LABA-1:

If the RTL module satisfies the RTL-kernel requirements as shown below,

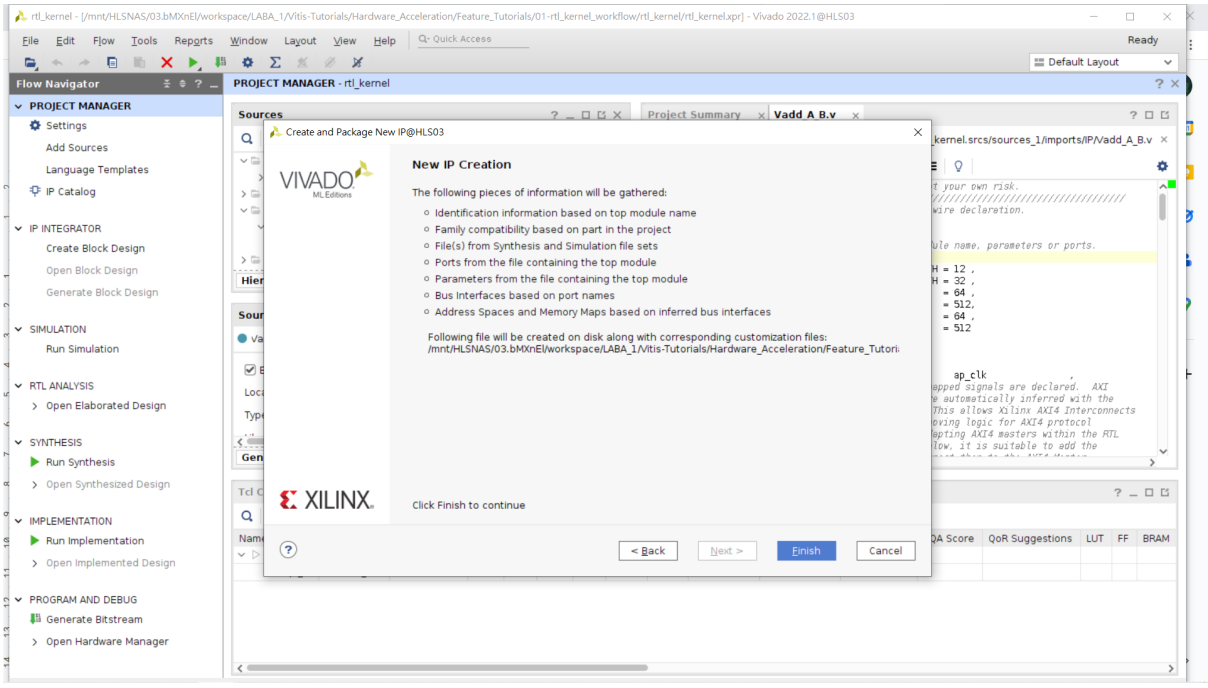
- The AXI4-Lite interface name must be packaged as `S_AXI_CONTROL`, but the underlying AXI ports can be named differently.
- Any memory-mapped AXI4 interfaces must be packaged as AXI4 master endpoints with 64-bit address support.

  ✎ **Recommended:** Xilinx strongly recommends that AXI4 interfaces be packaged with AXI meta data `HAS_BURST=0` and `SUPPORTS_NARROW_BURST=0`. These properties can be set in an IP-level `bd.tcl` file. This indicates wrap and fixed burst type is not used, and narrow (sub-size burst) is not used.

- You can also implement the AXI4-Stream interface.
- At least one clock is required for the kernel, though it can support multiple clocks.
  - Each clock must have an associated Bus Interface identifying it as a clock.
  - Each clock can have an optional active-Low reset, specified by the ASSOCIATED_RESET property on the clock.
  - A clock must be associated with each AXI4-Lite, AXI4, and AXI4-Stream interface on the kernel.

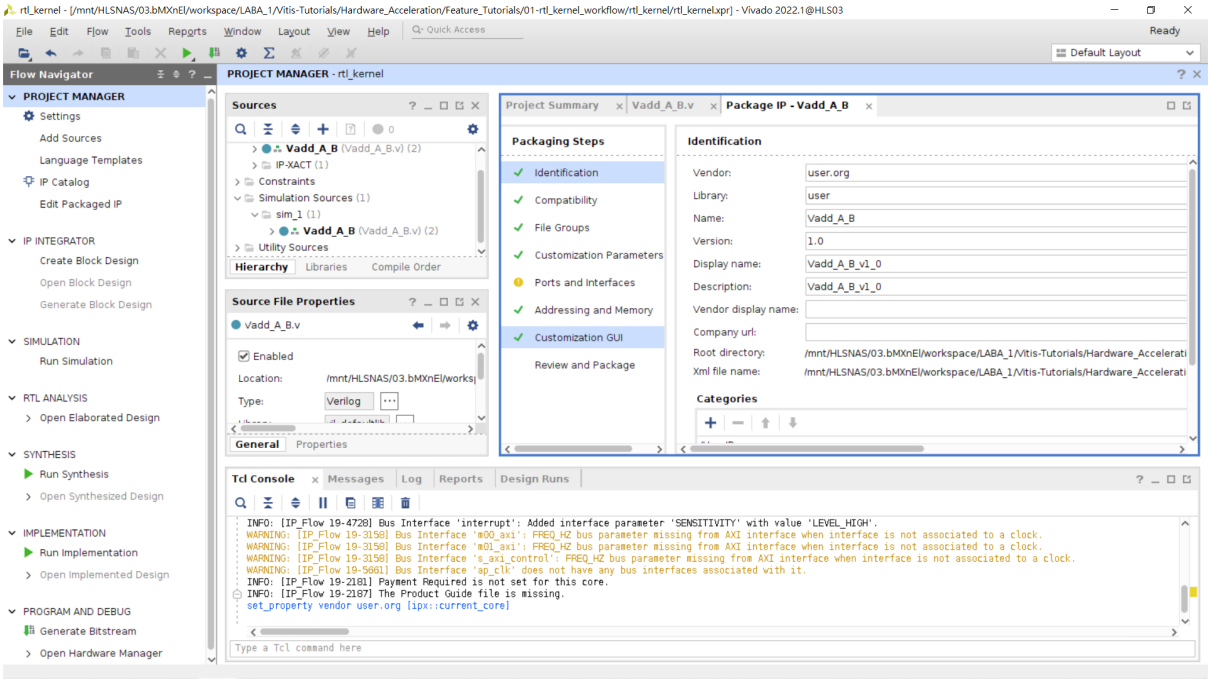
we can now click “Tools > Create and Package New IP” to start packaging our IP.

