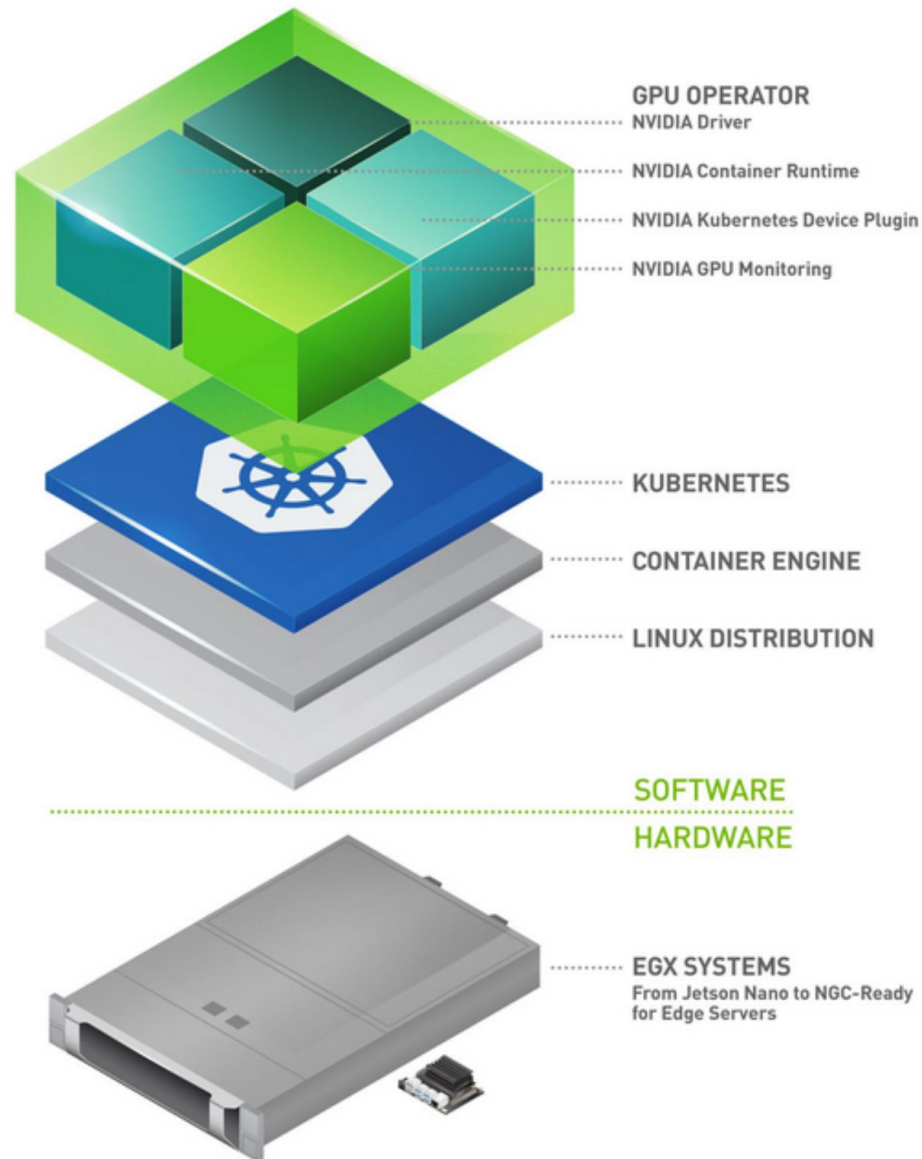
www.inaccel.com
info@inaccel.com

Main challenges on FPGA Deployment

- > What prevents the wide deployment of FPGA on Data centers/clusters



GPUs in the cloud

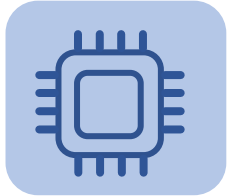


> Full ecosystem for easy deployment



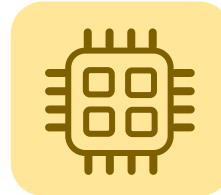
CPU – GPU - FPGAs

CPU



```
int threads = 100;
int id = 100;
#pragma omp parallel
{
    threads = omp_get_num_threads()
    id = omp_get_thread_num()
    std::cout << "hello from",id ;
}
return 0;
```

GPU



```
int main(void)
{
    int N = 1<<20;
    float *x, *y, *d_x, *d_y;
    x = (float*)malloc(N*sizeof(float));
    y = (float*)malloc(N*sizeof(float));

    cudaMalloc(&d_x, N*sizeof(float));
    cudaMalloc(&d_y, N*sizeof(float));

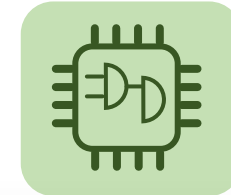
    for (int i = 0; i < N; i++) {
        x[i] = 1.0f;
        y[i] = 2.0f;
    }

    cudaMemcpy(...
cudaMemcpyHostToDevice);
    cudaMemcpy(...
cudaMemcpyHostToDevice);

    saxpy<<< (N+255)/256, 256>>>...;

    cudaMemcpy(y, d_y, N*sizeof(float),
cudaMemcpyDeviceToHost);
```

FPGA



```
std::string binaryFile = argv[1];
size_t vector_size_bytes = sizeof(int) * DATA_SIZE;
cl_int err;
cl::Context context;
cl::Kernel kernel_vector_add;
cl::CommandQueue q;
// Allocate Memory in Host Memory
// When creating a buffer with user pointer (CL_MEM_USE_HOST_PTR), under the hood user ptr
// is used if it is properly aligned. When not aligned, runtime had no choice but to create
// its own host side buffer. So it is recommended to use this allocator if user wish to
// create buffer using CL_MEM_USE_HOST_PTR to align user buffer to page boundary. It will
// ensure that user buffer is used when user create Buffer/Hem object with CL_MEM_USE_HOST_PTR
std::vector<int>, aligned_allocator<int>>> source_in1(DATA_SIZE);
std::vector<int>, aligned_allocator<int>>> source_in2(DATA_SIZE);
std::vector<int>, aligned_allocator<int>>> source_hw_results(DATA_SIZE);
std::vector<int>, aligned_allocator<int>>> source_sw_results(DATA_SIZE);

// Create the test data
std::generate(source_in1.begin(), source_in1.end(), std::rand);
std::generate(source_in2.begin(), source_in2.end(), std::rand);
for (int i = 0; i < DATA_SIZE; i++) {
    source_sw_results[i] = source_in1[i] + source_in2[i];
    source_hw_results[i] = 0;
}

// OPENCL HOST CODE AREA START
// get_xil_devices() is a utility API which will find the xilinx
// platforms and will return list of devices connected to Xilinx platform
auto devices = xcl::get_xil_devices();
// read_binary_file() is a utility API which will load the binaryFile
// and will return the pointer to file buffer.
auto fileBuf = xcl::read_binary_file(binaryFile);
cl::Program::binaries bins({fileBuf.data(), fileBuf.size()});
int valid_device = 0;
for (unsigned int i = 0; i < devices.size(); i++) {
    auto device = devices[i];
    // Creating Context and Command Queue for selected device
    OCL_CHECK(err, context = cl::Context({device}, NULL, NULL, &err));
    OCL_CHECK(err,
        q = cl::CommandQueue(
            context, {device}, CL_QUEUE_PROFILING_ENABLE, &err));

    std::cout << "Trying to program device[" << i
        << "]: " << device.getInfo<CL_DEVICE_NAME>() << std::endl;
    OCL_CHECK(err,
        cl::Program program(context, {device}, bins, NULL, &err));
    if (err != CL_SUCCESS) {
        std::cout << "Failed to program device[" << i
            << "]: " << device.getInfo<CL_DEVICE_NAME>() << std::endl;
    } else {
        std::cout << "Device[" << i << "]: program successful!\n";
        OCL_CHECK(err, kernel_vector_add = cl::Kernel(program, "vadd", &err));
        valid_device++;
    }
}
```

FPGA device

Bitstreams

Memory Allocation

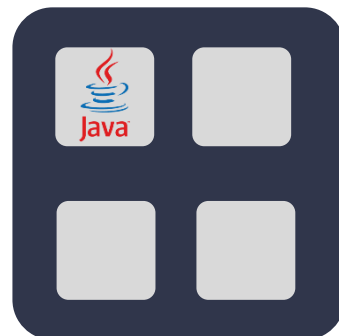
Transfer

Challenges on FPGAs – Deployment

> How can I **deploy** my FPGA accelerator easy?



> Without having to specify on host code about bitstreams, FPGA card, memory management, memory transfers



Challenges on FPGAs – Scaling

> How can I instantly **scale-out** my applications to multiple FPGAs?



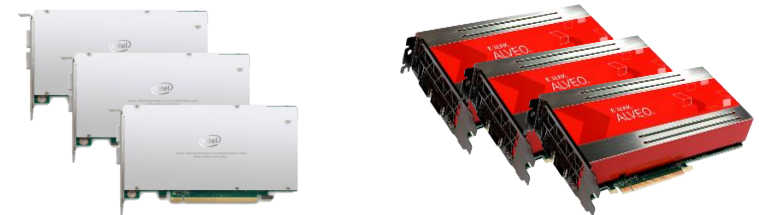
> Manually distribution on workload on different FPGAs.

- >> Error-prone
- >> Complex
- >> Not scalable

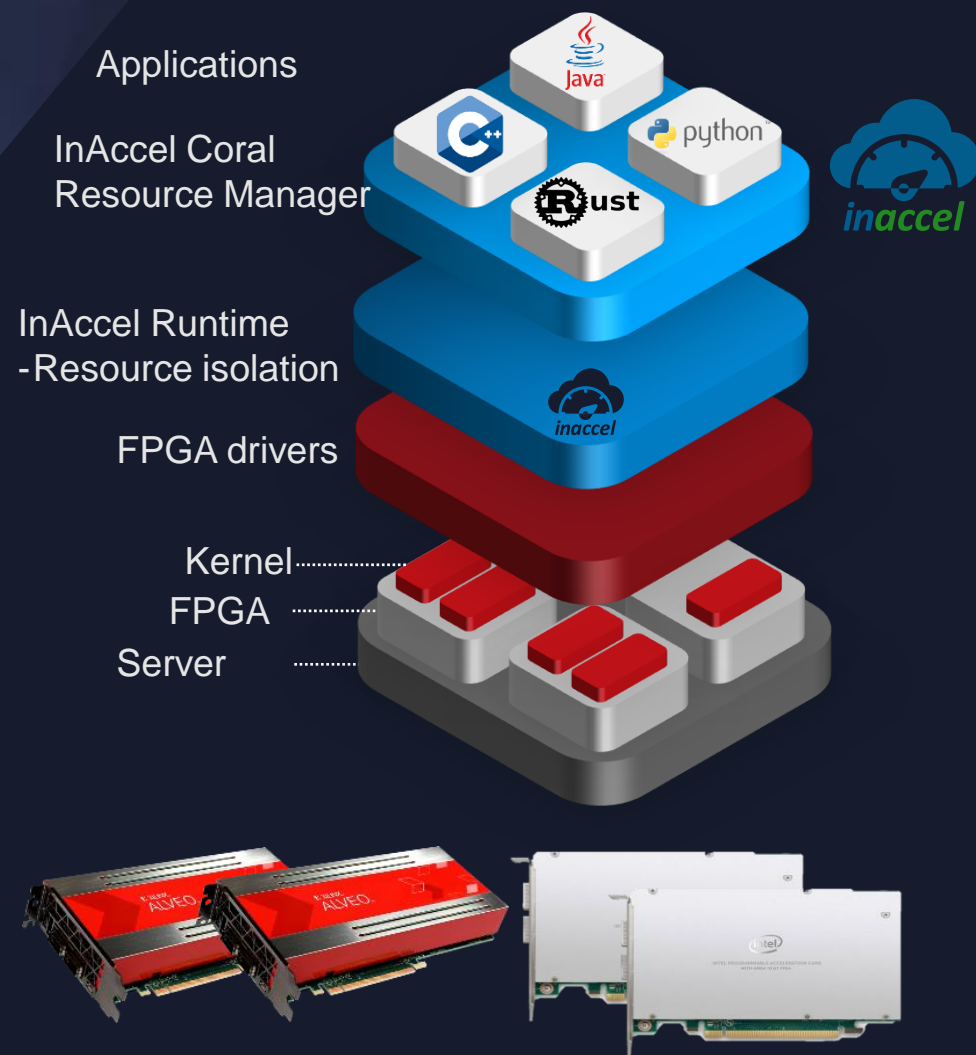


Challenges on FPGAs – Resource management

- > How can **multiple** users or applications **share** my FPGA cluster?
- > Currently only a single application can control the FPGA configuration
- > Hard to share FPGA resource among users/threads/processes



