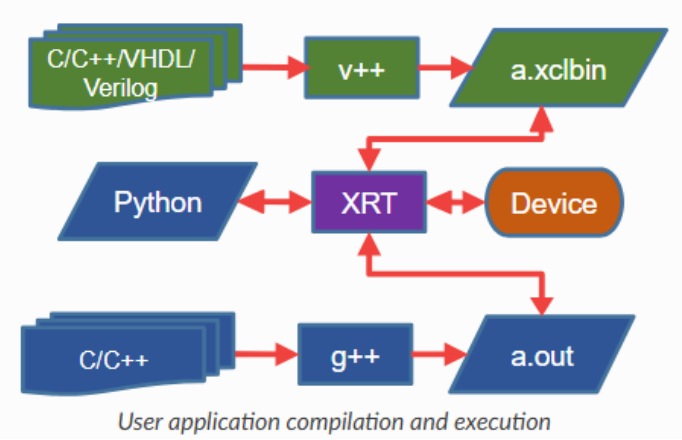
XRT guide

Vitis Platform Overview

User Application Compilation (PL, PR region configuration)



green: device code, blue: host code(C/C++ or Python OpenCL/XRT)

Different Regions in FPGA

PL(Programmable Logic)

a. 사용자가 원하는 대로 프로그래밍할 수 있는 영역

b. LUTs, FFs, DSP, BRAM ...

PR(Partial Reconfiguration)

1. PL의 일부분
2. 동적으로 재구성하여 리소스 최적화 또는 시스템 유연성 증가
3. Runtime configurable

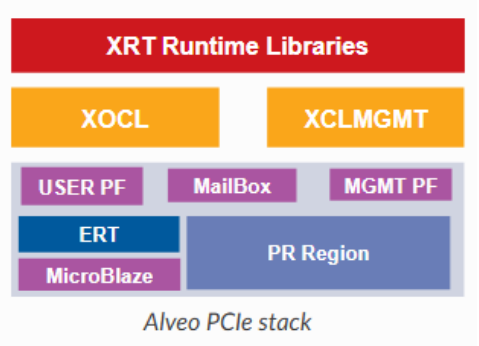
PS(Processing System)

1. e.g. ARM processor

PF(Platform Firmware)

1. Booting process
2. Basic infrastructures
3. Interface between Linux kernel driver and User Partition(or PL, PS)

PCIe based platforms(U280)



Alveo PCIe stack

Shell Partition

Provides basic infrastructure for the Alveo platform

- **PF**0(MGMTPF): Management physical function, access to privileged operations
  - XRT의 Linux kernel driver인 xclmgmt와 binded.
  - Downloading user compiled FPGA image (xclbin)
  - Loading firmware container (xsabin)
  - Access sensors
  - AXI firewall(security)
  - Communication with user PF through Mailbox
  - Interrupt handling
  - ...
- **PF**1(User PF) : Access to non-privileged operation, compute units in user partition
  - XRT Linux kernel driver xocl에 binding됨 → xrt.h에 정의됨
  - PCIe DMA engine 설정 및 관리
  - Memory-to-memory : DDR, PL-RAM, HBM
  - 버퍼 공유

- Compute units scheduling and pipeline management → ERT(Embedded Runtime) hardware scheduler 사용

  → ERT가 없으면 software 수준에서 xocl 드라이버에 의해 처리

User(PR-Region) Partition

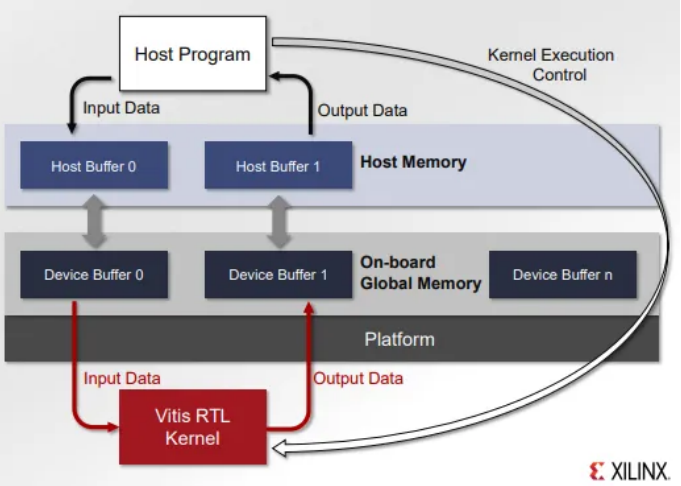
a. Contains user compiled binary

b. Dynamic Function Exchange(DFX) : load user compiled binary to the User partition

