STORAGE DEVELOPER CONFERENCE
SDC 22
Fremont, CA
September 12-15, 2022
BY Developers FOR Developers
A SNIA Event

Accelerating FaaS/Container Image Construction via IPU

Presented by

Ziye Yang, Cloud Software Engineer, Intel
Yadong Li, Principal Engineer, Intel
Zeng Jun, Cloud Software Engineer, Intel

Agenda

- Motivation and Background
- IPU Introduction & Use of IPU in FaaS/Container Acceleration
- Detailed Design for Accelerating FaaS/Container Image Operations using an IPU
- A Simple RunC Example to Illustrate the Design and Flows
- Summary

Motivation and Background

Motivation & Background

- Serverless computing & containers is one of the 4 emerging cloud-native trends, according to Gartner (posted in CNCF' blog).
  - Gartner analysts foresee the development of serverless computing in particular, i.e. function-as-a-service (FaaS).
  - FaaS service are usually deployed in containers.
- What is FaaS? (From Wikipedia)
  - A category of cloud computing services that provides a platform allowing customers to develop, run, and manage application functionalities without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app

Motivation & Background (cont.)

- FaaS are usually the short programs (functions), it's critical to execute such a function in a fast way without any unnecessary overhead of preparations.
- The main overhead comes from compiling the code into executable binary, pack the executable binary with required libraries into a file system, and then get the container environment up running.
- For FaaS deployed in containers, the following are the main places for optimizations to reduce the overhead of preparations:
  1. Quickly build the FaaS/Container related images.
  2. Get container execution environment ready asap, including unpacking the image of FaaS into a file system.
  3. Expose the bundle to the container, then execute the FaaS applications in the selected container.

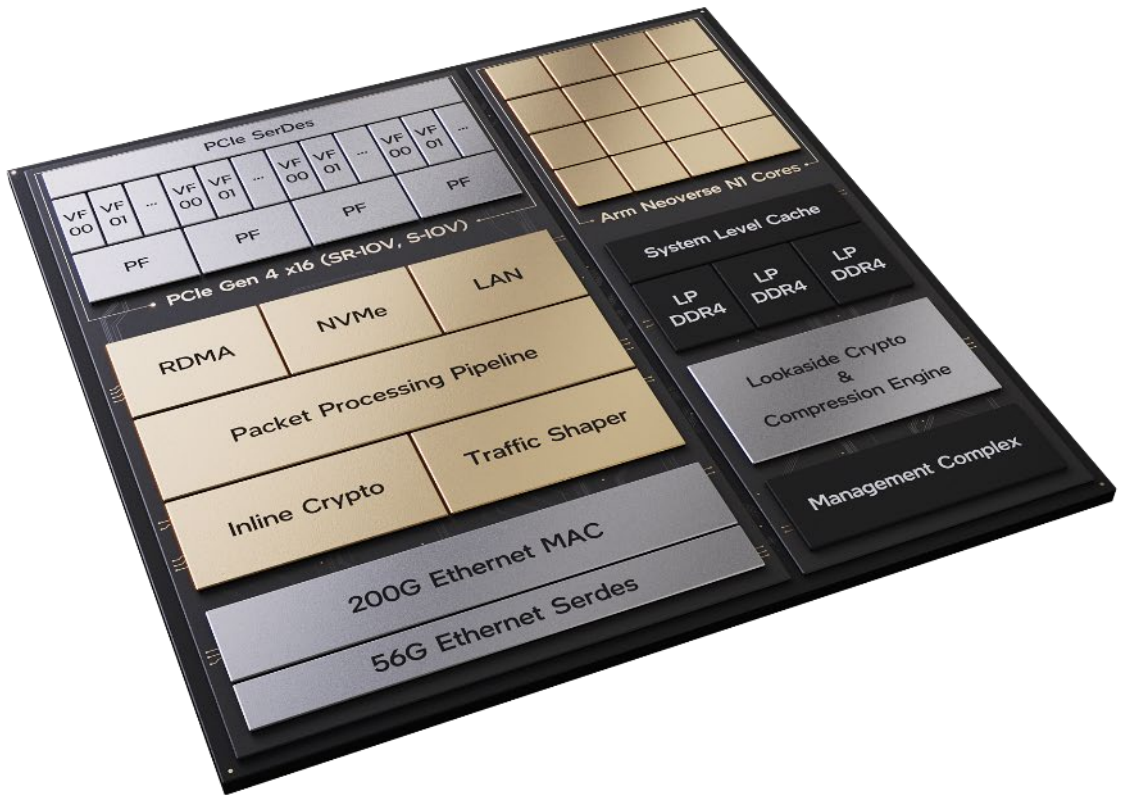
Motivation & Background (cont.)