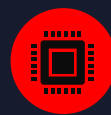
Scalable Orchestrator for FPGA clusters



Automated Deployment, Scaling and Management of FPGA clusters



Seamless invoking from C/C++, Python, Java and Scala. No need for OpenCL



Automatic configuration and management of the FPGA **bitstreams** and **memory**



Seamless **resource management** of the FPGA cluster from multiple threads/processes/applications/users



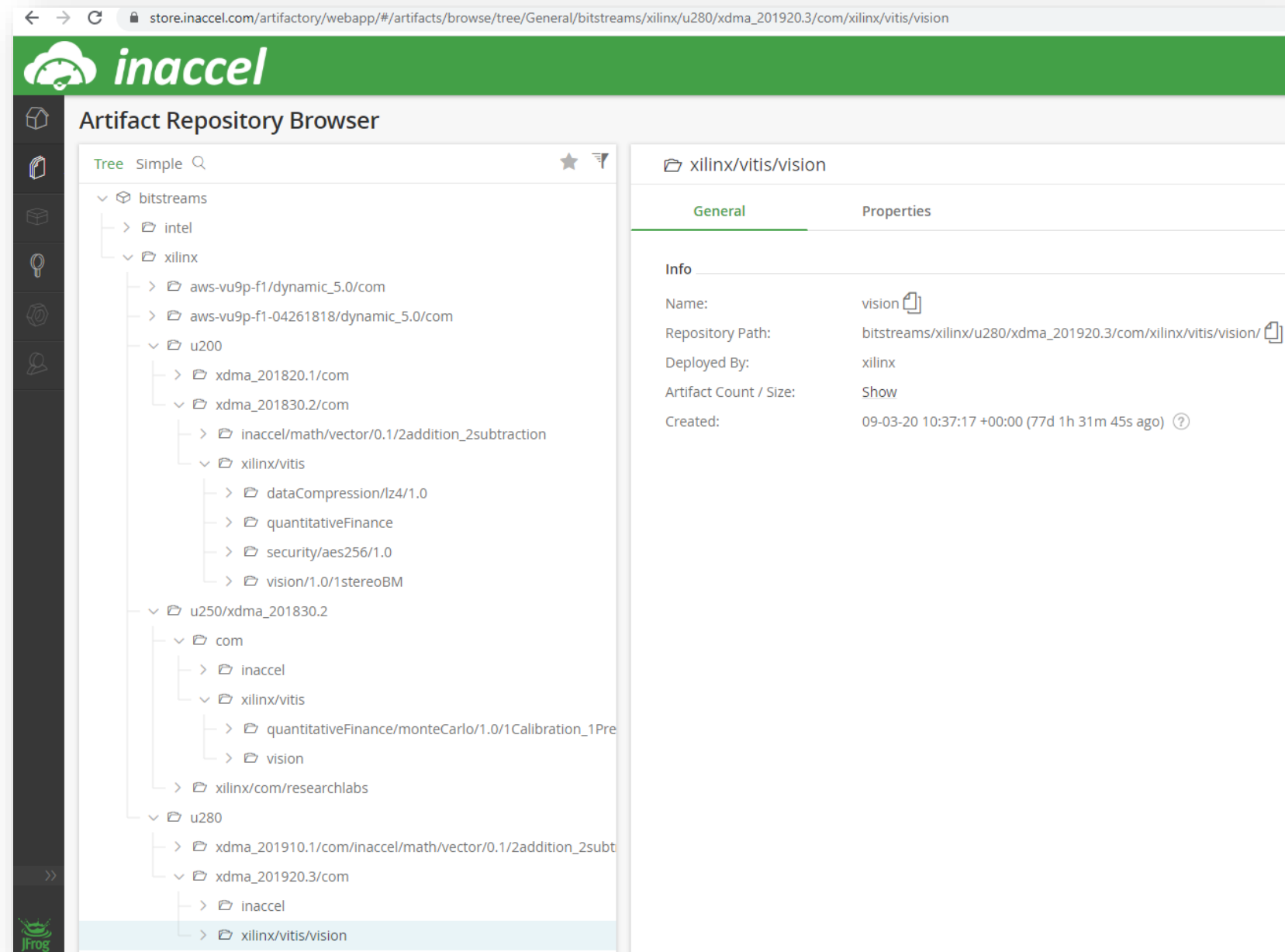
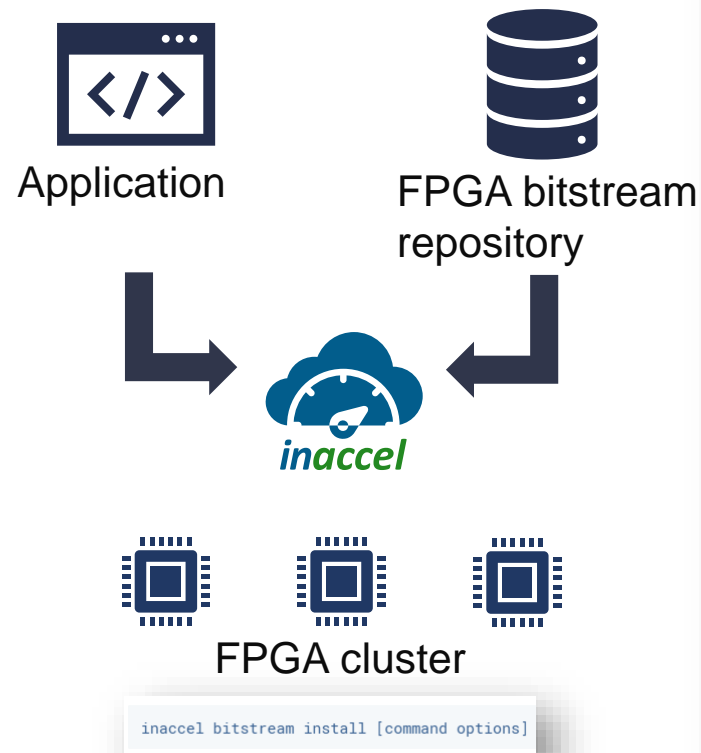
Fully **scalable**: Scale-up (multiple FPGAs per node) and Scale-out (multiple FPGA-based servers over Spark)

Bitstream repository



- > FPGA Resource Manager is integrated with a bitstream repository that is used to store FPGA bitstreams

<https://store.inaccel.com>



Simple deployment – InAccel API



```
std::string binaryFile = argv[1];
size_t vector_size_bytes = sizeof(int) * DATA_SIZE;
cl_int err;
cl::Context context;
cl::Kernel krnl_vector_add;
cl::CommandQueue q;
// Allocate Memory in Host Memory
// When creating a buffer with user pointer (CL_MEM_USE_HOST_PTR), under the hood user ptr
// is used if it is properly aligned. when not aligned, runtime had no choice but to create
// its own host side buffer. So it is recommended to use this allocator if user wish to
// create buffer using CL_MEM_USE_HOST_PTR to align user buffer to page boundary. It will
// ensure that user buffer is used when user create Buffer/Mem object with CL_MEM_USE_HOST_PTR
std::vector<int, aligned_allocator<int>> source_in1(DATA_SIZE);
std::vector<int, aligned_allocator<int>> source_in2(DATA_SIZE);
std::vector<int, aligned_allocator<int>> source_hw_results(DATA_SIZE);
std::vector<int, aligned_allocator<int>> source_sw_results(DATA_SIZE);

// Create the test data
std::generate(source_in1.begin(), source_in1.end(), std::rand);
std::generate(source_in2.begin(), source_in2.end(), std::rand);
for (int i = 0; i < DATA_SIZE; i++) {
    source_sw_results[i] = source_in1[i] + source_in2[i];
    source_hw_results[i] = 0;
}

// OPENCL HOST CODE AREA START
// get_xil_devices() is a utility API which will find the xilinx
// platforms and will return list of devices connected to Xilinx platform
auto devices = xcl::get_xil_devices();
// read_binary_file() is a utility API which will load the binaryFile
// and will return the pointer to file buffer.
auto fileBuf = xcl::read_binary_file(binaryFile);
cl::Program::Binaries bins({fileBuf.data(), fileBuf.size()});
int valid_device = 0;
for (unsigned int i = 0; i < devices.size(); i++) {
    auto device = devices[i];
    // Creating Context and Command Queue for selected Device
    OCL_CHECK(err, context = cl::Context({device}, NULL, NULL, NULL, &err));
    OCL_CHECK(err,
        q = cl::CommandQueue(
            context, {device}, CL_QUEUE_PROFILING_ENABLE, &err));

    std::cout << "Trying to program device[" << i
        << "]: " << device.getInfo<CL_DEVICE_NAME>() << std::endl;
    OCL_CHECK(err,
        cl::Program program(context, {device}, bins, NULL, &err));
    if (err != CL_SUCCESS) {
        std::cout << "Failed to program device[" << i
            << "]: " << " with xclbin file!\n";
    } else {
        std::cout << "Device[" << i << "]: program successful!\n";
        OCL_CHECK(err, krnl_vector_add = cl::Kernel(program, "vadd", &err));
        valid_device++;
    }
}
```



Host-side buffers only
Decouple applications from bitstreams
No platform-dependent device configurations



```
inaccel::request vadd("vector.addition");
vadd.arg(a).arg(b).arg(c).arg(size);
inaccel::submit(vadd).get();
```

- Simple programming using InAccel Coral API
- Asynchronous accelerator invocation
- No OpenCL directives
- Unified API in C/C++, Java, Python and Rust

<https://setup.inaccel.com/coral-api/#using-the-api>

```
unsigned int nbytes    = (width*height);

// Input and output buffers (Y,U,V)
YUVImage srcImage;
YUVImage dstImage;
srcImage.yChannel = (unsigned char *)malloc(nbytes);
srcImage.uChannel = (unsigned char *)malloc(nbytes);
srcImage.vChannel = (unsigned char *)malloc(nbytes);
dstImage.yChannel = (unsigned char *)malloc(nbytes);
dstImage.uChannel = (unsigned char *)malloc(nbytes);
dstImage.vChannel = (unsigned char *)malloc(nbytes);

// Create output buffers for reference results
unsigned char *y_ref = (unsigned char *)malloc(nbytes);
unsigned char *u_ref = (unsigned char *)malloc(nbytes);
unsigned char *v_ref = (unsigned char *)malloc(nbytes);

unsigned numRunsSW = comparePerf?numRuns:1;

#pragma omp parallel for num_threads(3)
for(unsigned int n=0; n<numRunsSW; n++)
{
    // Compute reference results
    Filter2D(filterCoeffs[filterType], factor, bias, width, height, srcImage.yChannel, y_ref);
    Filter2D(filterCoeffs[filterType], factor, bias, width, height, srcImage.uChannel, u_ref);
    Filter2D(filterCoeffs[filterType], factor, bias, width, height, srcImage.vChannel, v_ref);
}
```