The location where the IP will be packaged:

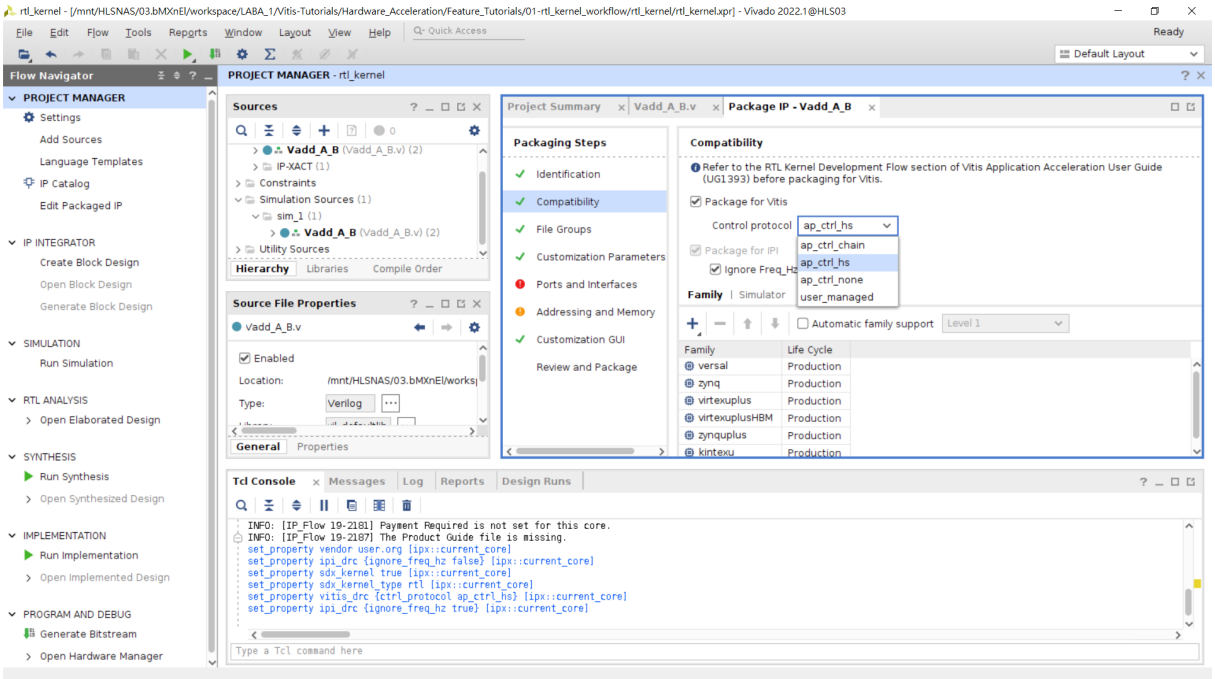


…../<thisvivadoproject>/rtl_kernel.srcs/sources_1/imports/IP

After clicking “Finish”, the Package IP window will show up:



Now, click “Compatibility” and enable “Package for Vitis”and “Ignore Freq_Hz”.
We can also see there are multiple choices for control protocol between host and kernel,

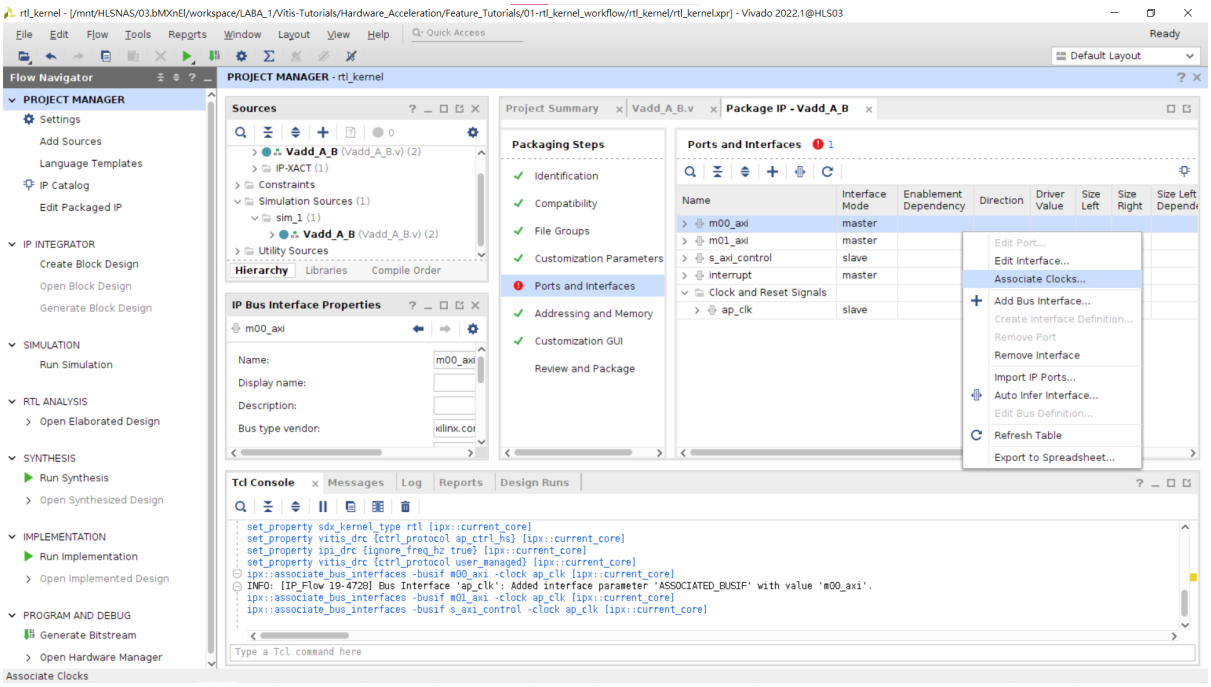


- `user_managed`: Defines a SW-controllable kernel, that is user-managed rather than XRT-managed. This is the preferred option. Refer to Creating User-Managed RTL Kernels for additional information.
- `ap_ctrl_hs`: This is the default, and specifies the simple sequential execution model for XRT - managed kernels as described in SW-Controllable Kernels.
- `ap_ctrl_chain`: Specifies a pipelined execution model for XRT-managed kernels.
- `ap_ctrl_none`: Indicates no control protocol as described in Non-Software Controlled Kernels.

If you choose “ap_ctrl_hs” or “ap_ctrl_chain” as control protocol, then the kernel will be XRT-managed kernel, on the contrary, it’ll be user managed if “user_managed” is chosen.

And the difference between user-managed kernel and XRT-managed kernel lies in the execution mode or method, namely how a kernel can be controlled by the host program. We choose “user_managed” here.

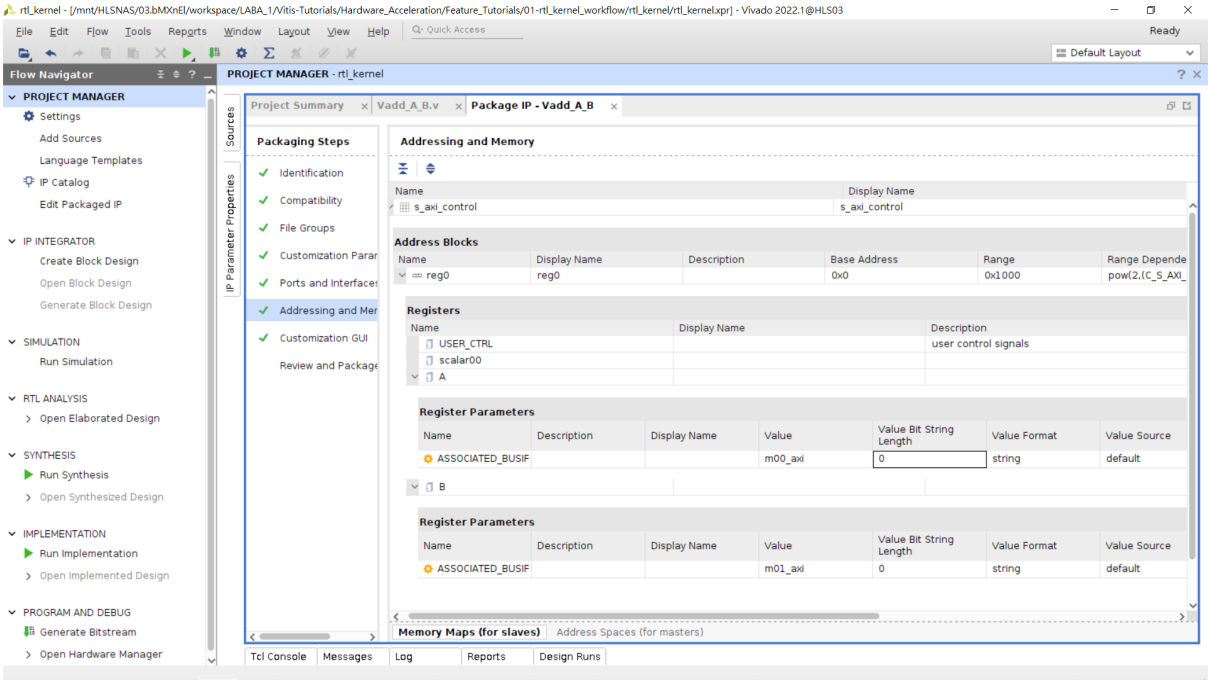
As stated in the kernel requirement, a clock must be associated with each AXI4-Lite, AXI4, and AXI4-Stream interface on the kernel. Thus, click the “Ports and interfaces” and associate ap_clk with m00_axi, m01_axi (M_AXI interfaces used for pointer arguments) and s_axi_control.