- As noted earlier, one of the main overhead of container comes from the preparation work, including
  - **Image pulling & file system bundle (e.g., rootfs) preparation**.
  - Start runtime shim.
  - The selected runtime class started to run (e.g., RUNC).
- We think IPU can help here. IPU can be used to accelerate container image pulling & file system bundle preparation.
  - The container image related operations can be moved into IPU.
  - IPU accelerators can be used for image decompression, decryption, etc.
  - IPU can cache the images and enable sharing of the unpacked image layers.

IPU Introduction & Use of IPU in FaaS/Container Acceleration

Mount Evans

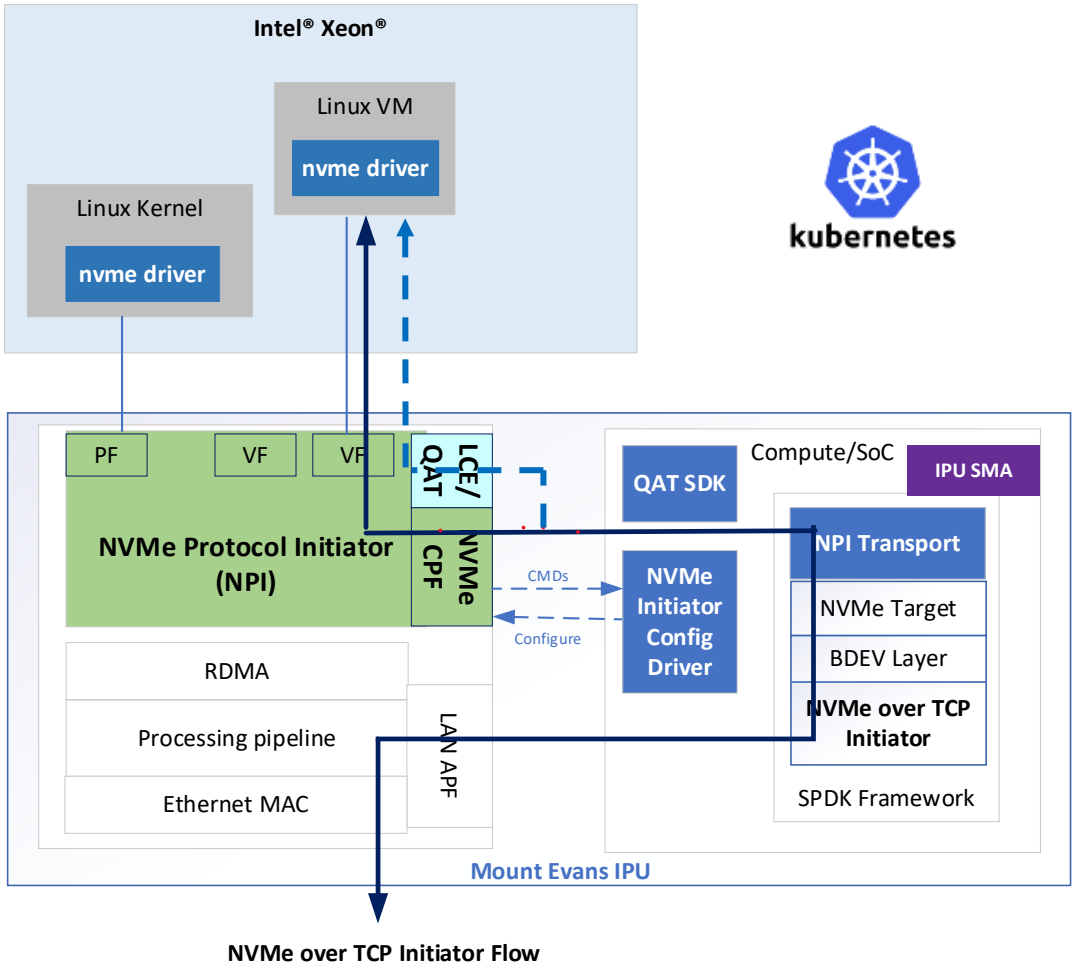
Intel's 200G IPU



Hyperscale Ready	Co-designed with **Google** Integrated learnings from multiple gen. of FPGA sNIC/IPU High performance under real world load Security and isolation from the ground up
Technology Innovation	Best-in-Class Programmable Packet Processing Engine NVMe storage interface scaled up from Intel Optane Tech Next Generation Reliable Transport Advanced crypto and compression accel.
Software	SW/HW/Accel co-design P4 Studio based on Barefoot technology Linux OS leveraging DPDK, SPDK & IPDK eco-systems VMWare's Project Monterey for telco & enterprise