---

Execution Flow

- XOCL Driver Interfaces ↔ PCIe User Physical Function
- XCLMGMT Driver Interfaces ↔ PCIe Management Physical Function



1. Image Download

Prerequisite: Compiles and links user's device code(RTL, c/c++) into xclbin with v++

1. Open a device
2. Load the xclbin file → xclmgmt drivers walk the xclbin
   - load bitstream on PL
   - read memory topology and initialize the memory manager
   - discover compute unit
3. FPGA Manager integeration

XRT Native API for Image Download

`xrt::device` and `xrt::xclbin`

1. Open a device

```
unsigned int dev_index = 0;
auto device = xrt::device(dev_index); // open a device
```

다음과 같이 PCIe BDF를 통해 선언하는 방법도 있다. (BDF(bus, device, function): PCI 장치의 식별번호)

```
auto device = xrt::device("0000:03:00.1");
```

2. Load the xclbin

```
auto xclbin_uuid = device.load_xclbin("kernel.xclbin"); // load the xclbin file from the filename
```

- `xrt::device::load_xclbin` returns the XCLBIN UUID, which is required to open the kernel

3. Create Kernel
   a. 모든 CU(compute unit)가 동일한 interface에 연결되어 있을 때

```
auto krnl = xrt::kernel(device, xclbin_uuid, name);
```

   b. CU가 각각 다른 interface와 연결되어 있을 때

   → 각 interface에 해당하는 CU를 따로 선언해줘야 한다.

   e.g. kernel name `foo` 에 세 CU `foo1, foo2, foo3` 가 있고, `foo1, foo2` 가 같은 interface, `foo3`가 다른 interface에 연결되어 있다고 하자

```
auto krnl_obj_1_2 = xrt::kernel(device, xclbin_uuid, "foo:{foo_1,foo_2}");
auto krnl_obj_3 = xrt::kernel(device, xclbin_uuid, "foo:{foo_3)");
```

> 💡 `xbutill`을 통한 CU 확인
>
> Xilinx의 `xbutil` **명령**을 사용하여 FPGA 장치에 배치된 CU를 확인할 수 있다.

```
$ xbutil examine -r compute-units
Compute Units
Name      Base Address   Status
--------- -------------  ------
foo_1     0x00000000     Ready
foo_2     0x00001000     Ready
```

---

2. Memory Management

- Inside Linux kernel driver
- Using DRM GEM framework
- Buffer allocation
- Buffer mmap
- Reference Counting
- DMA-BUF export/import

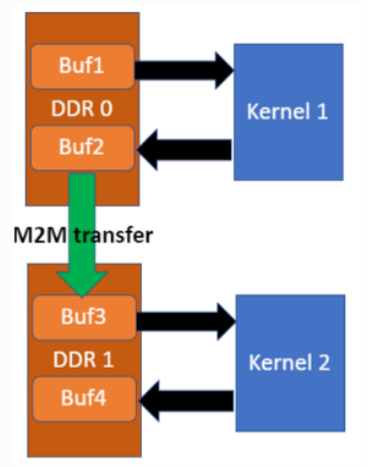
PCIe Peer-to-Peer(P2P)

- Host RAM 사용 없이 PCIe device간 직접적인 data transfer
  - Card0 의 DDR/HBM ↔ card1의 DDR/HBM
  - Card0 의 DDR/HBM ↔ thirdparty peer device(NVMe)

  Todo

Memory-to-Memory(M2M)

- Alveo card 내의 메모리 간의 data transfer가 가능하다.
- e.g) xclbin 내의 두 커널 K1과 K2가 있고, 각각 DDR0과 DDR1에 연결되어 있다. K1의 output이 K2의 input일 때, DDR0에서 DDR1로 data transfer를 해야한다.