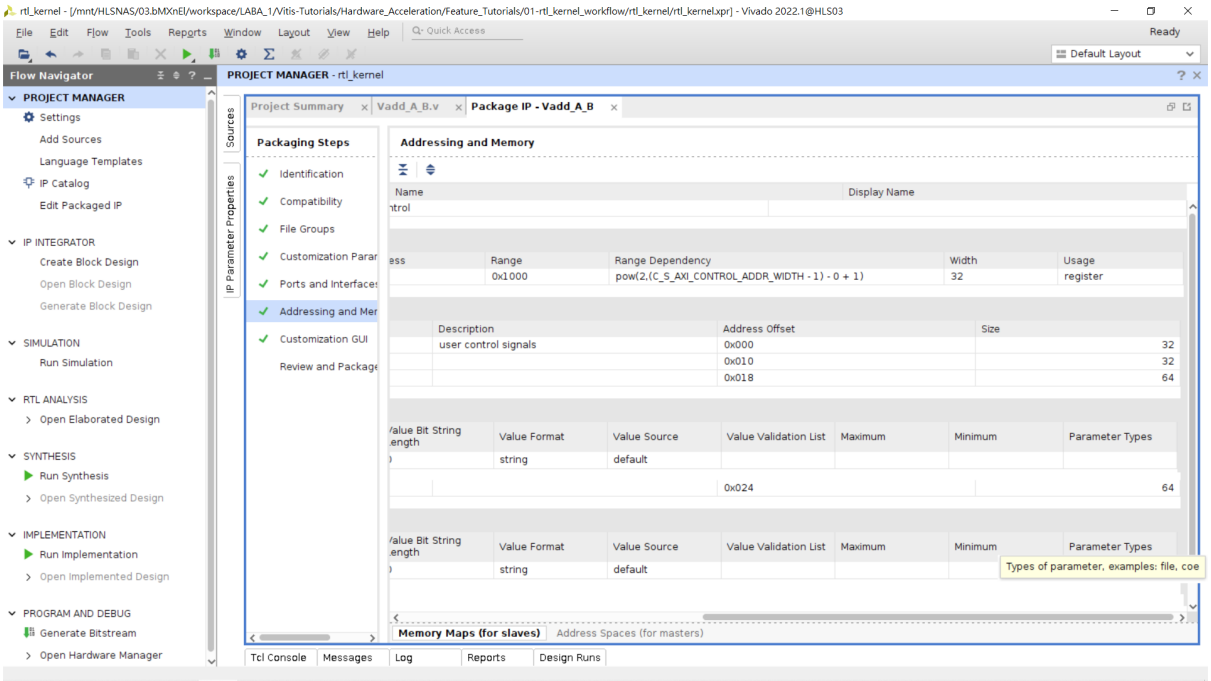


After that, click “Addressing and Memory” to add registers and address offsets. Right click the reg0 to add registers, and for each added registers, the size and offset is required to be specified, we fill them in accordance with the table below.

Name	Description	Offset	Size (bits)
USER_CTRL	Control Signals	0x000	32
scalar00	Scalar values	0x010	32
A	pointer argument	0x018	64
B	pointer argument	0x024	64

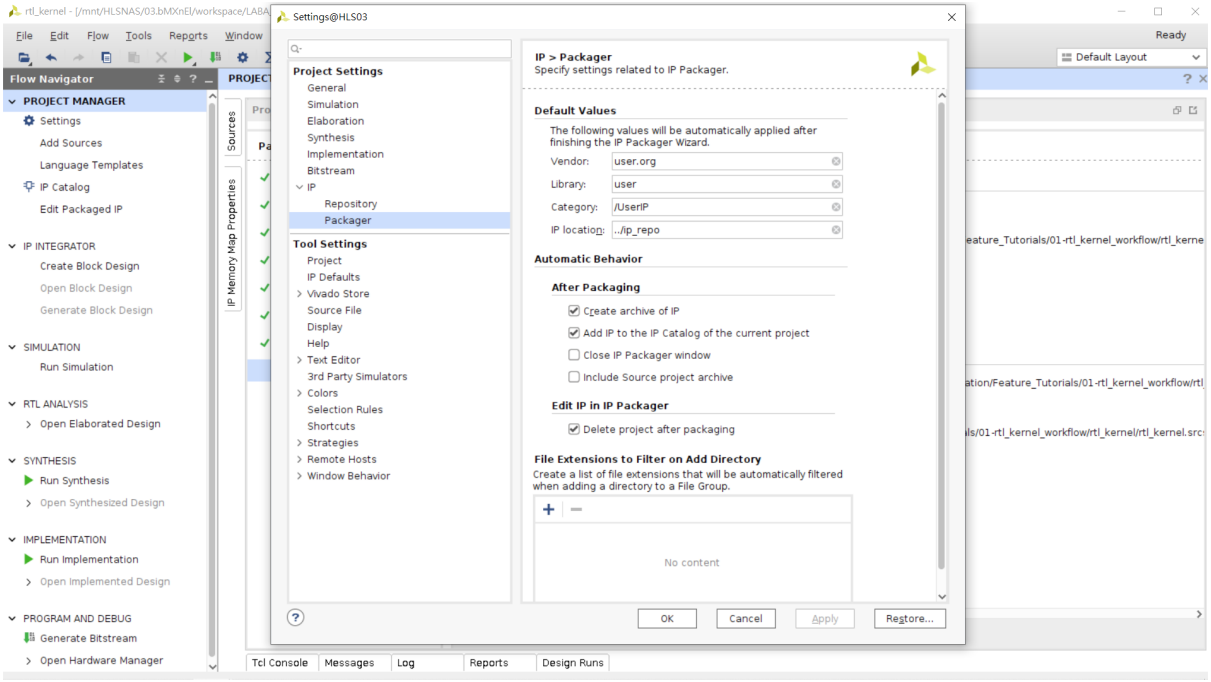
What's important is that we must also associate an M_AXI interface, m00_axi and m01_axi in this case, with each of the pointer arguments. Right click register A and add register parameters, enter "ASSOCIATED_BUSIF" in the box. This allow us to define an association between the register and the AXI4 Interface by entering the "m00_axi " in the value field.