Mount Evans NVMe over TCP Initiator

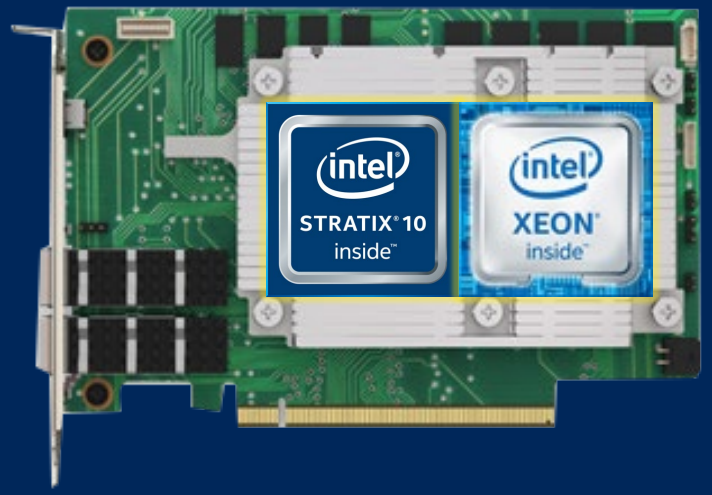
- NVMe HW (NPI) provides Host NVMe virtualization layer
- Store-and-forward data flow
  - NPI auto fetches NVMe CMDs into SoC memory.
  - SoC SW uses **QAT/LCE DMA engine** to fetch PRP lists and move data payloads.
  - **As part of the DMA flow, QAT/LCE chained ops available for compression, crypto and CRC offloads**.
- Fully integrated with **IPDK** and **SPDK** NVMe-oF SW stack
  - NPI Transport to interface with NVMf layer
  - NPI Transport uses NVMe Initiator Config Driver for device configurations and CMDs processing.
  - **IPU Storage Management Agent (SMA)** from SPDK provides a consistent and simplified API for integration with orchestration frameworks.



Big Spring Cannon (BSC) Introduction (Hardware)

Features

- Intel® Xeon® D processor and FPGA SmartNIC platform
- Virtual networking and storage combined
- Software and hardware programmable



- 2*25 Gbps SmartNIC Platform
- Intel® Xeon® D processor
- Intel® Stratix® 10 DX FPGA
- ½ length, full height PCIe card
- 75W TDP for key workloads

Benefits

- Large software ecosystem
- High performance
- Versatile and efficient
- Open Vswitch (OvS) with virtio-net, virtio-blk
- Resilient to future change
- Customizable