Reference ConvFilter

```
unsigned int nbytes    = (width*height);

// Input and output buffers (Y,U,V)
YUVImage srcImage;
YUVImage dstImage;
srcImage.yChannel = (unsigned char *)inaccel_alloc(nbytes);
srcImage.uChannel = (unsigned char *)inaccel_alloc(nbytes);
srcImage.vChannel = (unsigned char *)inaccel_alloc(nbytes);
dstImage.yChannel = (unsigned char *)inaccel_alloc(nbytes);
dstImage.uChannel = (unsigned char *)inaccel_alloc(nbytes);
dstImage.vChannel = (unsigned char *)inaccel_alloc(nbytes);

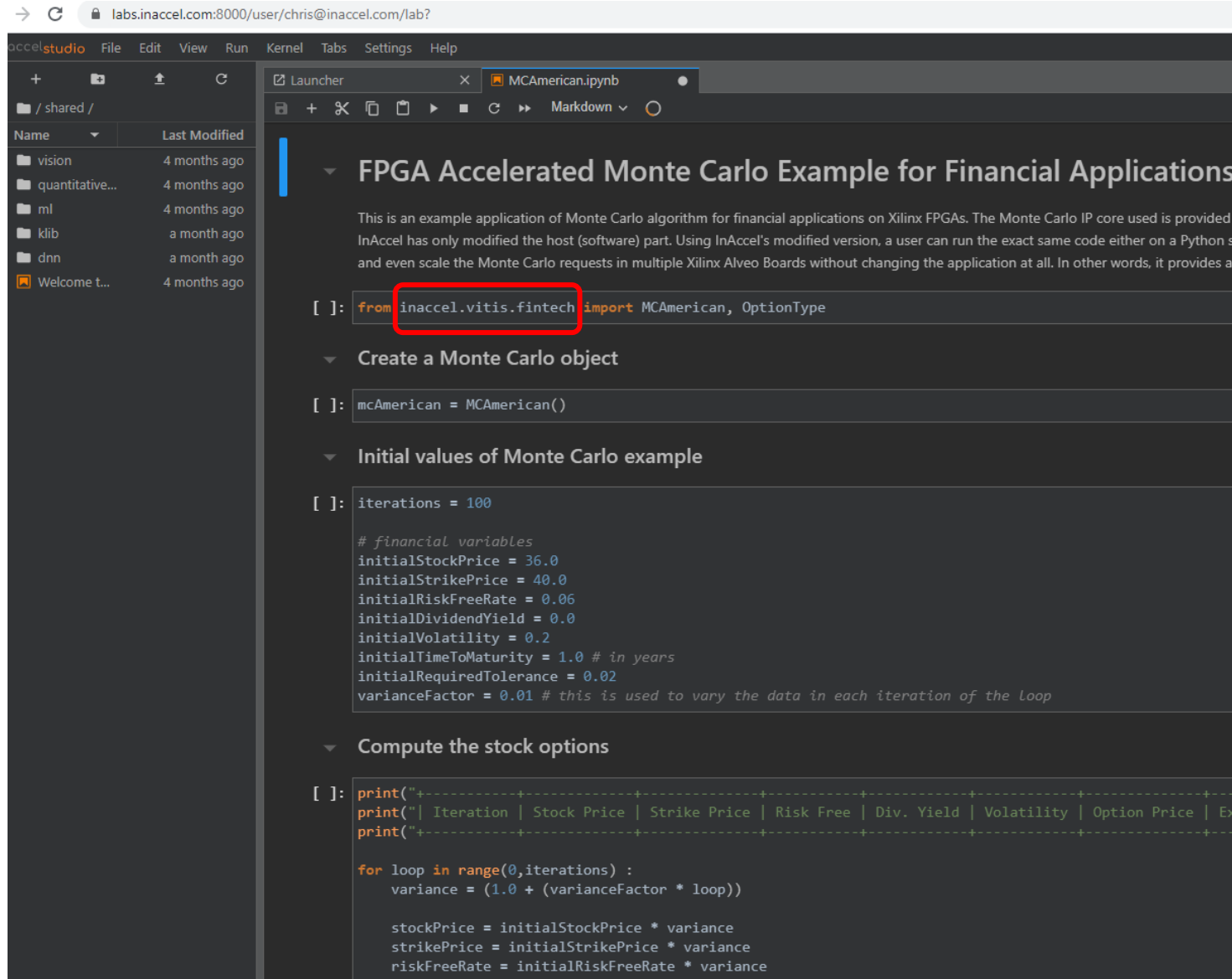
// Create output buffers for reference results
unsigned char *y_ref = (unsigned char *)inaccel_alloc(nbytes);
unsigned char *u_ref = (unsigned char *)inaccel_alloc(nbytes);
unsigned char *v_ref = (unsigned char *)inaccel_alloc(nbytes);

unsigned numRunsSW = comparePerf?numRuns:1;

#pragma omp parallel for num_threads(3)
for(unsigned int n=0; n<numRunsSW; n++)
{
    // Compute reference results
    Filter2D(filterCoeffs[filterType], factor, bias, width, height, srcImage.yChannel, y_ref);
    Filter2D(filterCoeffs[filterType], factor, bias, width, height, srcImage.uChannel, u_ref);
    Filter2D(filterCoeffs[filterType], factor, bias, width, height, srcImage.vChannel, v_ref);
}
```

InAccel ConvFilter

Ready to use accelerators



Instant evaluation of accelerations

No need for synthesis, P&R

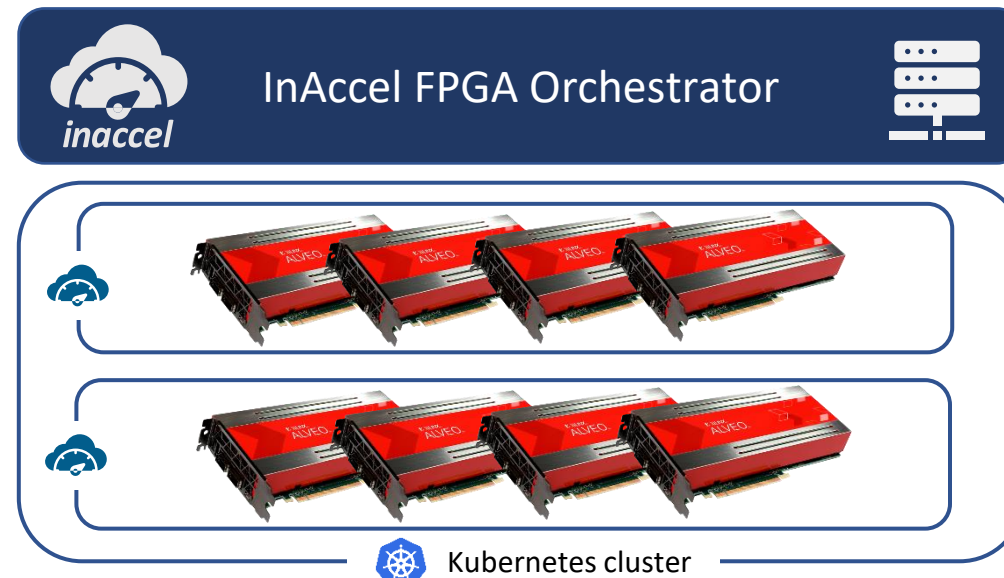
- No need for bitstreams
- No need for OpenCL
- No need for configuration/boards
- No need for an account



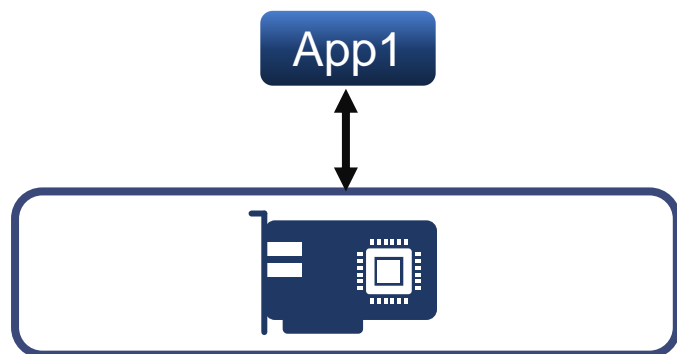
Multi-tenant Vitis deployment



- > Run Vitis from browser
- > Fully compatible with any Vitis library
- > Multi-tenant, multiple applications
- > Scalable deployment



From single node to scalable deployment



Single FPGA

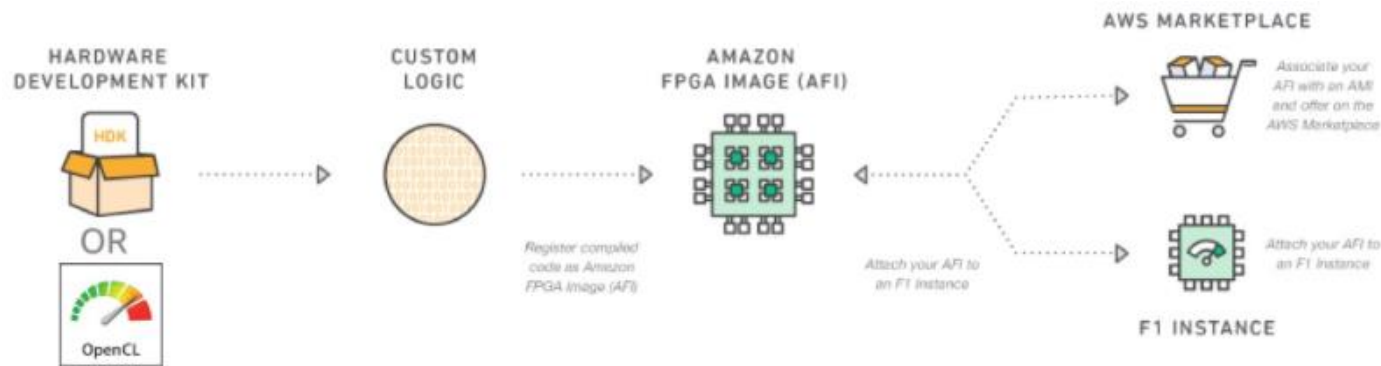


```
curl -sS https://setup.inaccel.com/repo | sh -s install
```



Deploy FPGAs on cloud

- > Several steps
- > Prior knowledge on FPGAs
 - >> Bitstream
 - >> Memory management
 - >> Communication
 - >> Challenges: Bitstream version, Firmware, SDK



How To Create an Amazon FPGA Image (AFI) From One of The CL Examples: Step-by-Step Guide

Fast path to running CL Examples on FPGA Instance

For developers that want to skip the development flow and start running the examples on the FPGA instance. You can skip steps 1 through 3 if you are not interested in the development process. Step 4 through 6 will show you how to use one of the predesigned AFI examples. By using the public AFIs, developers can skip the build flow steps and jump to step 4. Public AFIs are available for each example and can be found in the [example/README](#).

Step 1. Pick one of the examples and start in the example directory

It is recommended that you complete this step-by-step guide using HDK hello world example. Next use this same guide to develop using the `cl_dram_dma`. When your ready, copy one of the examples provided and modify the design files, scripts and constraints directory.

```
$ cd $HDK_DIR/cl/examples/cl_hello_world # you can change cl_hello_world to cl_dram_dma, cl_uram_example or cl_hello_world_vhdl
$ export CL_DIR=$(pwd)
```

Setting up the CL DIR environment variable is crucial as the build scripts rely on that value. Each example follows the recommended directory structure to match the expected structure for HDK simulation and build scripts.

Step 2. Build the CL

This [checklist](#) should be consulted before you start the build process.

NOTE This step requires you to have [Xilinx Vivado Tools and Licenses](#) installed

```
$ vivado -mode batch # Verify Vivado is installed.
```

Executing the `aws_build_dcp_from_cl.sh` script will perform the entire implementation process converting the CL design into a completed Design Checkpoint that meets timing and placement constraints of the target FPGA. The output is a tarball file comprising the DCP file, and other log/manifest files, formatted as `YY_MM_DD-Hmm.Developer.CL.tar`. This file will be submitted to AWS to create an AFI. By default the build script will use Clock Group A Recipe A0 which uses a main clock of 125 MHz.

```
$ cd $CL_DIR/build/scripts
$ ./aws_build_dcp_from_cl.sh
```

In order to use a 250 MHz main clock the developer can specify the A1 Clock Group A Recipe as in the following example:

```
$ cd $CL_DIR/build/scripts
$ ./aws_build_dcp_from_cl.sh -clock_recipe_a A1
```

Other clock recipes can be specified as well. More details on the [Clock Group Recipes Table](#) and how to specify different recipes can be found in the following [README](#).

NOTE: The DCP generation can take up to several hours to complete, hence the `aws_build_dcp_from_cl.sh` will run the main build process (`vivado`) in within a `nohup` context: This will allow the build to continue running even if the SSH session is terminated half way through the run

To be notified via e-mail when the build completes:

1. Set up notification via SNS:

```
$ pip install --user --upgrade boto3 # boto3 package is required by the notify_via_sns script
$ export EMAIL=your_email@example.com
$ AWS_FPGA_REPO_DIR/shared/bin/scripts/notify_via_sns.py
```

2. Check your e-mail address and confirm subscription
3. When calling `aws_build_dcp_from_cl.sh`, add on the `-notify` switch
4. Once your build is complete, an e-mail will be sent to you stating "Your build is done."
5. For each example the known warnings are documented in `warnings.txt` file located in the `$CL_DIR/build/scripts` directory `cl_hello_world` `warnings` `cl_dram_dma` `warnings` `cl_uram_example` `warnings`

Step 3. Submit the Design Checkpoint to AWS to Create the AFI

To submit the DCP, create an S3 bucket for submitting the design and upload the tarball file into that bucket. You need to prepare the following information:

1. Name of the logic design (Optional).
2. Generic description of the logic design (Optional).
3. Location of the tarball file object in S3.
4. Location of an S3 directory where AWS would write back logs of the AFI creation.
5. AWS region where the AFI will be created. Use `copy-fpga-image` API to copy an AFI to a different region.

To upload your tarball file to S3, you can use any of the [tools supported by S3](#).

Deployment of FPGA on Kubernetes – before



1. Install vendor-specific **FPGA drivers** and deployment shells on every node
2. Deploy the Intel/Xilinx FPGA **Device Plugin**
3. Develop OpenCL-based applications, that contain **platform-dependent** code
4. Build "fat" container images which **include large bitstream** files
5. Run **Kubernetes tasks** which are hard to maintain/upgrade
6. ... and still you need to **manually** perform **workload balancing** to distribute acceleration tasks along the requested FPGA resources

Kubernetes deployment - before



Using Alveo in a Kubernetes Environment

Article / Using Alveo in a Kubernetes Environment

Using Alveo in a Kubernetes Environment

Kester Aernoudt

Published: Apr 13, 2020

High-Performance Computing, Network Acceleration, Introductory Tutorials, Alveo U200, Vitis, Vitis Libraries

Overview

In a previous article (<https://developer.xilinx.com/en/articles/containerizing-alveo-accelerated-application-with-docker.html>) we showed how you can containerize Alveo Applications using Docker.

Kubernetes (K8s) is an open source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery.

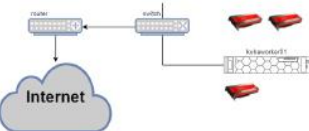
In this article, we will show how we can use Alveo enabled containers in a Kubernetes environment.