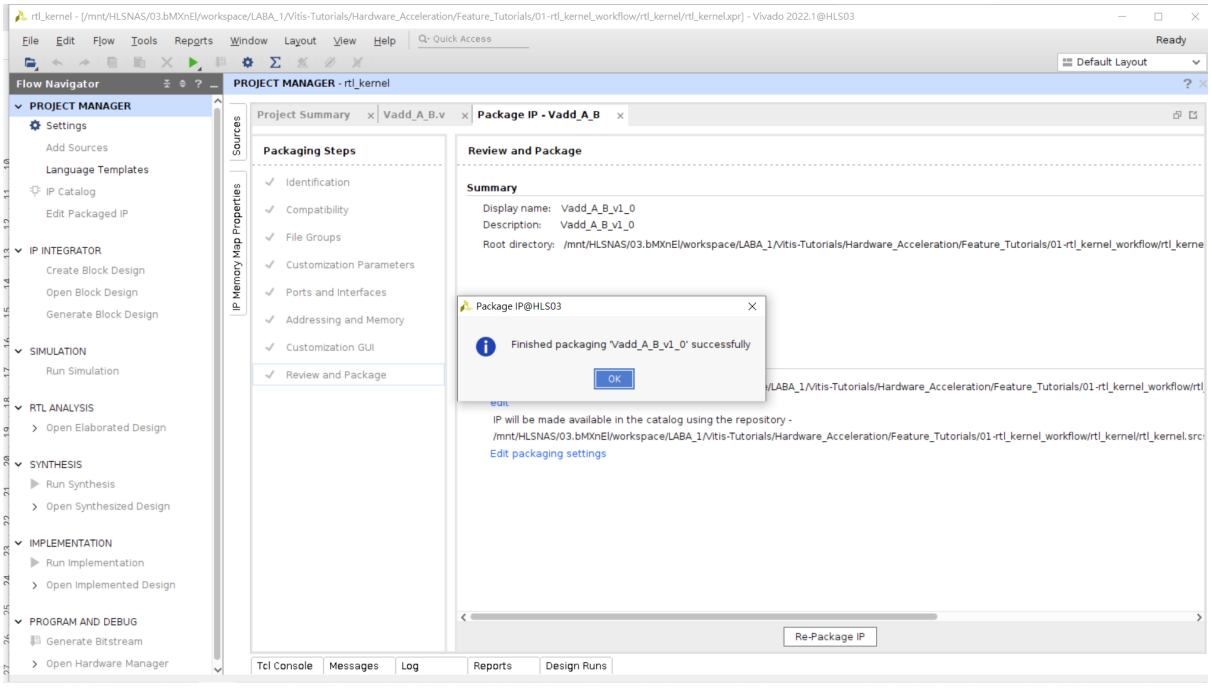




Now, the pointer arguments A and B use m00_axi interface and m01_axi_interface respectively.

Finally, remember to enable “**Create archive of IP**”. We can now return to “Review and Package” and click Package IP, the we’ll see the IP is successfully packaged.

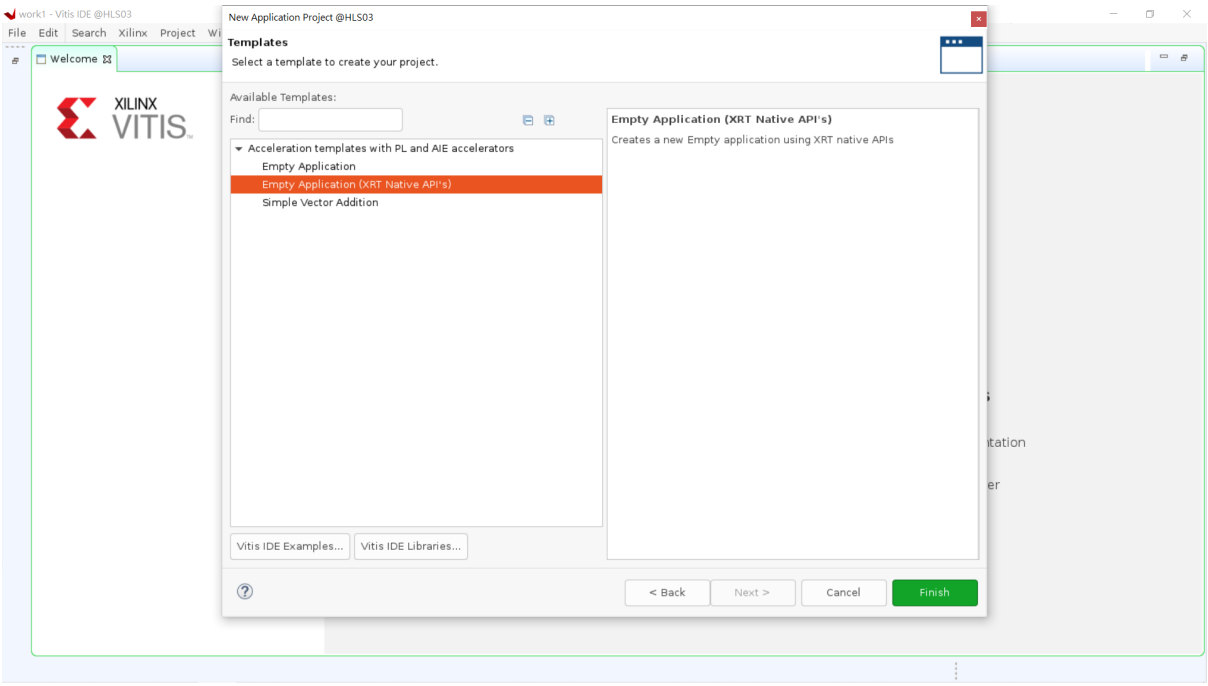




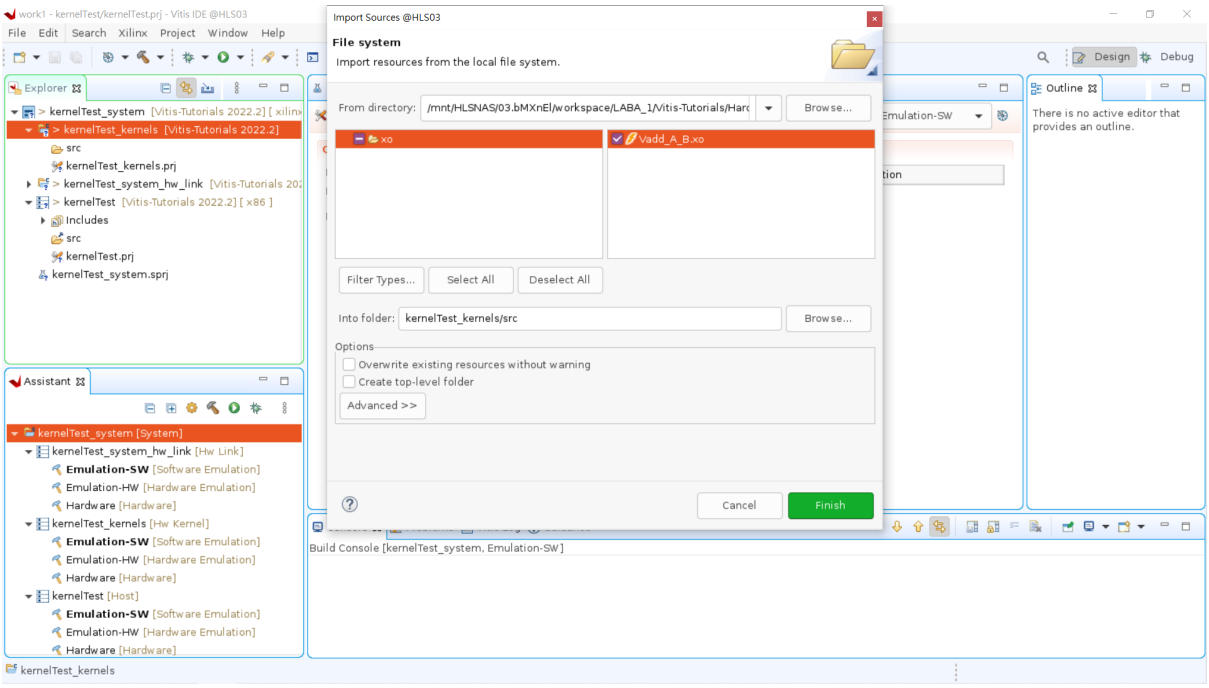
It also generates the Vitis kernel (.xo) file if enabling “Package for Vitis” in “Compatibility”, which will later be used in Vitis application acceleration. The .xo file contains the IP representation of the kernel and the kernel.xml file used by the Vitis compiler to connect to other HLS or RTL kernels during the linking process, and thereafter loaded into the device binary (.xclbin).