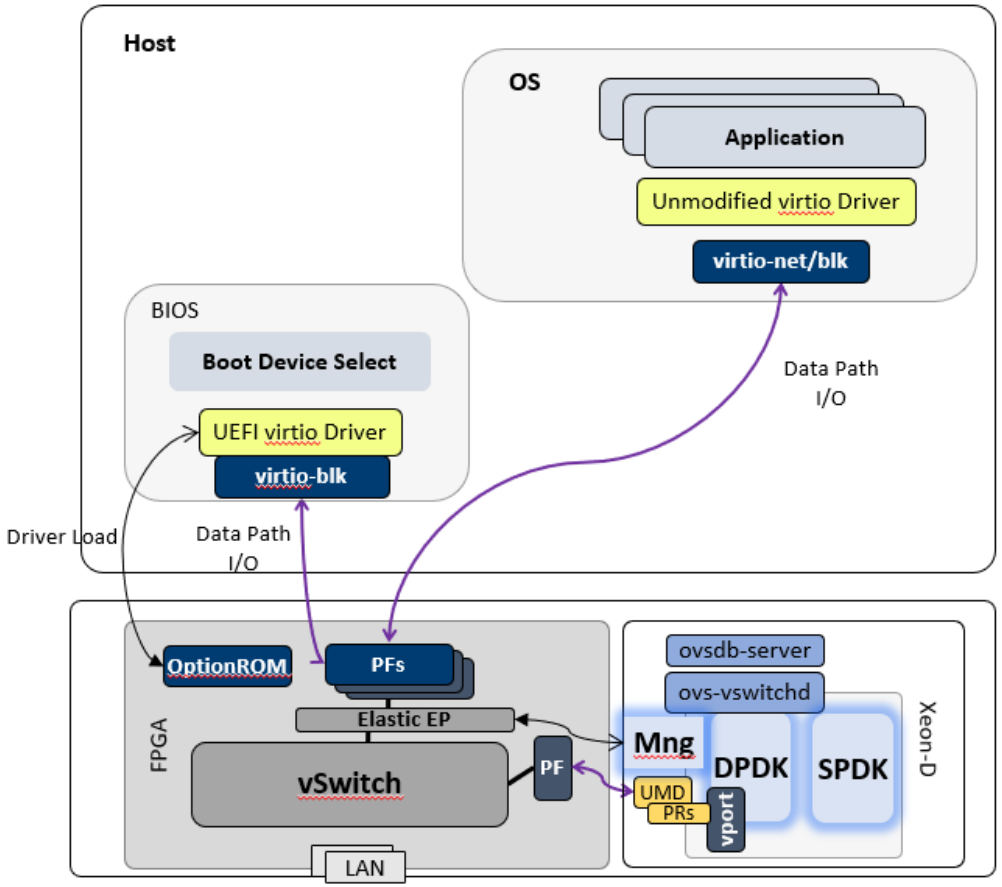
BSC Introduction Bare Metal (Software)

Host SW

- Converged virtio-net/blk vdpa driver
- UEFI OptionROM virtio driver probe and boot
- SPDK backend for cloud remote boot
- Elastic PFs (device hotplug)

BSC

- Virtio-net/blk FPGA back end
- Host-SoC bridge with FPGA
- Elastic Host PFs hotplug by Elastic EP IP
- Bare metal management SW package
- OptionROM for host UEFI virtio-net/blk driver