Local Kubernetes Network

For this example, we have set up a local environment with two servers. Server 1 (Kubernetes) has two Alveo U200 cards installed, server 2 (Kubeworker01) has one Alveo U200 card installed.

Feedback

Using Alveo in a Kubernetes Environment



This article is not meant to provide an overview on how to configure a Kubernetes cluster, but this is a short overview of what was done:

1. We installed CentOS 7.6 on both of the machines
2. Installed the necessary K8T and deployment shells on both of the servers. I used the 2019.2 version from <https://www.xilinx.com/products/boards-and-kits/alveo/package-files/archive/u200-2019-2.html>
3. Then we followed <https://www.linuxtutorials.com/install-kubernetes-1-7-centos7-dhe7/> to install Kubernetes on our machines

1. Since this is a small 2 machine cluster, I also configured the Kubernetes as a worker node:

```
kubectl taint nodes --all node-role.kubernetes.io/master-
```

After this procedure, we had Kubernetes running on our own small network.

Installing the Alveo Device Plugin

To add support to Kubernetes to manage Alveo cards, we install the Xilinx FPGA device plugin: https://github.com/Xilinx/FPGA_as_a_Service/tree/master/k8s-fpga-device-plugin/trunk.

Feedback

Using Alveo in a Kubernetes Environment

Discover the FPGAs installed in each node of the cluster and expose info of the FPGAs such as quantities, DSA(s) type, and Timestamp, etc.

Run FPGA accessible containers in the k8s cluster

Installing this plugin is very simple. On the Kubernetes machine, do this:

```
git clone https://github.com/Xilinx/FPGA_as_a_Service.git
cd k8s-fpga-device-plugin/trunk/
kubectl create -f fpga-device-plugin.yml
```

This will install the daemonset on all nodes, so in our setup, on both Kubernetes and kubeworker01. This can be verified using the following:

```
kubectl get pod -n kube-system
```

...snip...

fpga-device-plugin-daemonset-gg9d

1/1

Running

0

15d

fpga-device-plugin-daemonset-fg89

1/1

Running

0

15d

...snip...

If there is an Alveo card installed on the node, more info can be seen like this:

Feedback

Using Alveo in a Kubernetes Environment

```
time="2020-04-25T18:22:55Z" level=info msg="Starting FS watcher."
time="2020-04-25T18:22:55Z" level=info msg="Starting OS watcher."
time="2020-04-25T18:22:55Z" level=info msg="Starting to serve on /var/lib/kubelet
2020/04/25 18:22:55 grpc: Server.Serve failed to create ServerTransport: c
time="2020-04-25T18:22:55Z" level=info msg="Registered device plugin with Kube
time="2020-04-25T18:22:55Z" level=info msg="Sending 1 device(s) [kDeviceID:1,
time="2020-04-25T18:22:55Z" level=info msg="Receiving request 1"
time="2020-04-25T18:36:42Z" level=info msg="Receiving request 1"
```

To check the Alveo resource in the worker node, run this:

```
kubectl describe node kubeworker
```

...snip...

Capacity:

cpu: 4

ephemeral-storage: 4

memory: 16425412K

memory: 110

pod: 110

xilinx.com/fpga-xilinx_vu352_dynamic_3_1-152129439: 1

Allocatable:

cpu: 4

ephemeral-storage: 9463507952

memory: 0

memory: 16328012K

memory: 110

xilinx.com/fpga-xilinx_u200_xdma_201830_2-1561465320: 1

...snip...

Feedback

Using Alveo in a Kubernetes Environment

```
xilinx.com/fpga-cshwll--timestamp
```

For example:

```
xilinx.com/fpga-xilinx_u200_xdma_201830_2-1561465320
```

Here, xilinx_u200_xdma_201830_2 is the shell (DSA) version on the Alveo board, and 1561465320 is the timestamp when the shell was built.

The exact name of the Alveo resource on each node can be extracted from the output of:

```
kubectl describe node <node_name>
```

Deploy user pod

Here is an example of the yaml file which defines the pod to be deployed. In the yaml file, the docker image, which has been uploaded to a docker registry, should be specified. What should be specified as well is, the type and number of Alveo resources being used by the pod.

Feedback

Using Alveo in a Kubernetes Environment

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: xilinxorg/fpga-verify:latest
    resources:
      limits:
        xilinx.com/fpga-xilinx_u200_xdma_201830_2-1561465320: 1
      requests:
        xilinx.com/fpga-xilinx_u200_xdma_201830_2-1561465320: 1
    args: ["-c", "while true; do echo hello; sleep 5;done;"]
```

This pod can then be deployed as follows:

```
kubectl create -f dp-pod.yml
```

To check the status of the deployed pod:

Feedback

Using Alveo in a Kubernetes Environment

```
...snip...
mypod          1/1      Running    0       7s
...snip...

$kubectl describe pod mypod
...snip...

Limits:
  xilinx.com/fpga-xilinx_u200_xdma_201820_1-1535712995: 1
Requests:
  xilinx.com/fpga-xilinx_u200_xdma_201820_1-1535712995: 1
...snip...
```

To run hello world in the pod:

```
$kubectl exec -it mypod /bin/bash
my-pod-source /opt/xilinx/art/setup.sh
my-pod-xrtutil scan
my-pod-cd /tmp/alveo-u200/xilinx_u200_xdma_201830_1/test/
my-pod-./verify_xxx -./verify.xclbin
```

In this test case, the container image (xilinxorg/fpga-verify:latest) has been pushed to docker hub. It can be publicly accessed

The image contains verify.xclbin for many types of FPGAs, please select the type matching the FPGA resource the pod requests.

To run a more useful application, we pushed a couple of the compression examples of the Vitis Libraries (https://github.com/Xilinx/Vitis_Libraries) compiled for the U200 with XRT 2019.2 and DSA 2018.3.2 to dockerhub:

- https://hub.docker.com/v/testerarmoud/my_xlib

Feedback

Using Alveo in a Kubernetes Environment

```
apiVersion: batch/v1
kind: Job
metadata:
  name: mylib
spec:
  completions: 5
  parallelism: 5
  template:
    spec:
      containers:
      - name: mylib
        image: kesterarmoud/my_xlib:latest
        resources:
          limits:
            xilinx.com/fpga-xilinx_u200_xdma_201830_2-1561465320: 2
          requests:
            xilinx.com/fpga-xilinx_u200_xdma_201830_2-1561465320: 2
        command: ["/bin/sh"]
        args: ["-c", "/root/test.sh"]
      restartPolicy: Never
```

This job can be started using