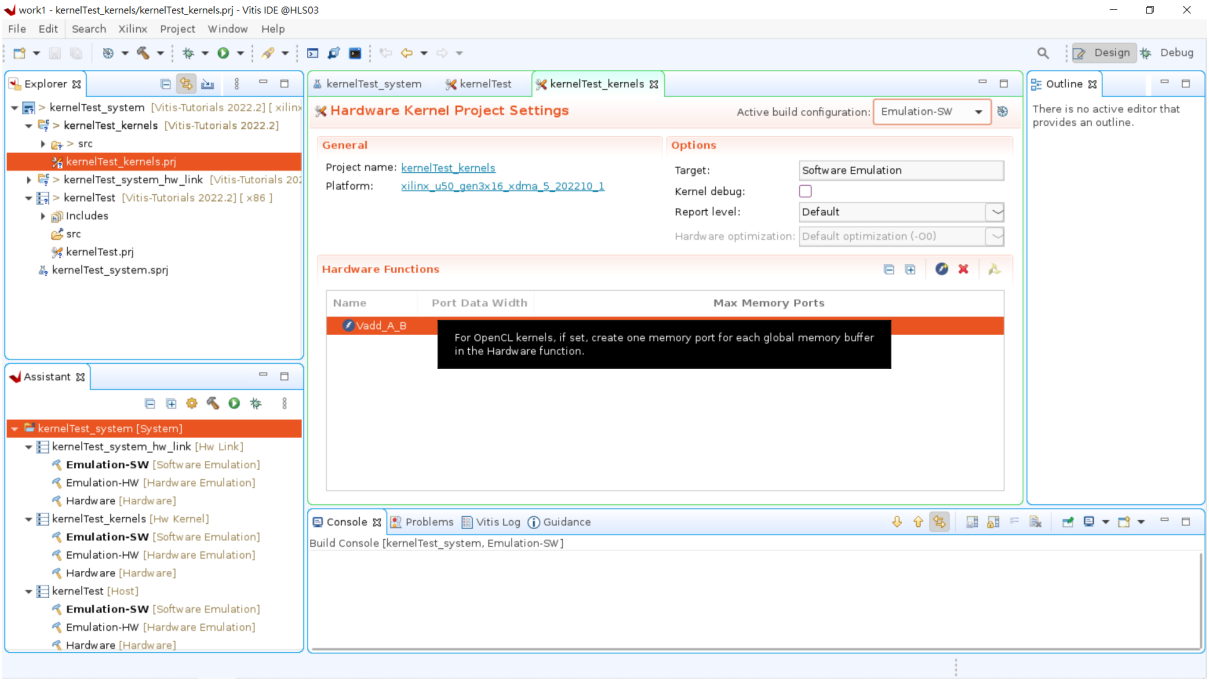
The generated XO file is located at .../<thisvivadoproject>/rtl_kernel.srcs/sources_1/imports

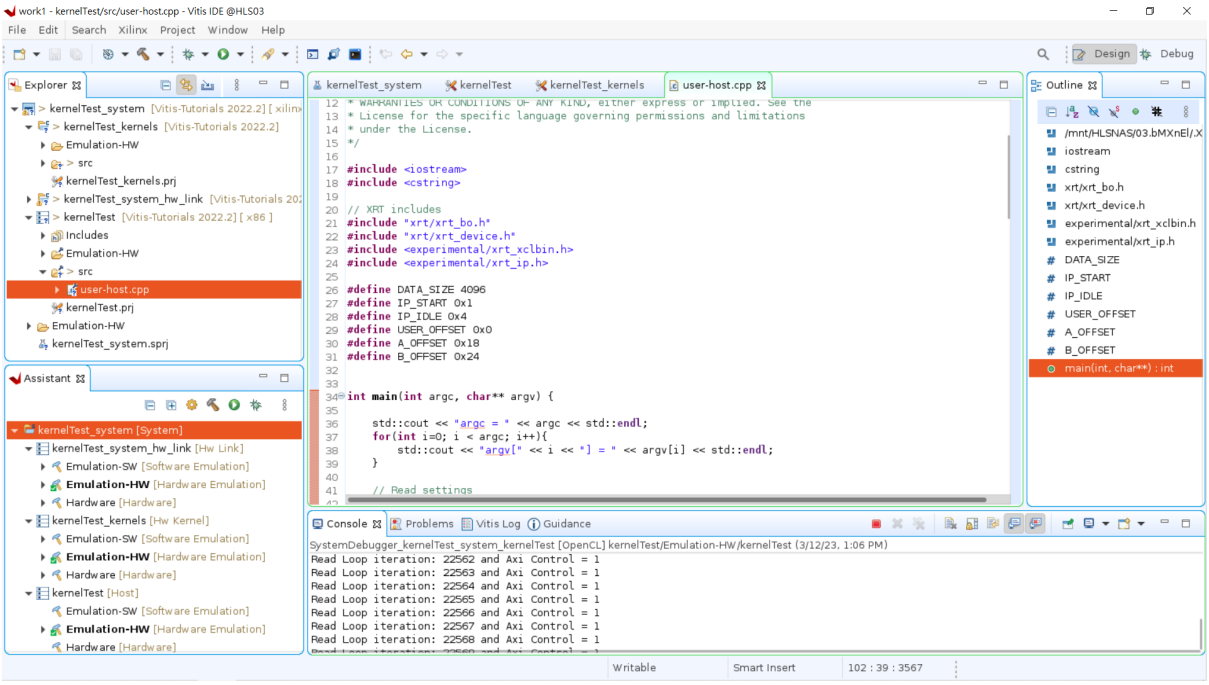
We can now test our rtl kernel, create a new application and choose Empty Application(XRT Native API)



import .xo (the vitis kernel file) and add hardware,







Host code introduction:

Common host code flow is:

(1) Open Xilinx **Device** and Load the **XCLBIN**

(2) Create **Buffer** objects to transfer data to kernel inputs and outputs

(3) Use the Buffer class member functions for the data transfer between host and device (before and after the kernel execution).