Buf1 → K1 → Buf2 → Buf3 → K2 → Buf4

```
//using OpenCL API
clEnqueueTask(queue, K1 ,0,nullptr,&events1); //K1 execution
clEnqueueCopyBuffer(queue, Buf2,Buf3,0,0,Buffer_size, 1, &event1, &event2); //Data transfer from Bu
f2 to Buf3
clEnqueueTask(queue,K2,1,&event2,nullptr); //K2 execution
```

- 다음과 같은 제한 사항이 있다.
  - DDR bank 사이의 M2M을 지원하지만, HBM이나 PLRAM에 대해서는 지원하지 않는다.
  - copy되는 data는 64 bit aligned 되어 있어야 하며, 그렇지 않은 경우 host를 통해 전송된다.
  - M2M copy of OpenCL sub-buffers is not properly supported

Host Memory Access

- Kernel이 card 내부 DDR, HBM을 경유하지 않고 직접 host memory에 접근하여 읽거나 쓸 수 있다.
- 기존 XDMA(PCIe DMA)는 PCIe를 통해 Host memory → FPGA global memory를 거쳐 kernel이 이 global memory에 접근하는 방식
- DMA가 불가능한 플랫폼(No-DMA)에서 특히 유용하다.

1. Kernel compilation 시 다음과 같이 v++ configuration option을 추가해 data가 AXI bridge를 통해 host memory로 전달될 수 있게끔한다.

```
## v++ configuration

[connectivity]
## Syntax
sp=my_kernel_1.m_axi_gmem:HOST[0]
```

2. Host Server Setup

   a. Hugepage Requirement(Optional)

   1GB보다 큰 host memory가 필요하다면, xrt는 host memory Hugepage를 allocate한다. 이 Hugepage는 FPGA shell 내부에서 remap되어 kernel은 contiguous bank-like memory를 보게 된다.

   ▼ Steps required to enable Hugepages

   1. `/etc/default/grub`을 다음과 같이 수정한다.
      `GRUB_CMDLINE_LINUX_DEFAULT="splash quiet noresume hugepagesz=1G hugepages=4"`
   2. `shell>update-grub`
   3. Reboot the server
   4. Verify the HugePage setting

```
shell>hugeadm --pool-list

  Size       Minimum  Current  Maximum  Default
  2097152       0       0       0       *
  1073741824    4       4       4
```

   b. Enabling the Host Memory by XRT

   - **xclbin을 load하기 전** `xbutil configure --host-mem` 명령어를 통해 kernel을 위한 host memory를 reserve해야한다.
   - 다음은 세 카드에 대해 각각 1G, 4G, 16G의 host memory를 reserve한 예시이다.

```
sudo xbutil configure --host-mem -d 0000:a6:00.1 --size 1G enable
sudo xbutil configure --host-mem -d 0000:73:00.1 --size 4G enable
sudo xbutil configure --host-mem -d 0000:17:00.1 --size 16G enable
```

   c. Maximum Host memory supported by the platform

   - Max Shared Host Memory: 플랫폼이 지원하는 최대 호스트 메모리 크기
   - Shared Host Memory: 카드에 할당된 호스트 메모리(`xbutil configure --host-mem`)
   - Enabled Host Memory: 현재 application이 실제로 사용되는 활성화된 호스트 메모리

   ▼ example

   1. Reserve 1GB of host memory(Shared host memory)

```
shell>>sudo xbutil configure --host-mem -d 0000:17:00.1 --size 1G enable

Host-mem enabled successfully
shell>>xbutil examine -r pcie-info -d 0000:17:00.1
```

```
----------------------------------------------
1/1 [0000:a6:00.1] : xilinx_u250_gen3x16_xdma_shell_3_1
----------------------------------------------
Pcie Info
  Vendor              : 0x10ee
  Device              : 0x5005
  Sub Device          : 0x000e
  Sub Vendor          : 0x10ee
  PCIe                : Gen3x16
  DMA Thread Count    : 2
  CPU Affinity        : 16-31,48-63
  Shared Host Memory  : 1 GB
  Max Shared Host Memory : 0 Byte
  Enabled Host Memory : 0 Byte
```

2. Load xclbin → Max shared host memory가 뜬다.

```
shell>>xbutil examine -r pcie-info -d 0000:17:00.1

----------------------------------------------
1/1 [0000:a6:00.1] : xilinx_u250_gen3x16_xdma_shell_3_1
----------------------------------------------
Pcie Info
  Vendor              : 0x10ee
  Device              : 0x5005
  Sub Device          : 0x000e
  Sub Vendor          : 0x10ee
  PCIe                : Gen3x16
  DMA Thread Count    : 2
  CPU Affinity        : 16-31,48-63
  Shared Host Memory  : 1 GB
  Max Shared Host Memory : 16 GB
  Enabled Host Memory : 0 Byte
```