```
kubectl apply -f my_xlib_job.yml
```

Getting status information for this job can be done as follows:

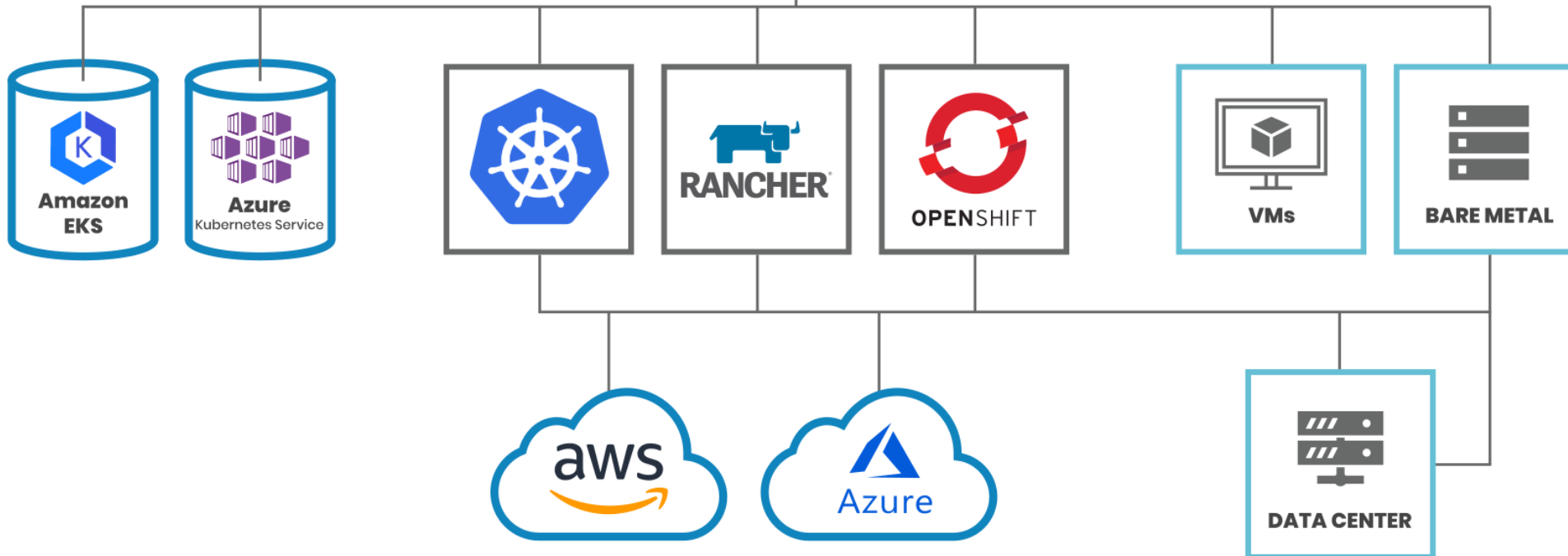
Feedback

<https://developer.xilinx.com/en/articles/using-alveo-in-a-kubernetes-environment.html>

Unique InAccel FPGA Operator

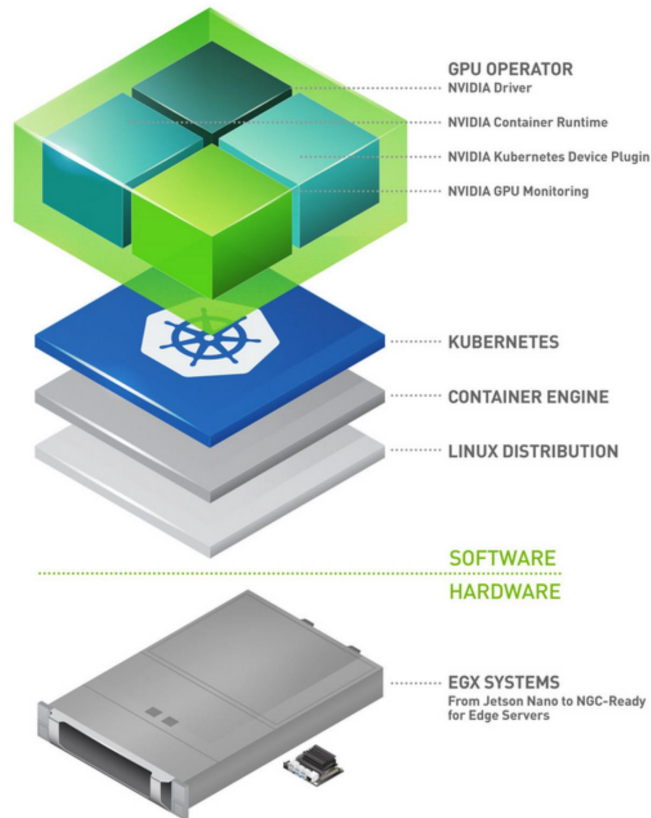


<https://artifacthub.io/packages/helm/inaccel/fpga-operator>



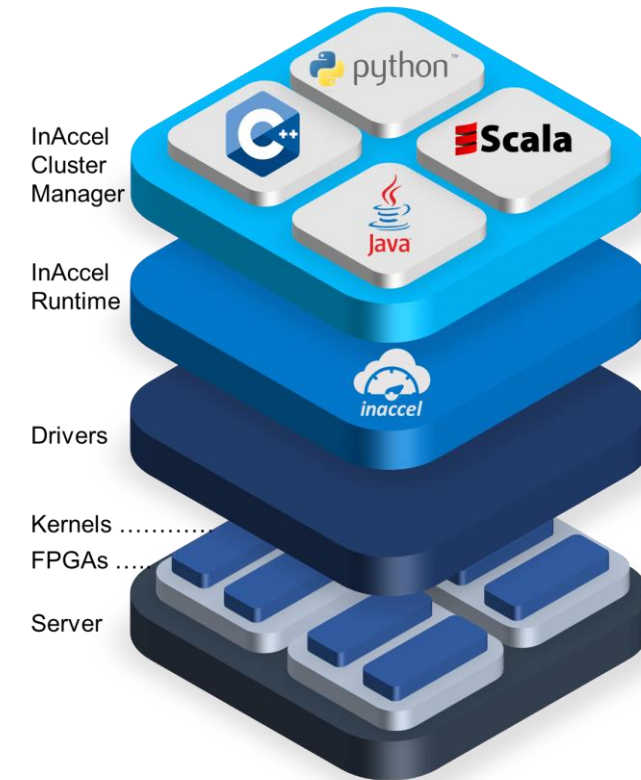
The InAccel FPGA operator manages FPGA resources in a Kubernetes cluster and automates tasks related to bootstrapping FPGA nodes.

Simplifying FPGA deployment in Kubernetes



NVIDIA - Full stack for GPUs

<https://developer.nvidia.com/blog/nvidia-gpu-operator-simplifying-gpu-management-in-kubernetes/>



Full stack for FPGAs (vendor agnostic)

FPGA deployment is a single step



1. Spawn a K8s cluster

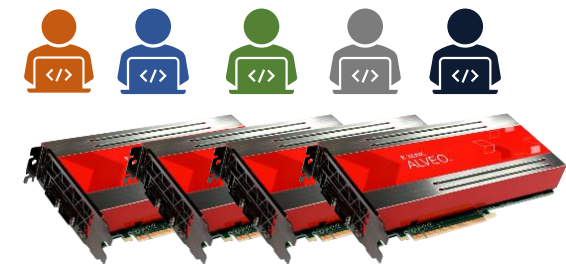
2. Deploy InAccel FPGA operator

1. `helm repo add inaccel https://setup.inaccel.com/helm`
2. `helm install my-fpga-operator inaccel/fpga-operator`

FPGA drivers/runtime, InAccel Coral Resource manager + Monitor

3. Run your application targeting FPGA resources

- > Multiple users
- > Auto-scaling
- > Easy resource management



Cloud deployment on Kubernetes



- > FPGA deployment on EKS cluster using Rancher UI and InAccel FPGA Operator

Cluster Explorer

All Namespaces

Cluster Manager

Jump to... (Ctrl+K)

Cluster Dashboard

Cluster

- Namespaces
- Nodes

Workload

- Overview
- CronJobs
- DaemonSets**
- Deployments
- Jobs
- StatefulSets
- Pods

Service Discovery

- HorizontalPodAutoscalers
- Ingresses
- NetworkPolicies
- Services

Storage

- PersistentVolumes
- StorageClasses
- ConfigMaps
- PersistentVolumeClaims
- Secrets

RBAC

DaemonSet: fpga-operator (Active)

Namespace: kube-system Age: 14 mins

Image: inaccel/coral:2.0

Labels: app.kubernetes.io/instance: my-release-p-b6lhx app.kubernetes.io/managed-by: Helm app.kubernetes.io/name: fpga-operator app.kubernetes.io/version: 2.2.0 helm.sh/chart: fpga-operator-2.2.0 io.cattle.field/appId: my-release-p-b6lhx

Annotations: Show 3 annotations

Pods by State

- 2 Active
- 0 Transitioning
- 0 Warning
- 0 Error

Pods Conditions Recent Events Related Resources

State	Name	Node	Image
Running	fpga-operator-6xrmf	ip-192-168-135-176.ec2.internal	inaccel/coral:2.0
Running	fpga-operator-sdtij	ip-192-168-87-154.ec2.internal	inaccel/coral:2.0

v2.5.9 Docs Forums Slack File an Issue

<https://www.youtube.com/watch?v=lqhIkX7oLBs>

FPGA on Kubernetes using InAccel



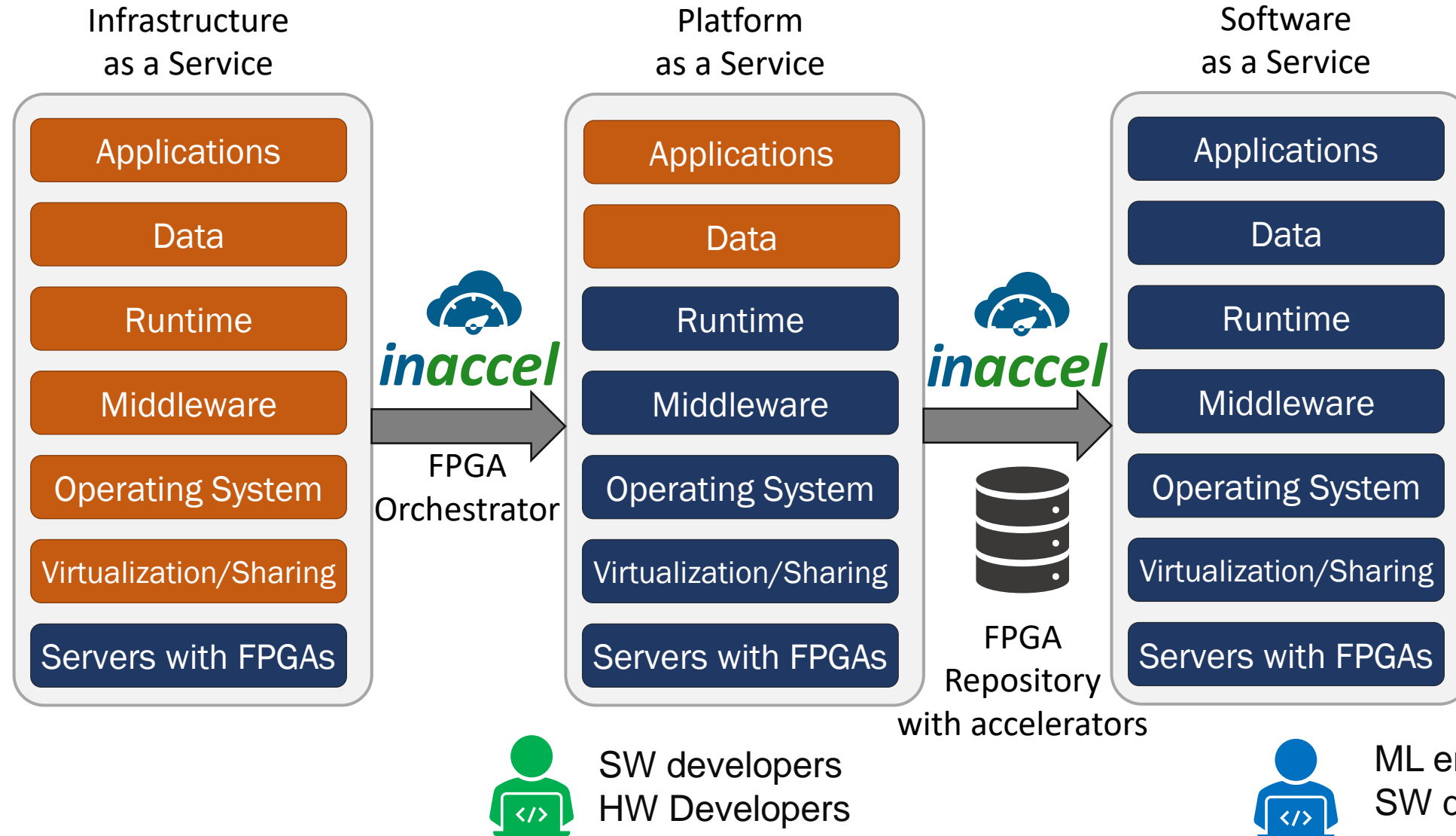
```
Kubernetes on FPGAs
```

This snippet explains how to run FPGAs in Kubernetes clusters like a pro

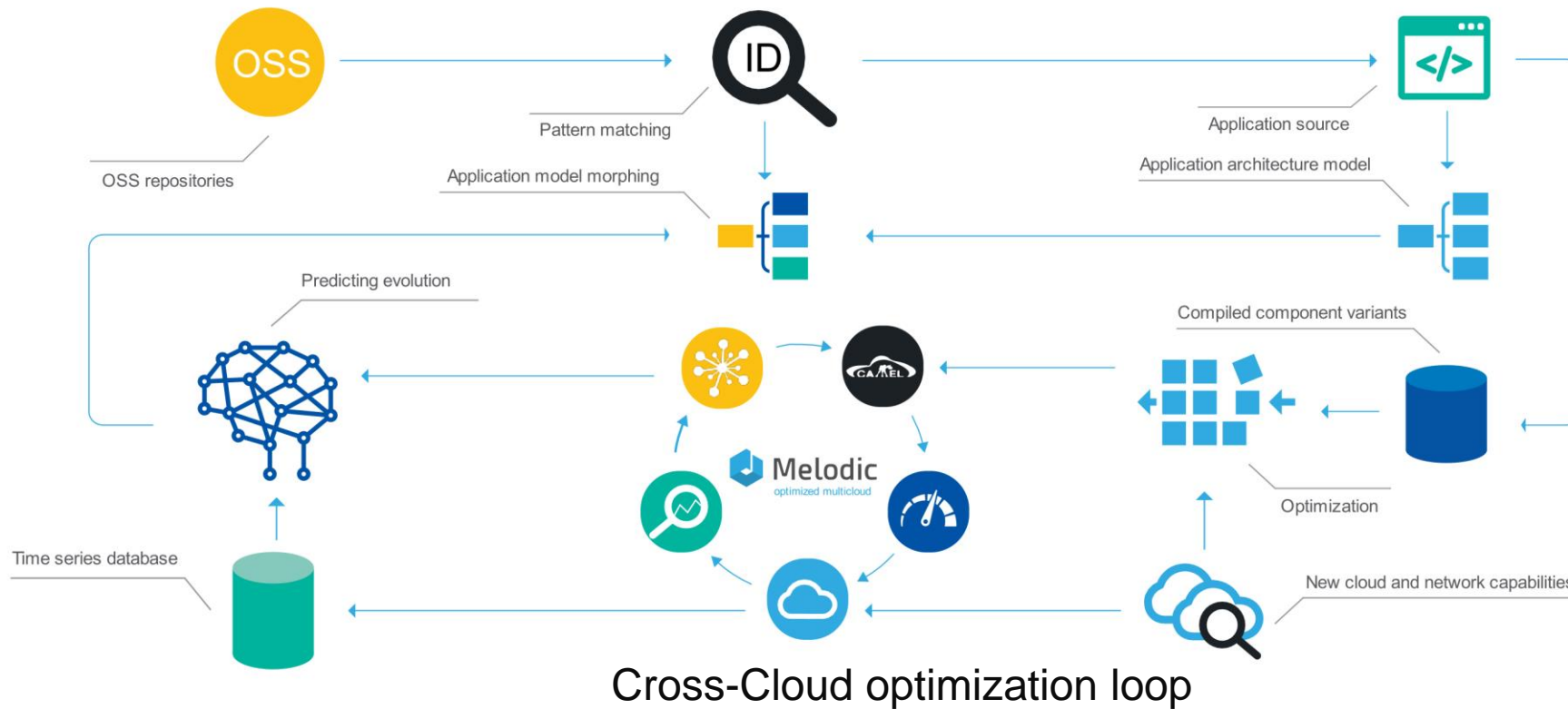
```
apiVersion: v1
kind: Pod
metadata:
  name: example
spec:
  containers:
  - name: example
    image: inaccel/jupyter:lab
    ports:
    - containerPort: 8888
```

<https://www.youtube.com/watch?v=E94YTh4mm1g>

PaaS and SaaS for FPGA clusters



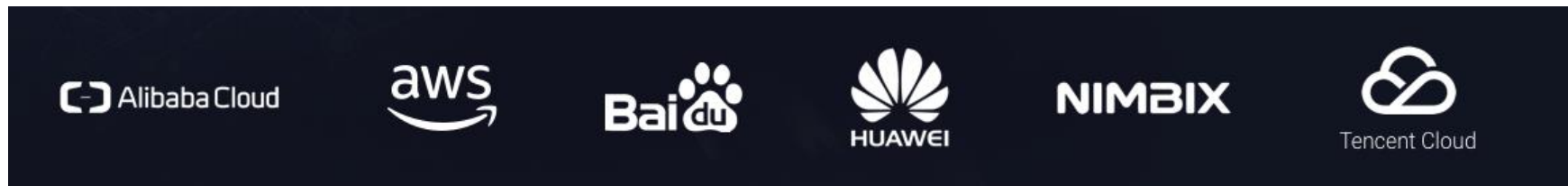
H2020: Multi-cloud deployment



How hardware accelerators can be deployed on multi-cloud environment?

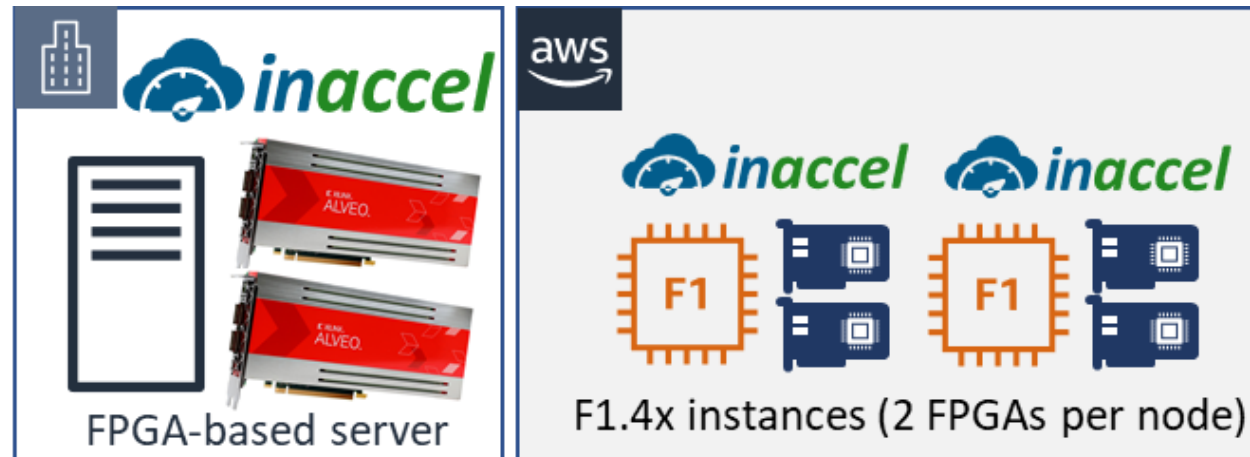


<https://www.morpheMIC.cloud/>



More Challenges

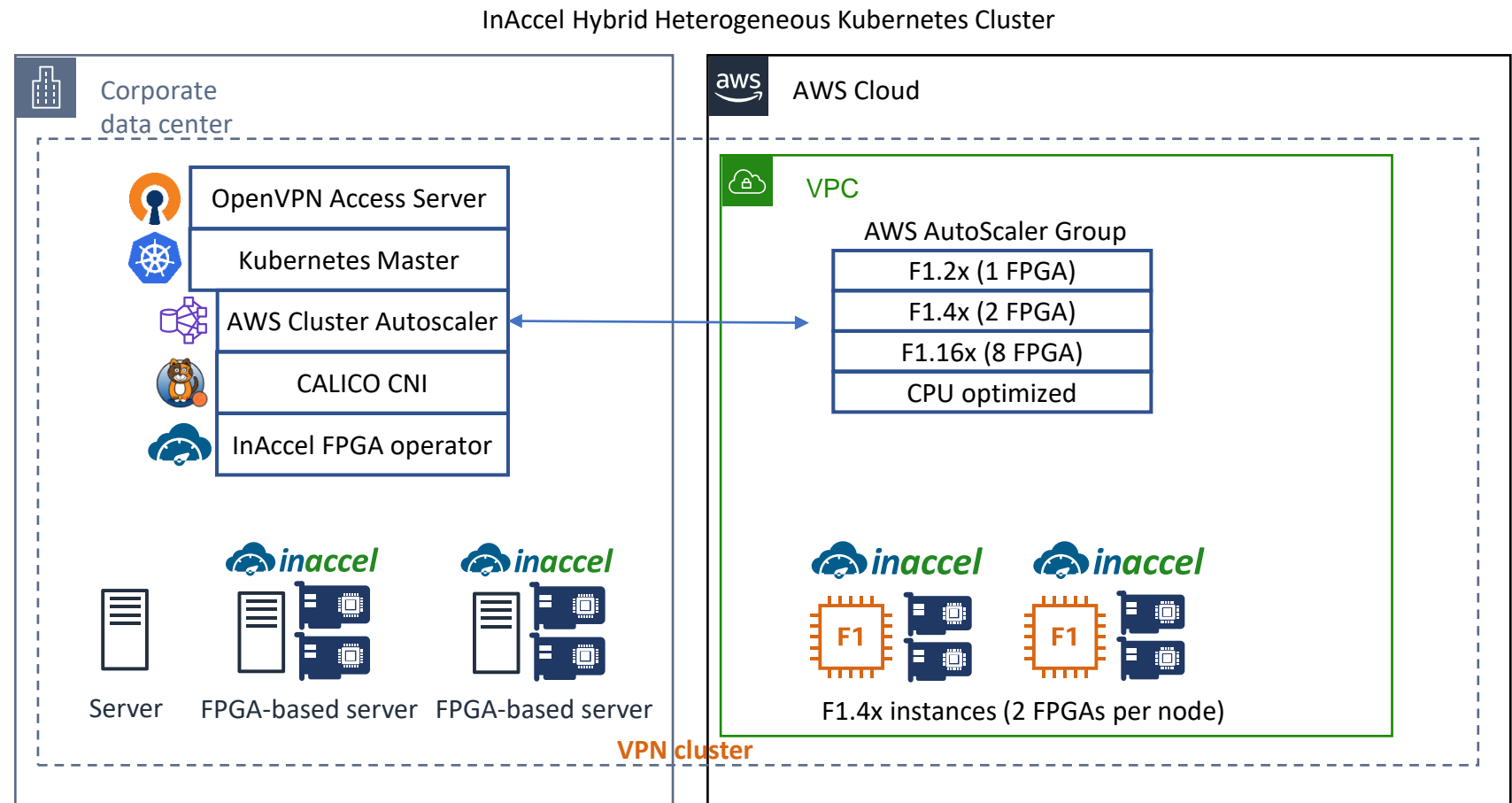
- > How can **scale-out** my application on-prem and on cloud?



Auto-scalable deployment



- > Starting on prem
- > Moving to the cloud
 - >> Automatically
 - >> Instantly



Auto-scalable FPGA deployment



Setup the Master node

1. Initialize the Kubernetes control-plane. Use the VPN IP, that the OpenVPN Access Server has assigned to that node (e.g 172.27.224.1), as the IP address the API Server will advertise it's listening on.