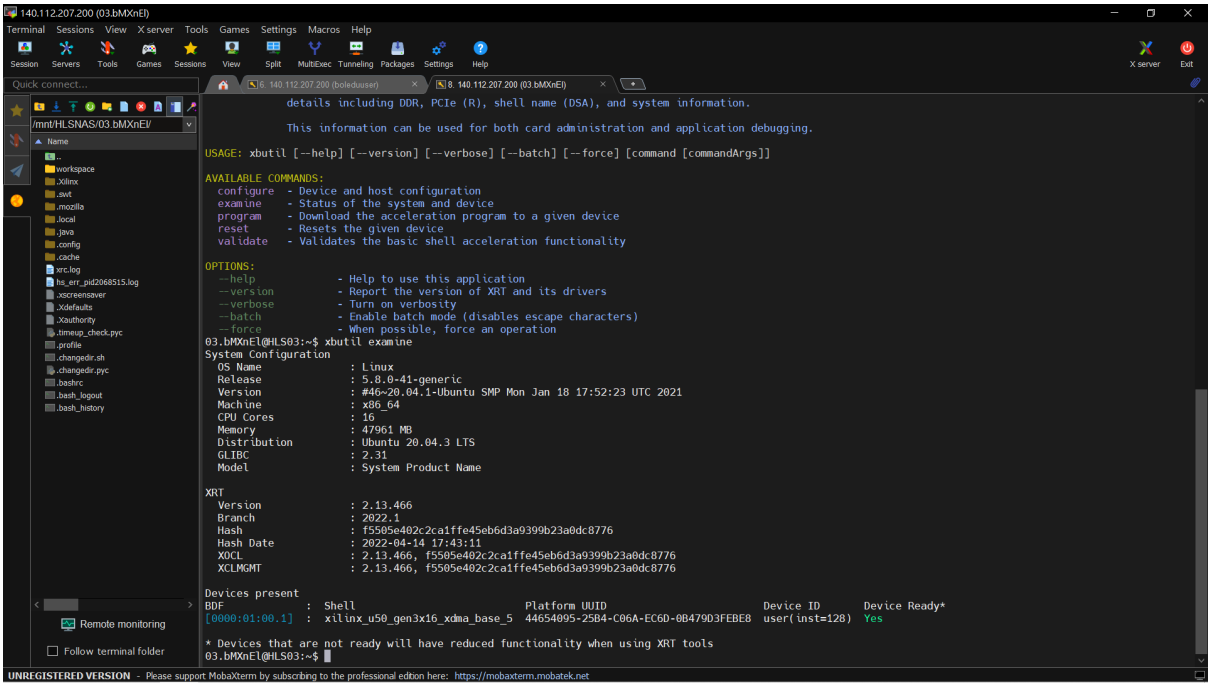
(4) Use **Kernel** and **Run** objects to offload and manage the compute-intensive tasks running on FPGA.

Setup:

include necessary XRT API library and define parameters

```
#include <iostream>
#include <cstring>

// XRT includes
#include "xrt/xrt_bo.h"
#include "xrt/xrt_device.h"
#include <experimental/xrt_xclbin.h>
#include <experimental/xrt_ip.h>

#define DATA_SIZE 4096
#define IP_START 0x1
#define IP_IDLE 0x4
#define USER_OFFSET 0x0
#define A_OFFSET 0x18
#define B_OFFSET 0x24
```

You can check the device and system informations through the command “xbutil examine”



(1) **Open Xilinx Device and Load the XCLBIN:**

xrt::device(device index) create a device object through device ID.
device.load_xclbin(binaryfile) load the .xclbin into the device object and return uuid which will be used later.(The compiler generates unique xclbin file tagged with UUID for every application compiled).

```
41    // Read settings
42      std::string binaryFile = argv[1];
43    auto xclbin = xrt::xclbin(binaryFile);
44    int device_index = 0;
45
46    std::cout << "Open the device " << device_index << std::endl;
47    auto device = xrt::device(device_index);
48    std::cout << "Load the xclbin " << binaryFile << std::endl;
49    auto uuid = device.load_xclbin(binaryFile);
50
51    size_t vector_size_bytes = sizeof(int) * DATA_SIZE;
52
53    auto ip1 = xrt::ip(device, uuid, "Vadd_A_B:{Vadd_A_B_1}");
54    //auto ip2 = xrt::ip(device, uuid, "Vadd_A_B:{Vadd_A_B_2}");
55    //auto ip3 = xrt::ip(device, uuid, "Vadd_A_B:{Vadd_A_B_3}");
```

line 53 creates an instance of the kernel named ip1, we can create multiple instances of the kernel in this way.

(2) **Create Buffer objects to transfer data to kernel inputs and outputs**

We create a buffer for the memory bank used by a specific argument. Buffer objects are created using xrt::bo, the third argument is used to specify the enumerated memory bank index (to specify the buffer location) where the buffer should be allocated.

We can also define a buffer as a kernel only or a host only buffer. Besides, lines 67~69 map the content of the buffer object into host memory so as to interact with the buffer (mapping the host-side buffer backing pointer to a user pointer).

```
57    std::cout << "Allocate Buffer in Global Memory\n";
58    //auto boA = xrt::bo(device, vector_size_bytes, krnl.group_id(1)); //Match ker
59    //auto boB = xrt::bo(device, vector_size_bytes, krnl.group_id(2));
60    auto ip1_boA = xrt::bo(device, vector_size_bytes, 0);
61    auto ip1_boB = xrt::bo(device, vector_size_bytes, 1);
62    //auto ip2_boA = xrt::bo(device, vector_size_bytes, 0);
63    //auto ip2_boB = xrt::bo(device, vector_size_bytes, 1);
64    //auto ip3_boA = xrt::bo(device, vector_size_bytes, 0);
65    //auto ip3_boB = xrt::bo(device, vector_size_bytes, 1);
66
```

```
67      // Map the contents of the buffer object into host memory
68      auto bo0_map = ip1_boA.map<int*>();
69      auto bo1_map = ip1_boB.map<int*>();
70
71      std::fill(bo0_map, bo0_map + DATA_SIZE, 0);
72      std::fill(bo1_map, bo1_map + DATA_SIZE, 0);
73
```

We must use the buffer for the specified kernel argument when performing a **read_register** or **write_register** command for the kernel, for example:

ip1.write_register(register offset for argument A, address of buffer for argument A)

(3) **Use the Buffer class member functions for the data transfer between host and device (before the kernel execution).**

Now, we can prepare the test data by which we will use them to set the content of the buffer with the sync function.

```
    // Create the test data
    int bufReference[DATA_SIZE];
    for (int i = 0; i < DATA_SIZE; ++i) {
        bo0_map[i] = i;
        bo1_map[i] = i;
        bufReference[i] = bo0_map[i] + bo1_map[i]; //Generate check data for validation
    }

    std::cout << "loaded the data" << std::endl;
    uint64_t buf_addr[2];
    // Get the buffer physical address
    buf_addr[0] = ip1_boA.address();
    buf_addr[1] = ip1_boB.address();

    // Synchronize buffer content with device side
    std::cout << "synchronize input buffer data to device global memory\n";
    ip1_boA.sync(XCL_BO_SYNC_BO_TO_DEVICE);
    ip1_boB.sync(XCL_BO_SYNC_BO_TO_DEVICE);
```

These operations mean we will set the value of input arguments of the kernel, and the value of the elements of bufReference is simply the value of A+B.

(4) **Use Kernel and Run objects to offload and manage the compute-intensive tasks running on FPGA.**

Firstly ,we have to pass the buffer contents which had been set up in the previous section, to the kernel arguments A and B by specifying the corresponding registers and buffers.