3. Run an application

```
shell>>xbutil examine -r pcie-info -d 0000:17:00.1

----------------------------------------------
1/1 [0000:a6:00.1] : xilinx_u250_gen3x16_xdma_shell_3_1
----------------------------------------------
Pcie Info
  Vendor              : 0x10ee
  Device              : 0x5005
  Sub Device          : 0x000e
  Sub Vendor          : 0x10ee
  PCIe                : Gen3x16
  DMA Thread Count    : 2
  CPU Affinity        : 16-31,48-63
  Shared Host Memory  : 1 GB
```

```
Max Shared Host Memory : 16 GB
Enabled Host Memory    : 1 GB
```

3. Host Code Guideline
   - Host-only buffer를 사용하기 위해서 buffer를 선언할 때 flag를 지정해야 한다.

```
// XRT Native API
xrt::bo buffer_in (device, size, xrt::bo::flags::host_only, kernel.group_id(0));
xrt::bo buffer_out(device, size, xrt::bo::flags::host_only, kernel.group_id(1));
```

   - Let XRT allocate the buffer as shown in the above code examples. Do not create a buffer from an already created user-space memory. The host code should map the buffer object to the user-space for read/write operation.
   - Regular data transfer APIs (OpenCL: `clEnqueueMigramemObjects` / `clEnqueueWriteBuffer`, XRT Native API: `xrt::bo::sync()`) should be used. Though these API will not do any DMA operation, they are used for Cache Invalidate/Flush as the application works on the Cache memory.

XRT Native API for Memory Management(buffer management)

1. **Buffer Allocation and Deallocation**
   - Regular buffer 생성

     `xrt::bo( const xrt::device &device, size_t sz, bo::flags flags, memory_group grp)`

```
auto bank_grp_arg0 = kernel.group_id(0); // Memory bank index for kernel argument 0
auto bank_grp_arg1 = kernel.group_id(1); // Memory bank index for kernel argument 1

auto input_buffer = xrt::bo(device, buffer_size_in_bytes,bank_grp_arg0);
auto output_buffer = xrt::bo(device, buffer_size_in_bytes, bank_grp_arg1);
```

     - `xrt::bo` 로 할당된 regular buffer는 4K aligned된다 (flag를 통해 다른 buffer도 선언 가능).
     - memory bank index의 두 지정 방식
       a. 커널 인자를 통한 지정: `xrt::kernel::group_id()` 를 사용해 메모리 뱅크를 지정

          e.g. `auto input_buffer = xrt::bo(device, buffer_size_in_bytes, kernel.group_id(0));`

       b. 직접 지정:

          e.g. `auto input_buffer = xrt::bo(device, buffer_size_in_bytes, 1);`

          → 메모리 뱅크 인덱스는 `xbutil examine --report memory` 명령을 사용해 확인(xclbin load 이후)

   - User pointer를 통한 buffer 생성

     `xrt::bo( const xrt::device &device, void *userptr, size_t sz, bo::flags flags, memory_group grp)`

```
// Host Memory pointer aligned to 4K boundary
 int *host_ptr;
 posix_memalign(&host_ptr,4096,MAX_LENGTH*sizeof(int));

 // Sample example filling the allocated host memory
 for(int i=0; i<MAX_LENGTH; i++) {
```

```
host_ptr[i] = i;  // whatever
}

auto mybuf = xrt::bo (device, host_ptr, MAX_LENGTH*sizeof(int), kernel.group_id(3));
```

▼ flag를 통한 여러 종류의 buffer 생성

- `xrt::bo::flags::normal` : Regular buffer (default)
- `xrt::bo::flags::device_only` : Device only buffer (meant to be used only by the kernel, there is no host backing pointer).
- `xrt::bo::flags::host_only` : Host only buffer (buffer resides in the host memory directly transferred to/from the kernel)
- `xrt::bo::flags::p2p` : P2P buffer, A special type of device-only buffer capable of peer-to-peer transfer
- `xrt::bo::flags::cacheable` : Cacheable buffer can be used when the host CPU frequently accessing the buffer (applicable for edge platform).

2. **Data transfer using Buffers**

   a. Host ↔ device by Buffer read/write API

   host buffer에 read/write한 후 read/write API를 통해 kernel buffer에 접근하는 방식이다.