```
sudo kubeadm init \
  --apiserver-advertise-address=172.27.224.1 \
  --kubernetes-version stable-1.18
```

To make `helm` and `kubectl` work for your non-root user, use the commands from the `kubeadm init` output.

2. Deploy **Calico** network policy engine for Kubernetes.

```
kubectl apply -f https://docs.projectcalico.org/v3.14/manifests/calico.yaml
```

3. Deploy **Cluster Autoscaler** for AWS.

```
helm repo add stable https://kubernetes-charts.storage.googleapis.com
helm install cluster-autoscaler stable/cluster-autoscaler \
  --set autoDiscovery.clusterName=InAccel \
  --set awsAccessKeyId=<your-aws-access-key-id> \
  --set awsRegion=us-east-1 \
  --set awsSecretAccessKey=<your-aws-secret-access-key> \
  --set cloudProvider=aws
```

4. Deploy **InAccel FPGA Operator**.

```
helm repo add inaccel https://setup.inaccel.com/helm
helm install inaccel inaccel/fpga-operator \
  --set license=<your-license> \
  --set nodeSelector.inaccel/fpga=enabled
```

Setup the local Worker nodes

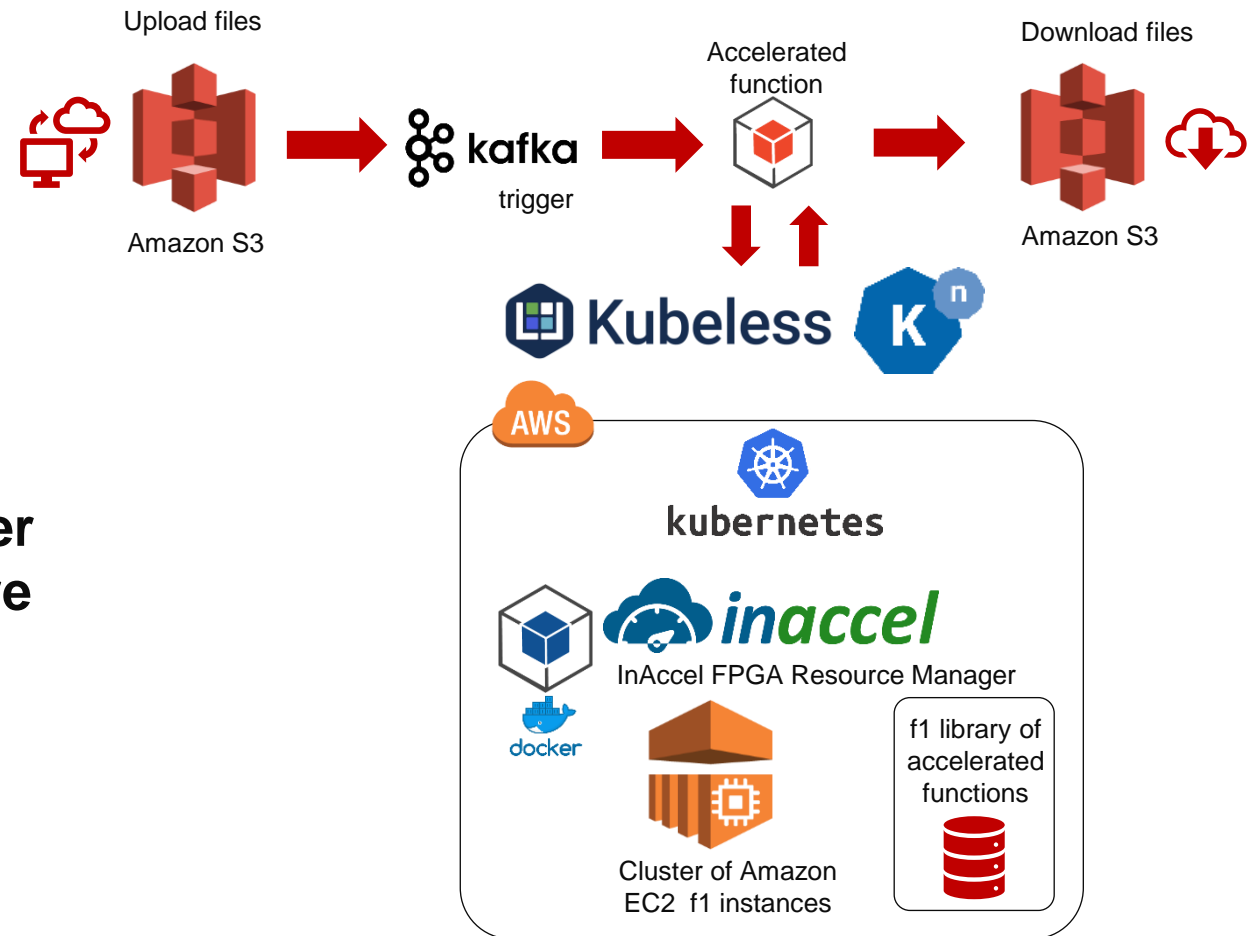


<https://docs.inaccel.com/labs/auto-scaling-aws/>

<https://www.youtube.com/watch?v=CVVyvXY4w5w>

Serverless deployment

- > **Integrated framework for serverless deployment**
- > **Compatible with Kubeless, Knative**
- > Users only have to **upload the images** on the S3 bucket and then InAccel's FPGA Manager **automatically deploy the cluster of FPGAs**, process the data and then **store back the results** on the S3 bucket.
- > Users do not have to know anything about the FPGA execution.



<https://medium.com/@inaccel/fpgas-goes-serverless-on-kubernetes-55c1d39c5e30>

Insight into the FPGA utilization



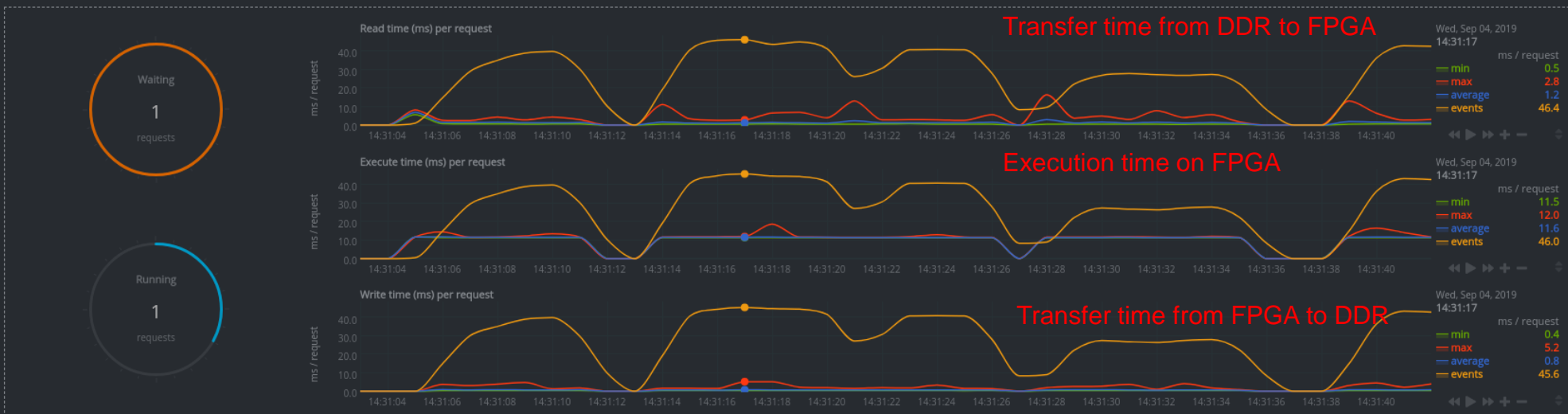
Xilinx 0 Xilinx 1

- com.inaccel.ml.logisticregression.gradients

kernel0

com.Inaccel.ml.logisticregression.gradients (1225630506)

Success: 3151 requests Failure: 0 requests



kernel1

com.Inaccel.ml.logisticregression.gradients (1225630505)

Success: 3445 requests Failure: 0 requests

Keras Deployment on Alveo cards



> Easy deployment of Keras applications