In user managed kernel, we use two states to control the kernel, “IP_START” and “IP_IDLE”, we can change the control register by setting the value of axi_ctrl.

```
    std::cout << "INFO: Setting IP Data" << std::endl;
    std::cout << "Setting Register \"A\" (Input Address)" << std::endl;
    ip1.write_register(A_OFFSET, buf_addr[0]);
    ip1.write_register(A_OFFSET + 4, buf_addr[0] >> 32);

    std::cout << "Setting Register \"B\" (Input Address)" << std::endl;
    ip1.write_register(B_OFFSET, buf_addr[1]);
    ip1.write_register(B_OFFSET + 4, buf_addr[1] >> 32);

    uint32_t axi_ctrl = IP_START;
    std::cout << "INFO: IP Start" << std::endl;
    //axi_ctrl = IP_START;
    ip1.write_register(USER_OFFSET, axi_ctrl);

    // Wait until the IP is DONE
    int i = 0;
    //axi_ctrl = 0;
    while (axi_ctrl != IP_IDLE) {
    //while ((axi_ctrl & IP_IDLE) != IP_IDLE) {
        axi_ctrl = ip1.read_register(USER_OFFSET);
        i = i + 1;
        std::cout << "Read Loop iteration: " << i << " and Axi Control = " << axi_ctrl << "\n";
        if (i > 100000) {
      axi_ctrl = IP_IDLE;
          ip1.write_register(USER_OFFSET, axi_ctrl);
        }
    }

    std::cout << "INFO: IP Done" << std::endl;
```

(5) **Use the Buffer class member functions for the data transfer between host and device (after the kernel execution).**

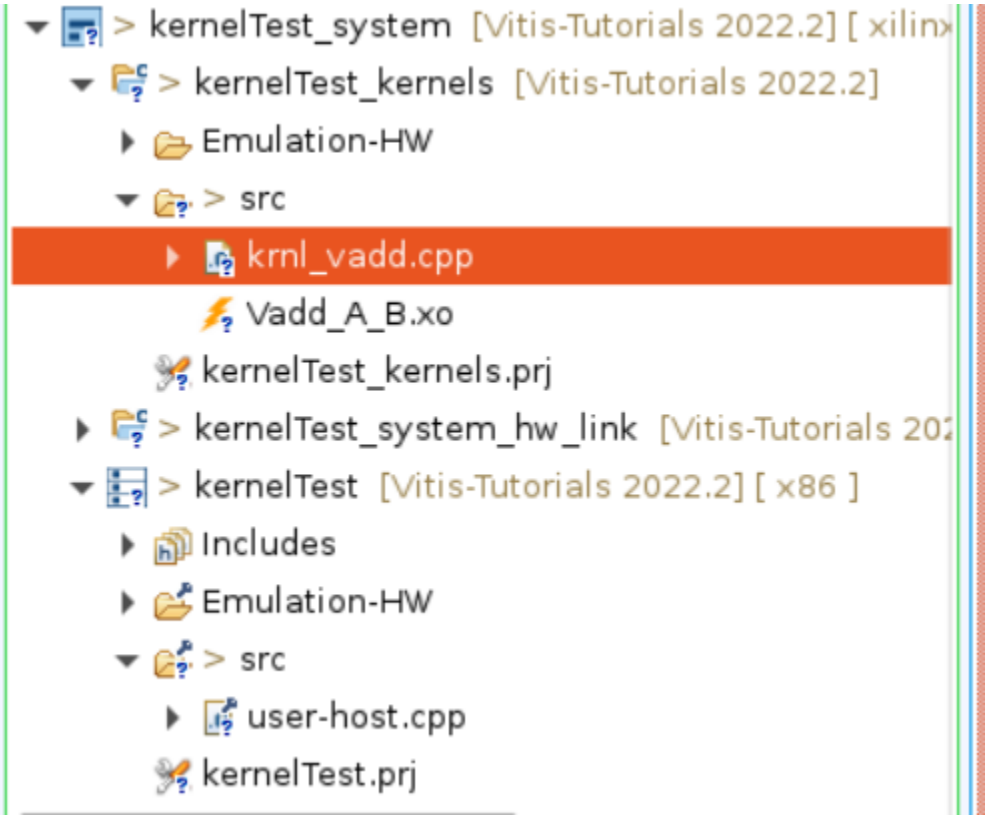
Finally, we got the output data computed by the kernel with the sync function.

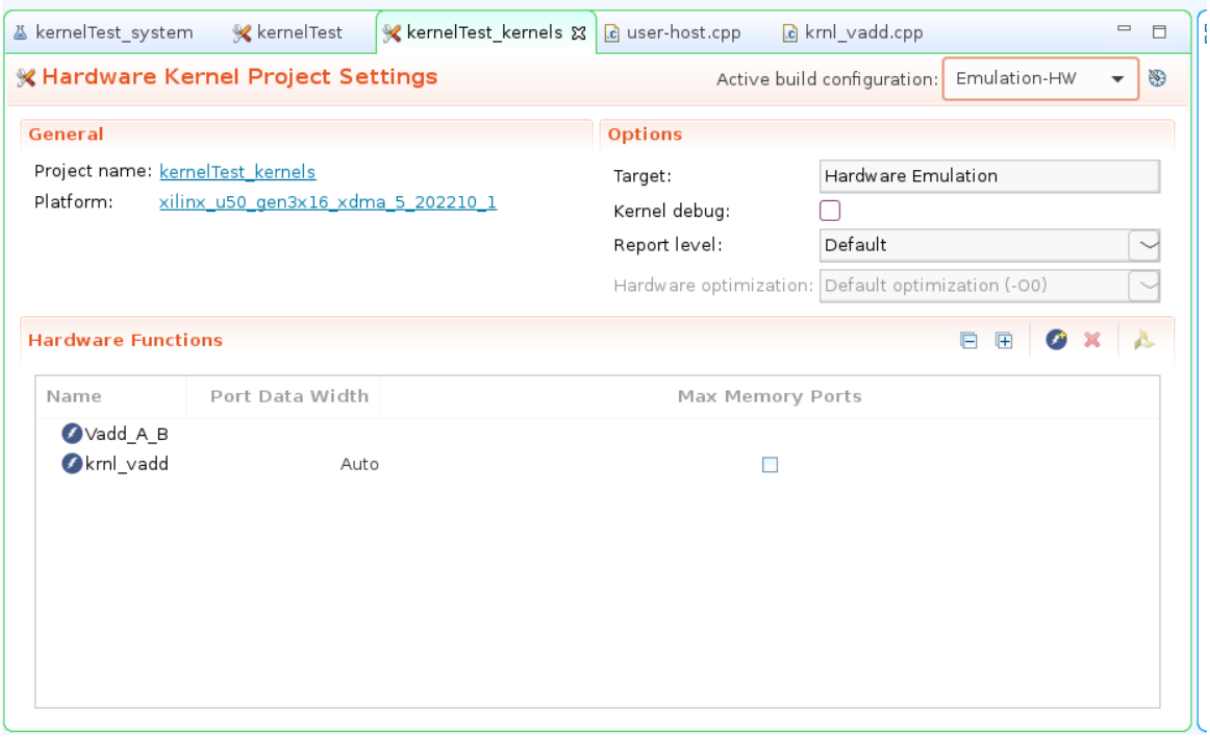
```
    // Get the output;
    std::cout << "Get the output data from the device" << std::endl;
    ip1_boB.sync(XCL_BO_SYNC_BO_FROM_DEVICE);

    // Validate results
    if (std::memcmp(bo1_map, bufReference, DATA_SIZE))
        throw std::runtime_error("Value read back does not match reference");

    std::cout << "TEST PASSED\n";
    return 0;
}
```

LABA-2:

I use the krnlVaddl.cpp file in https://github.com/Xilinx/Vitis-Tutorials/tree/2022.1/Hardware_Acceleration/Feature_Tutorials/02-mixing-c-rtl-kernels as my C++ kernel, and add it to my project. The cpp kernel will do the operation stated in the above file.