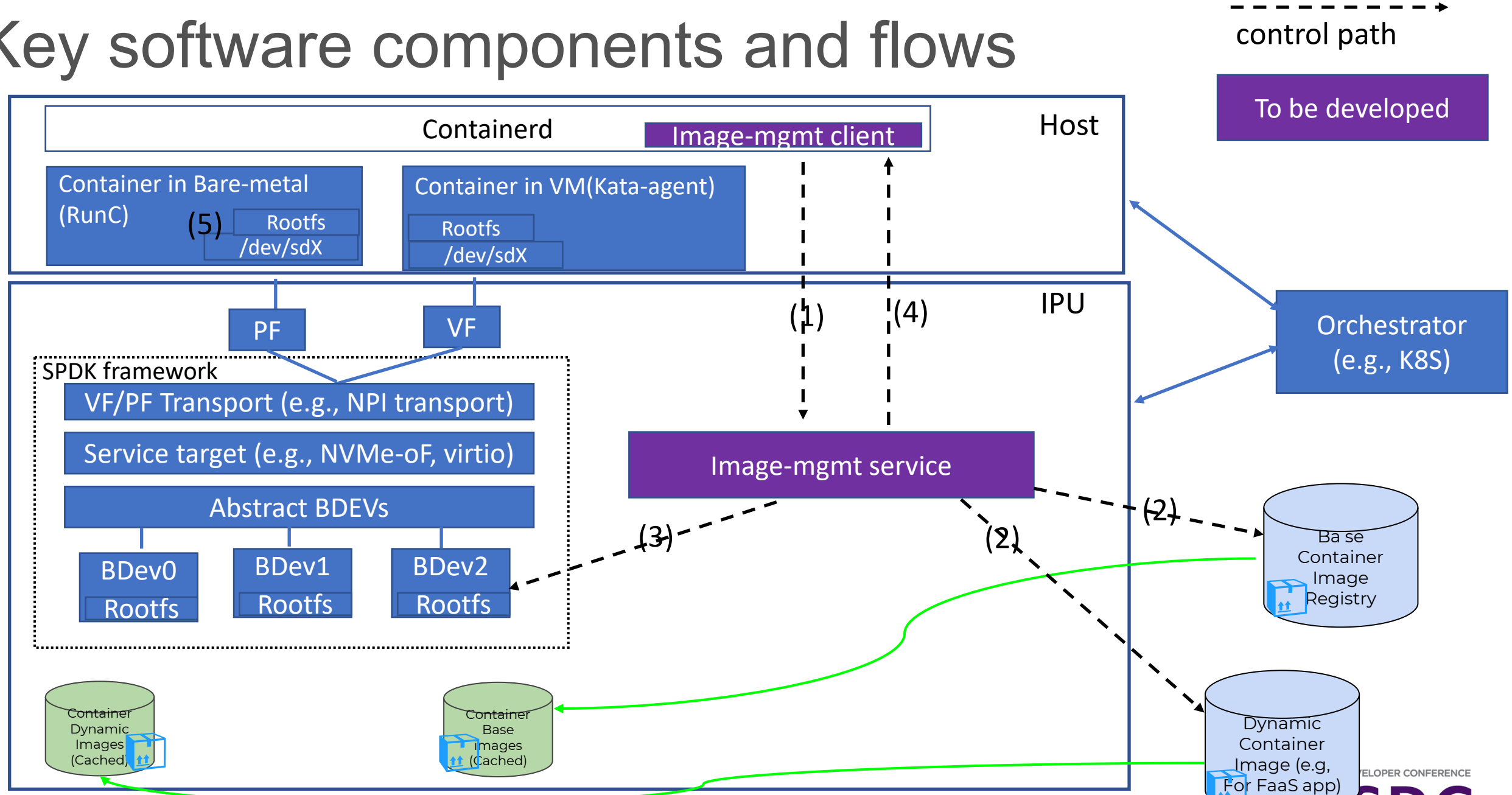


Ingredients for the Solution

Ingredient	IPU Support	Benefit from IPU based solution
Block Device Interface to Host	NVMe, virtio-blk interfaces	Fully leverage on IPU based storage disaggregation solution
Block device Hot-plug	Yes, IPU designed for bare-metal and virtualization usages	Align with IPU/DPU's long term strategy as a control point in data center
Container image pulling & caching	Download container images from image registry and cache images	IPU as a control point is the idea choice to manage container images
Container Rootfs construction	Construct the rootfs in an assigned bdev provided by the block service target in the IPU	IPU can offload such work from the host
Unpacking rootfs or sharing filesystem among containers	Leverage the snapshot or cloning features of the bdev in block service target are required.	IPU can offload such work efficiently because of integrated accelerators such as decompress engines.
Control path communication between IPU and container runtime software	Provide related RPC service to interact with container management software.	Such RPC service is supported in IPU's architecture and design

Detailed Design for Accelerating FaaS/Container Image Operations using an IPU

Key software components and flows



Steps to construct FaaS/container images

- There will be the following steps to **construct the execution environment** for FaaS/Container usage scenario:
  1) The Orchestrator (e.g., K8S/containerd/runC|Kata Containers) communicates with the IPU after receiving the FaaS running request. Then the IPU receives the requirements from upper running software in the host.
  2) The image-mgnt service downloads the dynamic images and base image from each image registry
  3) The image-mgnt service unpacks the images into an assigned Bdev (e.g., Bdev2 in the diagram)
  4) Then the virtual block device target service (e.g., NVMe-oF, virtio) with designated transports (e.g., NPI for NVMe) exports this bdev via a VF or PF to the host. And IPU notifies the container software via RPC.
  5) When the VM or host kernel sees the VF/PF, it finally gets a block device after loading the related device drivers. Then container management software mounts the block device (e.g., /dev/sdX) to a specific mounting point.
- Finally, containers can be started with the rootfs bundle contained in the block device

Steps to destroy FaaS/container images by IPU

- There are the following steps to destroy the resources when the IPU receives the request from the host,
  1) Containerd like software communicates with the image-mgnt service in IPU that the container is shutdown.
  2) Image-mgnt service in IPU communicates with block service target to locate the bdev which used by the container.
  3) Block service target in IPU hot removes the VF/PF device related with the bdev to the host, and there is an event sent to host. Meanwhile, it also destroys the bdev.