```
pip install inaccel-keras
```

```
import time

from inaccel.keras.applications.resnet50 import ResNet50
from inaccel.keras.preprocessing.image import ImageDataGenerator

model = ResNet50(weights='imagenet')

data = ImageDataGenerator(dtype='int8')
images = data.flow_from_directory('imagenet/', target_size=(224, 224), class_mode=None, batch_size=64)

begin = time.monotonic()
preds = model.predict(images, workers=16)
end = time.monotonic()

print('Duration for', len(preds), 'images: %.3f sec' % (end - begin))
print('FPS: %.3f' % (len(preds) / (end - begin)))
```

2897 fps on U250



<https://docs.inaccel.com/project/keras/>

Quantized ResNet50 on multiple Alveo cards



1 Application => 2 Alveo



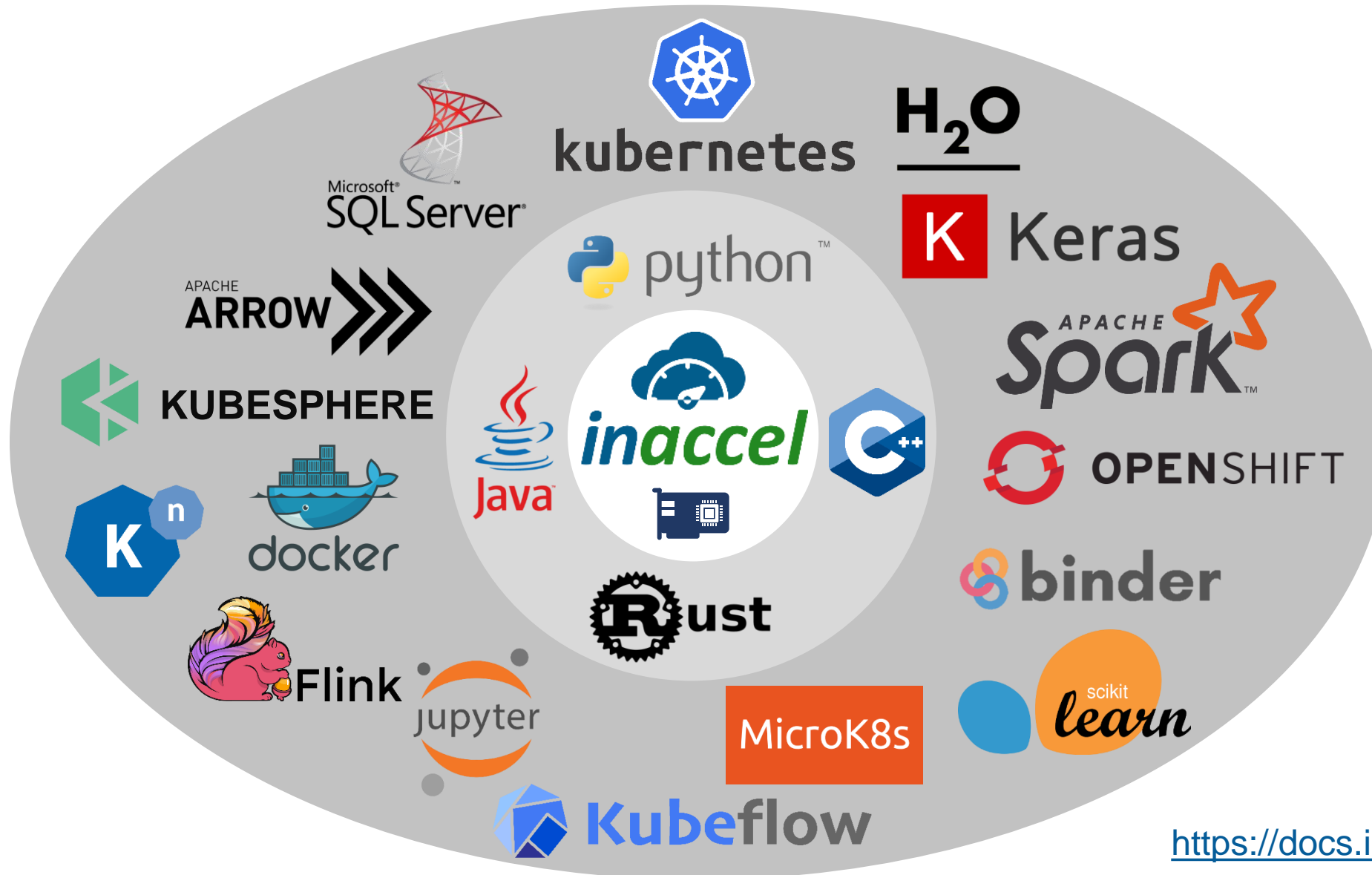
2 Applications => 1 Alveo



2 Applications => 2 Alveo



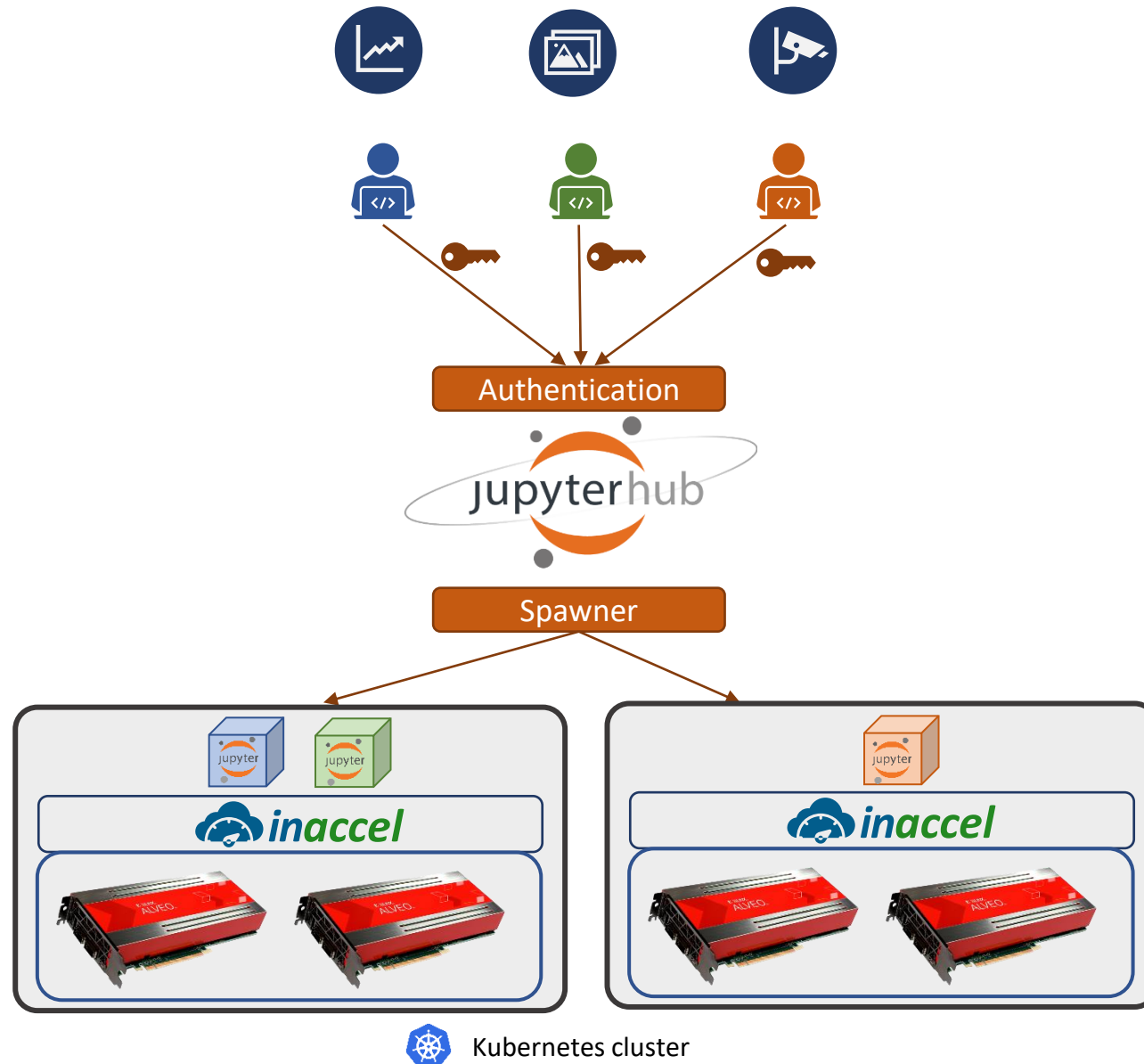
Successful Use cases, Integrations



<https://docs.inaccel.com/>

JupyterHub on FPGAs

- > Instant acceleration of Jupyter Notebooks with zero code-changes
- > Offload the most computational intensive tasks on FPGA-based servers



Applications



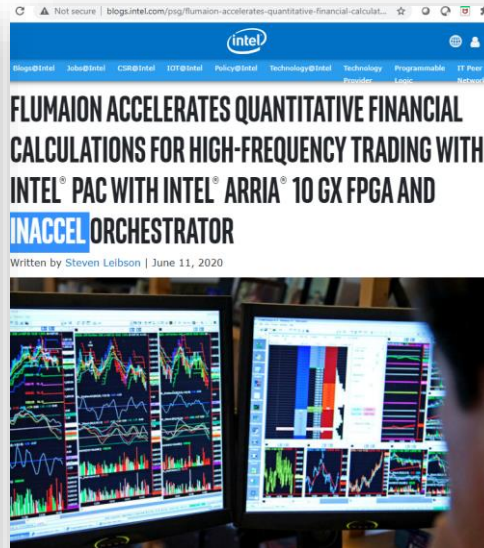
Machine Learning



<https://blogs.intel.com/psg/inaccels-accelerated-ml-suite-boosts-spark-ml-performance-by-as-much-as-7x-on-fpga-based-alibaba-cloud-f1-instances/>



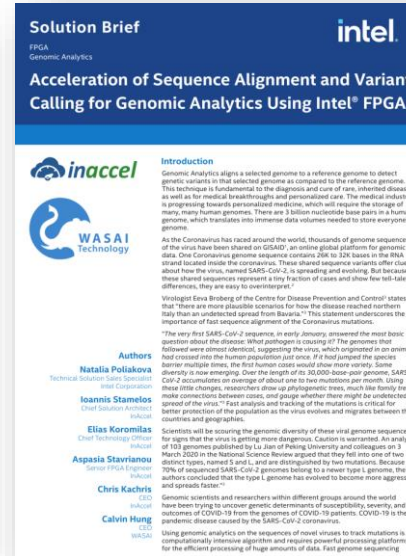
Quantitative Finance



<https://blogs.intel.com/psg/flumaion-accelerates-quantitative-financial-calculations/>



Genomics



<https://www.intel.com/content/dam/www/programmable/us/en/pdfs/lit/erature/solution-sheets/sb-gemomic-analytics-using-intel-fpga.pdf>



Deep Learning



<https://inaccel.com/wp-content/uploads/Inspur-Solution-Brief-Inference.pdf>













Video Analytics



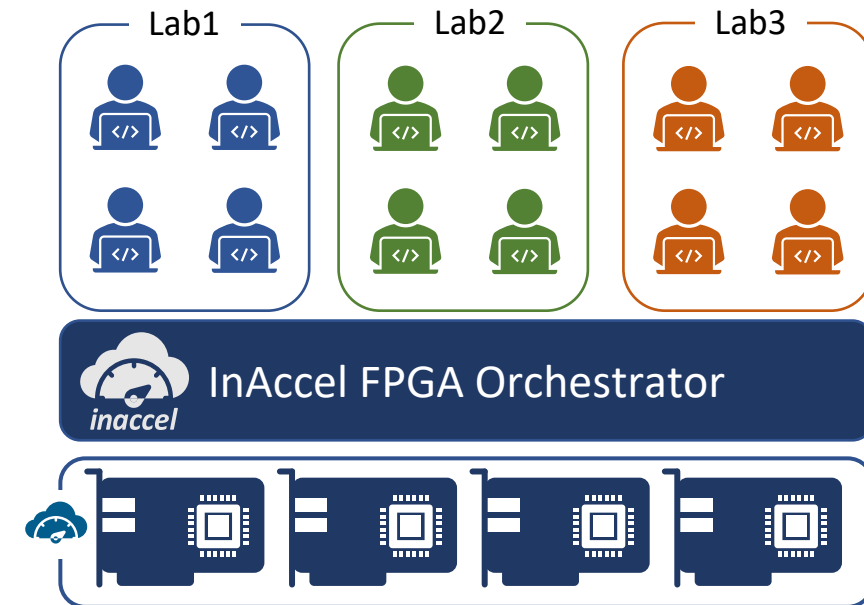
https://inaccel.com/wp-content/uploads/Face-detection_inaccel.pdf

Data Science platforms

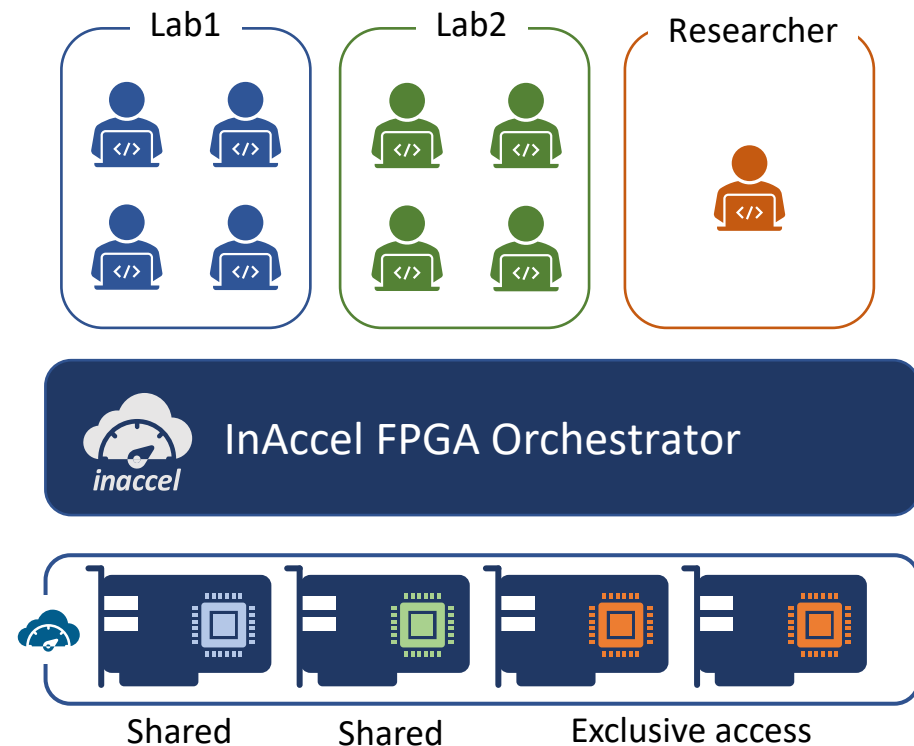
	GPU	FPGA
<i>inaccel</i>		XILINX® intel®
 Azure Notebooks	 NVIDIA®	
 Amazon SageMaker	 NVIDIA®	
 colab	 NVIDIA®	
 DataCamp	 NVIDIA®	
 kaggle	 NVIDIA®	