The kernel function in the explorer and hardware function will be like:





The host code is almost the same as that of the previous section(LabA_1), but add some code as the following:

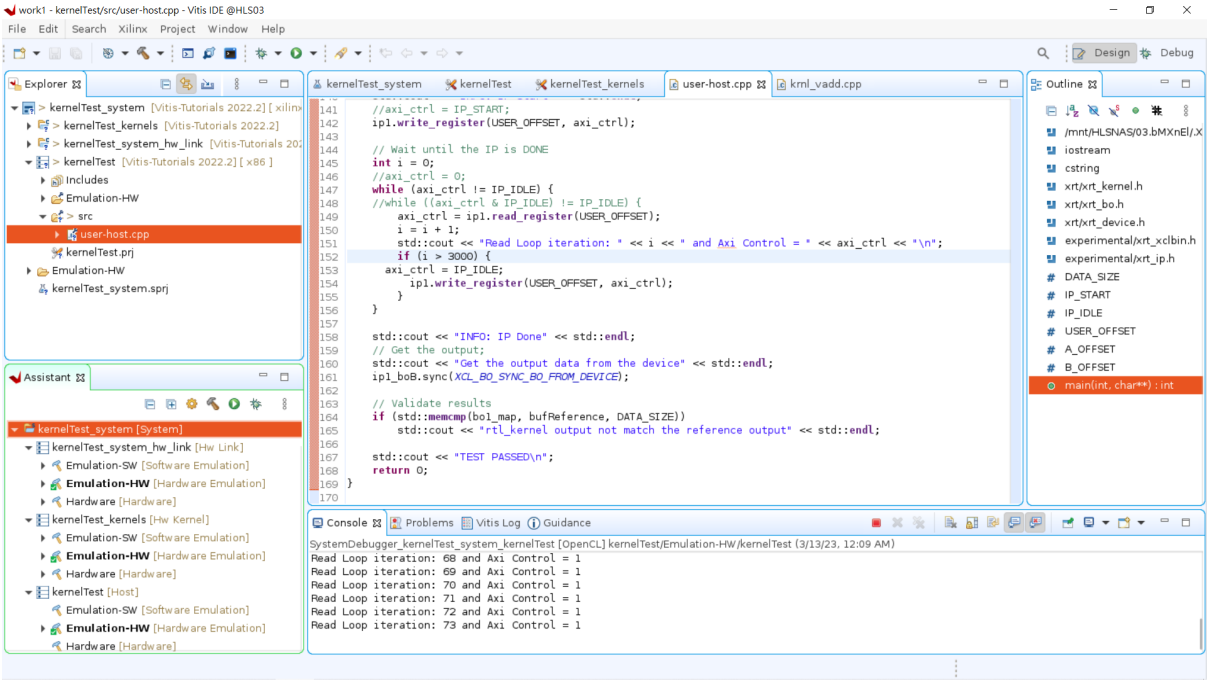
```
////////cpp_kernel/////////
std::cout << "setting up cpp_kernel" << std::endl;
auto cpp_krnl=xrt::kernel(device,uuid,"krnl_vadd:{krnl_vadd_1}");

auto bo_a=xrt::bo(device, vector_size_bytes, cpp_krnl.group_id(0));
auto bo_b=xrt::bo(device, vector_size_bytes, cpp_krnl.group_id(1));
auto bo_c=xrt::bo(device, vector_size_bytes, cpp_krnl.group_id(2));

auto bo_a_map = bo_a.map<int*>(); //it's like copy the pointer
auto bo_b_map = bo_b.map<int*>();
auto bo_c_map = bo_c.map<int*>();

int cpp_ref_outcome[DATA_SIZE];
for (int i = 0; i < DATA_SIZE; ++i) {
    bo_a_map[i] = i;
    bo_b_map[i] = i;
    cpp_ref_outcome[i] = bo_a_map[i] + bo_b_map[i]; //Generate check data for validation
}

std::cout << "Execution of the cpp_krnl\n";
auto run = cpp_krnl(bo_a, bo_b, bo_c, DATA_SIZE);
run.wait();

//get the output
bo_c.sync(XCL_BO_SYNC_BO_FROM_DEVICE);
if (std::memcmp(bo_c_map, cpp_ref_outcome, DATA_SIZE))
   std::cout << "cpp_kernel output not match the reference output" << std::endl;
```

I found that I have to treat the C++ kernel as XRT-managed kernel (see the instantiation using xrt::kernel rather than xrt::ip).

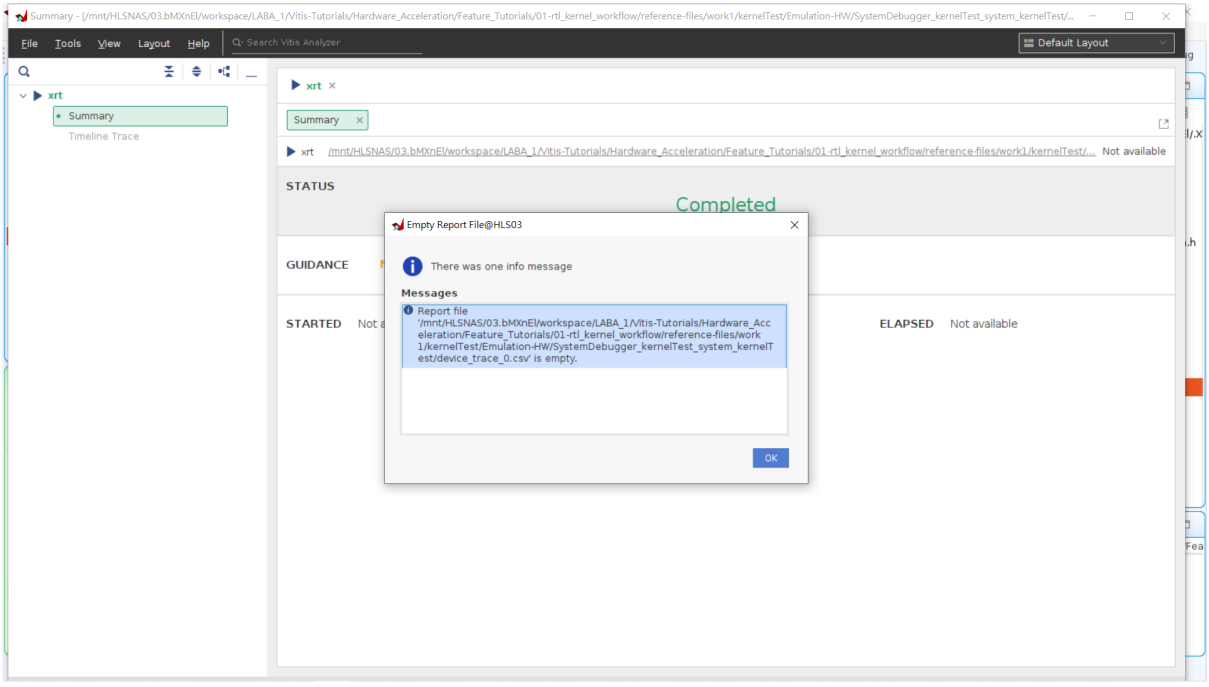
I don't think one can mix opencl api and xrt api in a host code at the same time because the xclbin should be created uniquely.

I also find that user managed kernel need more specification than xrt managed kernel, while it also reflects the flexibility of user managed kernel.

run emulation:



For timeline trace, I face some issue:



I need some time to upload the project because of the upload speed of my internet in my house is poor…, I'm late for a few minutes because I'm still finding the solution of timeline trace problem, but pls be kind to me ;_;

Github:https://github.com/ccontemplator/Vitis-mixed-c-rtl-kernel