A Simple runC Example to Illustrate the Design and Flows

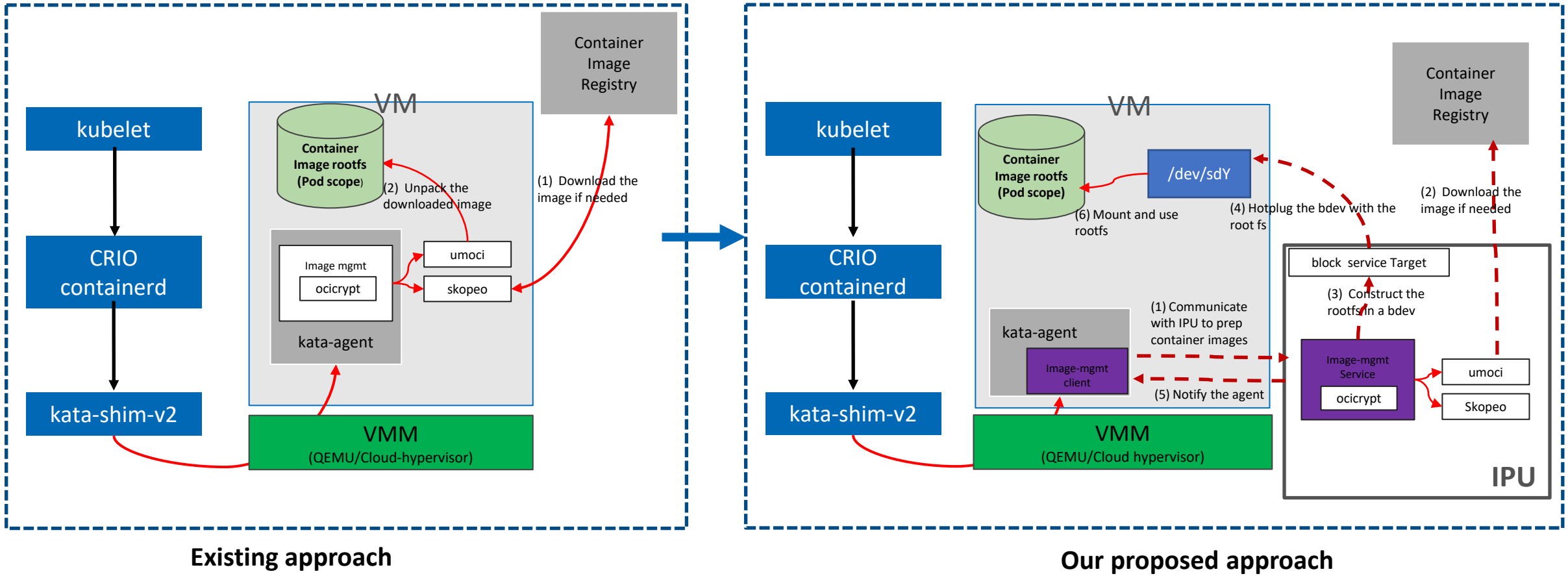
3 steps to run a simple container by RunC in a host

- RunC: a CLI tool for spawning and running containers on Linux according to the OCI specification.
- We use **busybox:latest** container image as an example
- The following shows the 3 steps to start a container in a host:
  1. # **skopeo** copy docker://busybox:latest oci:busybox:latest
  2. # **umoci** unpack --image busybox:latest **busybox_bundle (destination folder)**
  3. # **runc** run test --bundle busybox_bundle

4 steps in IPU to run a simple container by RunC

- Step1: Download or cache the images in the IPU. We can still use **skopeo** to download the images and cache it.
- Step2: Unpack the images into a bdev exported by the block service target.
  - Then the rootfs can be viewed and operated in the host side. For bdevs with **snapshot** and **clone** feature, it can be helpful to share basic root file systems among different containers.
- Step3: Hotplug bdev into host and notify container management software.
  - The container management software can view the rootfs folder inside the block device after mounting.
- Step4: Use RunC to run the container with the rootfs in the mounting point.

Work in progress: Container image offloading with Kata Containers



Summary

Summary

- In this presentation, we proposed that FaaS/container images' construction operations (e.g., **Image pulling** and **rootfs preparation**) can be offloaded to IPU, which saves host CPU/memory resources.
- IPU based FaaS image construction acceleration moves the overhead from host to IPU. It opens the opportunities for optimizations and innovations.
  - To illustrate details of our idea, we did a quick PoC with Intel's IPU to prove our FaaS/container image offloading idea.
- We plan to continue the development work:
  - Development of the control path between the IPU side (e.g., RPC service) and the Container Runtime software components in the host.
  - Performance evaluation and optimizations.