- > **How do you allow multiple students to share the available FPGAs?**
- > Many universities have limited number of FPGA cards that want to share with multiple students.
- > InAccel FPGA orchestrator allows multiple students to share one or more FPGAs seamlessly.
- > It allows students to just invoke the function that want to accelerate and InAccel FPGA manager performs the serialization and the scheduling of the functions to the available FPGA resources.



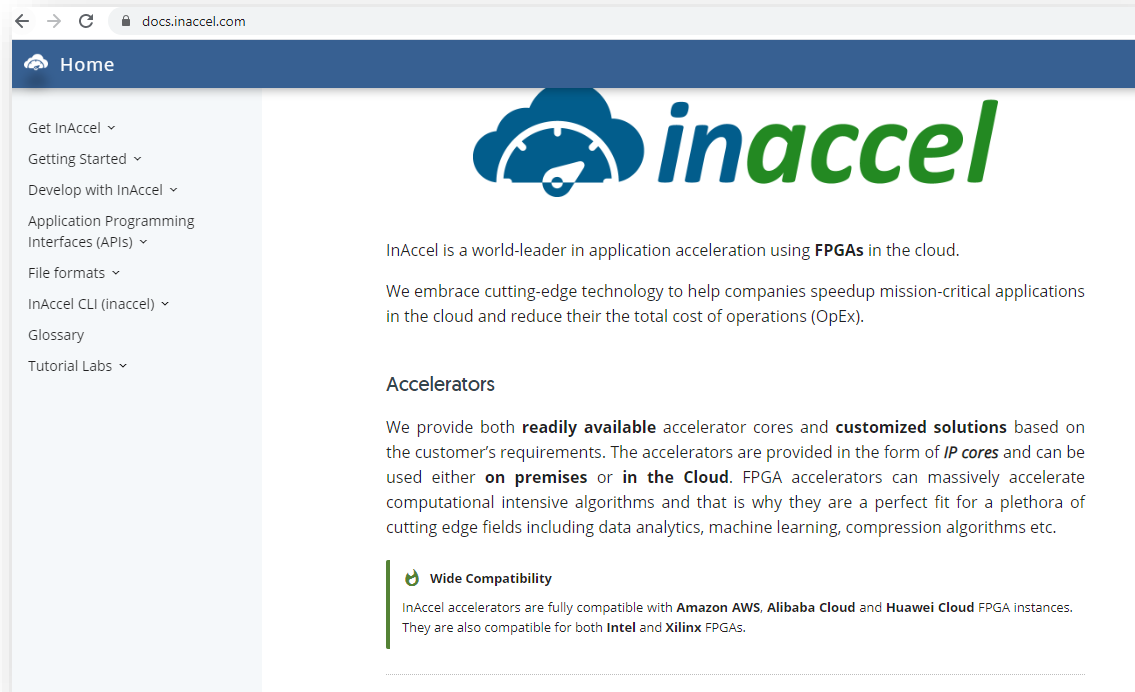
- > **But the researchers want exclusive access**
- > InAccel orchestrator allows to select which FPGA cards will be available for multiple students and which FPGAs can be allocated exclusively to researchers and Ph.D. students (so they can get accurate measurements for their papers).
- > The FPGAs that are shared with multiple students will perform on a best-effort approach (InAccel manager performs the serialization of the requested access) while the researchers have exclusive access to the FPGAs with zero overhead.



Test it on your prem or on your browser

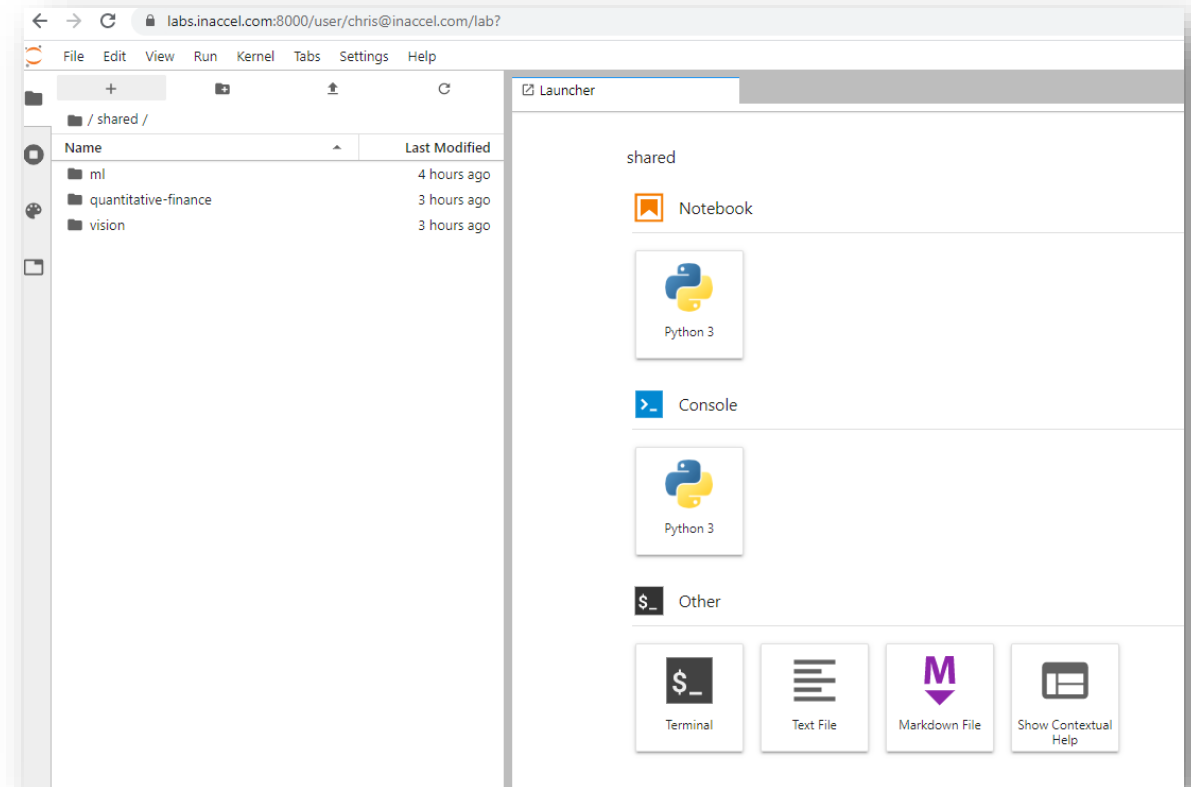


On-prem



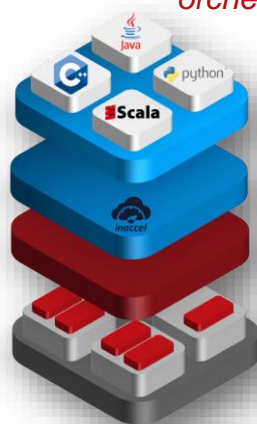
<https://docs.inaccel.com/>

Online - Browser

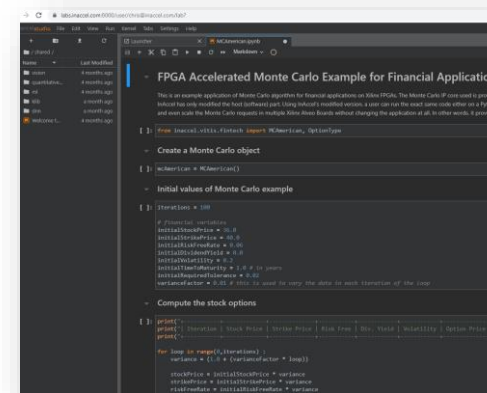
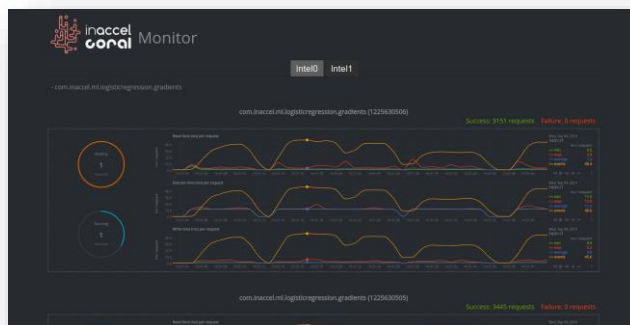


<https://studio.inaccel.com>

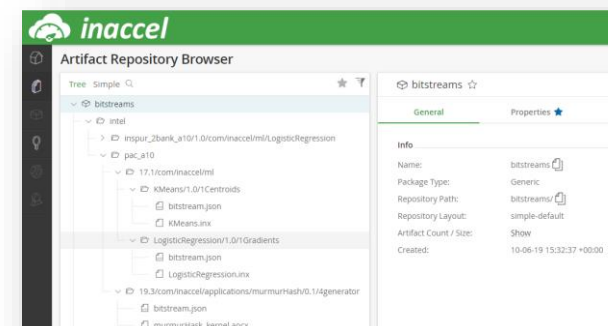
InAccel solutions



<https://inaccel.com/fpga-manager/>



<https://studio.inaccel.com>



<https://store.inaccel.com>

InAccel, Inc. Corporate overview

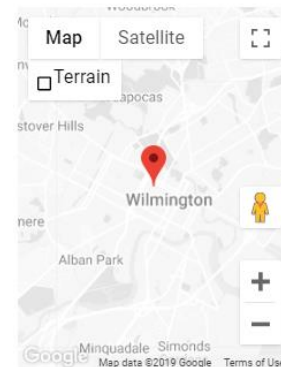


- > Founded in January 2018
- > Registered in Delaware, USA

> Membership:



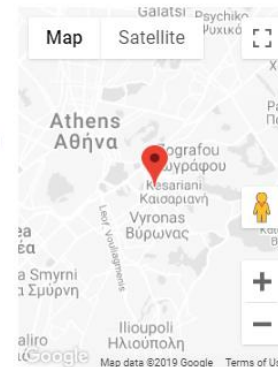
Registered
Technology
Partner



Headquarters

500 Delaware Ave STE 1, #1960
Wilmington, DE 19801
USA

(+1) 408 260 5724



Design Center

Formionos 47
Kesariani 116 33
Athens, Greece

(+30) 211 1825 436





Application Acceleration, seamlessly

www.inaccel.com

info@inaccel.com

USA:

500 Delaware Ave STE 1, #1960
Wilmington, DE 19801
USA

Europe (Design Center):

Formionos 47
Kesariani 116 33
Athens, Greece