   - buffer writing
     i. `xrt::bo::write()`
     ii. `xrt::bo::sync()` with flag `XCL_BO_SYNC_BO_TO_DEVICE`
   - buffer reading
     i. `xrt::bo::sync()` with flag `XCL_BO_SYNC_BO_FROM_DEVICE`
     ii. `xrt::bo::read()`

```
auto input_buffer = xrt::bo(device, buffer_size_in_bytes, bank_grp_idx_0);
// Prepare the input data
int buff_data[data_size];
for (auto i=0; i<data_size; ++i) {
  buff_data[i] = i;
}

input_buffer.write(buff_data);
input_buffer.sync(XCL_BO_SYNC_BO_TO_DEVICE);
```

   b. Data transfer between host and device by Buffer map API(`xrt::bo::map()`)

   host buffer를 kernel buffer에 mapping한 후 mapping된 host buffer에 read/write하는 방식이다.

> 💡 mapped pointer에 write한 후나 read하기 전에 꼭 API `xrt::bo::sync()`을 방향에 맞게 사용하여 synchronization해야 하는 것을 잊지 말자

```
auto input_buffer = xrt::bo(device, buffer_size_in_bytes, bank_grp_idx_0);
auto input_buffer_mapped = input_buffer.map<int*>();

for (auto i=0;i<data_size;++i) {
   input_buffer_mapped[i] = i;
}

input_buffer.sync(XCL_BO_SYNC_BO_TO_DEVICE);
```

c. Data transfer between the buffers by copy API

두 buffer의 deep-copy를 하는 방식이다. Platform에 따라 deep-copy를 지원하지 않을 수도 있으니 확인해야 한다. 위에서 서술한 M2M 방식을 참조하자.

```
dst_buffer.copy(src_buffer, copy_size_in_bytes);
```

3. **Miscellaneous**

a. DMA-BUF API

- 장치간 (P2P) 또는 process 사이에서(IPC, ex: unix domain socket) data transfer
- `xrt::bo::export_buffer()` 는 buffer handler에 대응되는 file descriptor를 출력한다.
- e.g. device1 → device2(inside same host process)

```
auto buffer_exported = buffer_device_1.export_buffer();
auto buffer_device_2 = xrt::bo(device_2, buffer_exported);
```

b. Sub-buffer support

parent buffer로부터 sub-buffer를 선언할 수 있다.

```
size_t sub_buffer_size = 4;
size_t sub_buffer_offset = 0;

auto sub_buffer = xrt::bo(parent_buffer, sub_buffer_size, sub_buffer_offset);
```

---

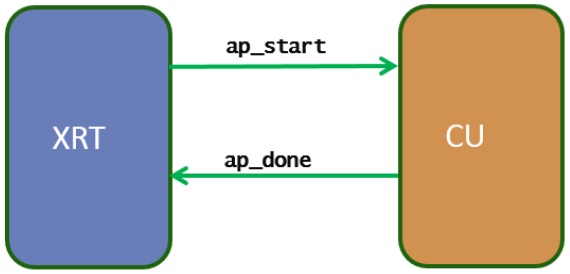
3. Execution Management

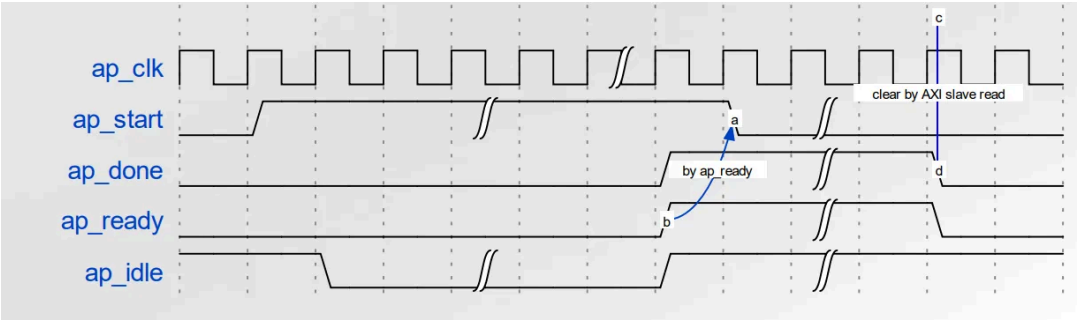
- OpenCL(`clEnqueueTask`) 또는 XRT APIs(`xrt::run`)를 통해 kernel 실행 가능
- Kernel의 interface를 따로 handle할 필요 없다.
- XRT의 automatic execution flow의 장점을 활용하기 위해서 RTL kernel을 설계할 때 XRT supported execution model을 잘 이해해야 한다.

- Kernel은 AXI4-Lite Slave interface의 control & status register(0x0 mapped)를 통해 control된다.