Please take a moment to rate this session.

Your feedback is important to us.

Appendix

Detailed steps and scripts for a running example

Leverage BSC to offloading image operations

- Leverage BSC card (An FPGA based IPU) to export virtio_blk PF/VF to host side
  1. When the BSC receives the request, it creates vhost ctrlrs on BSC card.
  2. We can still use the skopeo and umoci to download the image, and put unpacked container image file into a formatted lvol_bdev which created from SPDK running on BSC through NBD feature.
  3. Create snapshot based on the previous lvol_bdev and then make clones from it (For sharing purpose.)
  4. Use the cloned bdevs to server as backend storages for vhost ctrlrs
- Then the host leverages the block devices initialized from virtio-blk PF/VF as Container images, e.g.,
  - Mount the bdevs created from virtio-blk PF/VF, and can find container images under the mounted directories

Key steps to construct rootfs of a container in BSC

- Config and start the SPDK based block target in BSC
  - Start the related block target
  - Open management interface of blk related device
  - Construct blk related vhost device
- Create lvol bdev and copy unpacked container image
  - Create lvol bdev and export it through NBD mechanism
  - Format the nbd block device and mount it to a folder in order to copy downloaded/cached container image
  - Create snapshot from the lvol bdev and then create clone from snapshot
- Map lvol bdev as backend storage for a blk device exported by the block target.
  - For example, Map the clone bdev to specific port of blk device

Scripts on BSC side

- Create blk related device (example code with rpc.py script)
- Create lvol bdev and export it through NBD (network block devices) mechanism
  - dd if=/dev/zero of=image_test.file bs= 1M count = 512
  - ./scripts/rpc.py scripts/rpc.py bdev_aio_create image_test.file aio0 4096
  - ./scripts/rpc.py scripts/rpc.py bdev_lvol_create_lvstore aio0 lvol0
  - ./scripts/rpc.py scripts/rpc.py bdev_lvol_create -l lvol0 bdev_lvol0 768
  - ./scripts/rpc.py scripts/rpc.py nbd_start_disk lvol0/bdev_lvol0 /dev/nbd0

Scripts on BSC side (Cont.)

- Format the nbd block device and copy unpacked container image
  - mkfs -t ext4 /dev/nbd0
  - mkdir /mnt/test_for_nbd
  - mount /dev/nbd0 /mnt/test_for_nbd/
  - cp -r busybox_bundle /mnt/test_for_nbd/
  - ./scripts/rpc.py nbd_stop_disk /dev/nbd0
- Create snapshot and clone from lvol bdev
  - ./scripts/rpc.py bdev_lvol_snapshot lvol0/bdev_lvol0 snap_bdev_lvol0
  - ./scripts/rpc.py bdev_lvol_clone lvol0/snap_bdev_lvol0 clon0
  - ./scripts/rpc.py bdev_lvol_clone lvol0/snap_bdev_lvol0 clon1
  - ./scripts/rpc.py bdev_lvol_clone lvol0/snap_bdev_lvol0 clon2
  - //The 3 bdevs can be used by 3 different containers. Map cloned bdevs to different port of blk related devices

Steps to use virtio-blk PF/VF on host side

- When the container runtime software is notified that the rootfs of the container is prepared, then
  - Modprobe virtio_blk driver in the kernel to initialize block device
  - Create VF from virtio_blk PF and initialize block device from the VF if need to use in VM.
  - Mount the obtained block device and use the container image in mounted directories

Scripts on Host side

- When the container runtime software on host are notified that the image are prepared
  - List virtio-blk PF provided by BSC
    - lspci | grep -i virtio
  - Probe virtio-blk driver and create VFs
    - # modprobe virtio_blk
    - # echo 2 > /sys/bus/pci/devices/0000\:18\:00.3/sriov_numvfs
  - Select of the device, then mount block devices initialized from virtio-blk PF/VF
    - # mount /dev/vdc /mnt/test_for_runc/
  - Use runC to run container image
    - # runc run test --bundle /mnt/test_for_runc/busy_bundle/
    - We could mount block devices initialized from PF and VFs separately and use them independently