Sequential Execution Model (a.k.a ap_ctrl_hs)

- Axi slave register signals : `ap_start` (0) and `ap_done` (1)
- 현재 execution 종료 전에는 다음 execution 불가능 → sequential execution



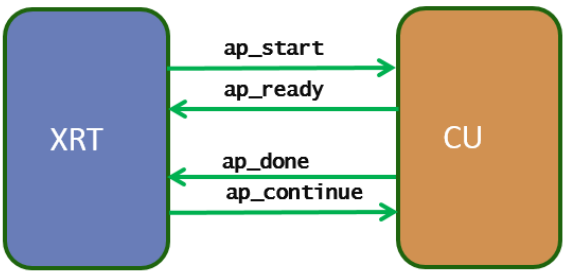


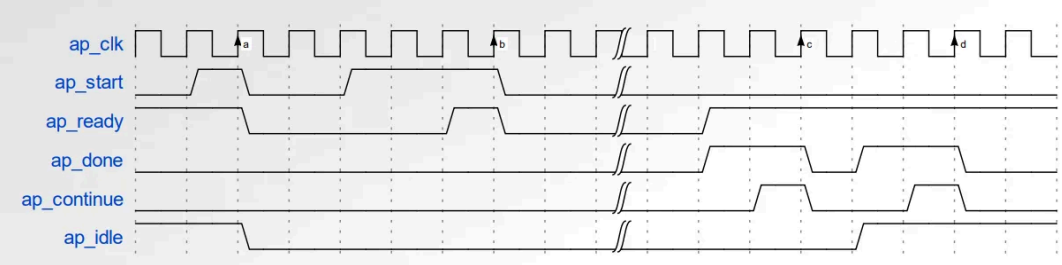
1. The XRT driver writes a 1 in ap_start to start the kernel
2. The XRT driver waits for ap_done to be asserted by the kernel (guaranteeing the output data is fully produced by the kernel).
3. Repeat 1-2 for next kernel execution

> 💡 e.g. START1⇒DONE1⇒START2⇒DONE2⇒START3⇒DONE3

Pipelined Execution Model (a.k.a ap_ctrl_chain)

- Axi slave register signals for input sync : `ap_start` (0) and `ap_done` (1)
- Axi slave register signals for output sync : `ap_ready` (3) and `ap_continue` (4)
- Kernel이 다른 execution 중에도 restart 가능
- 여러 execution이 overlap되며 pipeline 방식으로 실행된다. → pipelined execution





Input과 output synchronization은 asynchronous하게 일어난다.

- Input sync
  1. The XRT driver writes a 1 in ap_start to start the kernel
  2. The XRT driver waits for ap_ready to be asserted by the kernel (guaranteeing the kernel is ready to accept new data for next execution, even if it is still working on the previous execution request).
  3. The XRT driver writes 1 in ap_start to start the kernel operation again

  > 💡 START1⇒START2⇒START3⇒DONE1⇒START4⇒DONE2⇒START5⇒DONE3⇒DONE4⇒DONE5

- Output sync
  1. The XRT driver waits for ap_done to be asserted by the kernel (guaranteeing the output data is fully produced by the kernel).
  2. The XRT driver writes a 1 in ap_continue to keep kernel running

XRT Native API for Execution

1. Prerequisite

   위에서 device를 open하고 xclbin을 load한 후, kernel을 kernel하고 kernel과 연결된 device memory bank에 접근하기 위한 host buffer를 열었다.

   ```
   //Open device
   auto device = xrt::device(dev_index);

   //Load xclbin
   auto xclbin_uuid = device.load_xclbin("kernel.xclbin");

   //Create kernel
   auto krnl = xrt::kernel(device, xclbin_uuid, name);

   //Allocate buffer
   auto input_buffer = xrt::bo(device, buffer_size_in_bytes, krnl.group_id(0));
   ```

2. Executing the kernel
   - The kernel execution using the `xrt::kernel()` operator with the list of arguments that returns a `xrt::run` object. This is an asynchronous API and returns after submitting the task.
   - `xrt::run::wait()` 을 통해 current execution이 끝날 때까지 현재 thread를 block할 수 있다.

- `xrt::run::set_arg()` 을 통해 kernel argument를 일부 수정할 수 있다. 아래의 예에서는, 2번째 argument가 수정되었다.
- `xrt::run::start()` 는 다음 kernel execution을 시작한다..

```
//kernel execution
// 1st kernel execution
auto run = kernel(krnl);
run.wait();

// 2nd kernel execution with just changing 2nd argument
run.set_arg(1,scalar_2); // Arguments are specified starting from 0
run.start();
run.wait();
```