School of Electrical & Computer Engineering - Aristotle University of Thessaloniki

Improving Performance Lab using Alveo U200

on PowerEdge R730 (2022.2)

Improving Performance

Introduction

In previous lab you learned how to create a project using GUI mode and went through entire design flow. At the end of the lab, you saw the limited transfer bandwidth due to 32-bit data operations. This bandwidth can be improved, and in turn system performance can be improved, by transferring wider data, and performing multiple operations in parallel. This is one of the common optimization methods to improve the kernel's bandwidth.

Objectives

After completing this lab, you will learn to:

- Create a project using Empty Application template in the Vitis GUI flow
- Import provided source files
- Run Hardware Emulation to see increased bandwidth
- Build the system and test it in hardware
- Perform profile and application timeline analysis in hardware emulation

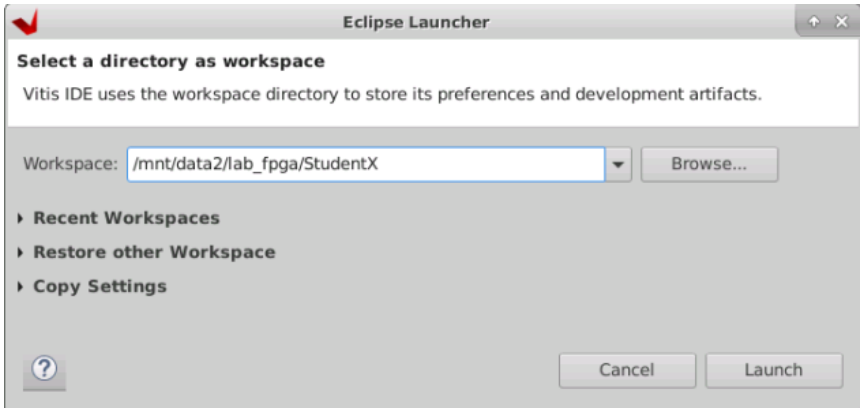
Steps

Create a Vitis Project

1. Invoke GUI by executing the following command:

```
vitis &
```

2. Set workspace as you can see in the following image (where the X, is your team number) and click **Launch**



Continue with the workspace you have used in previous lab

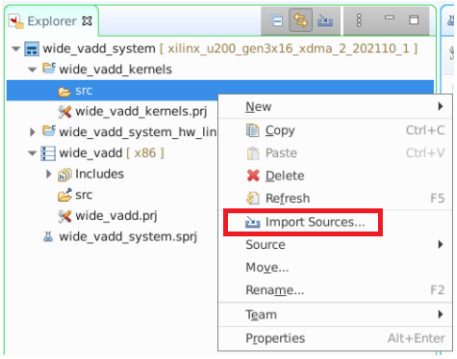
3. Select the platform `xilinx_u200_gen3x16_xdma...`, and click **OK**

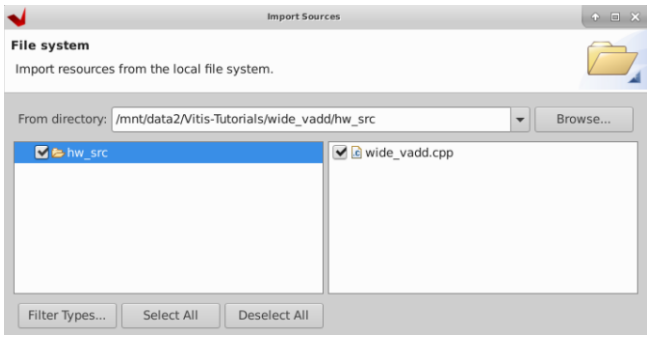
Create a new acceleration project giving `wide_vadd` as the project name, and click **Next>**

4. Select `Empty Application` as the template and click **Finish**

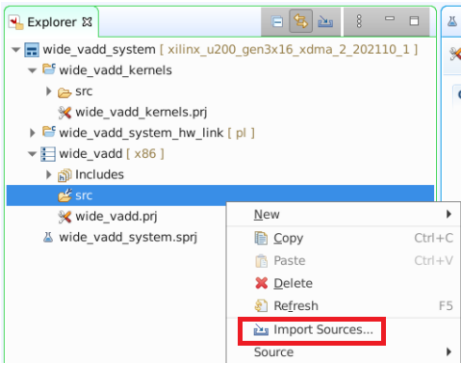
   The project is generated

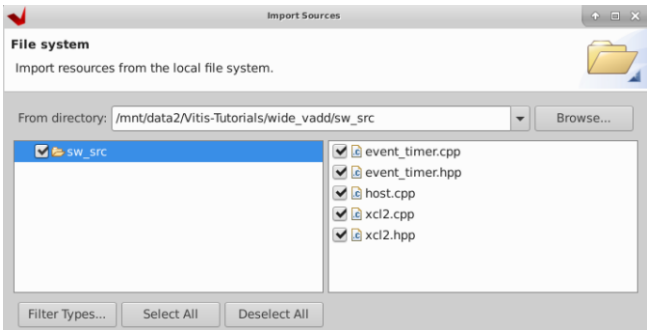
5. Import provided source files from vadd sub-directories.
   Select **vadd_kernels --> src** directory in Explorer view, right click and select Import Sources... in order to add the kernel
   Import `/mnt/data2/Vitis-Tutorials/wide_vadd/hw_src/wide_vadd.cpp` for **hardware accelerator** and press **Finish**



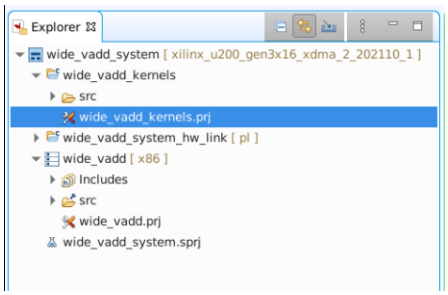


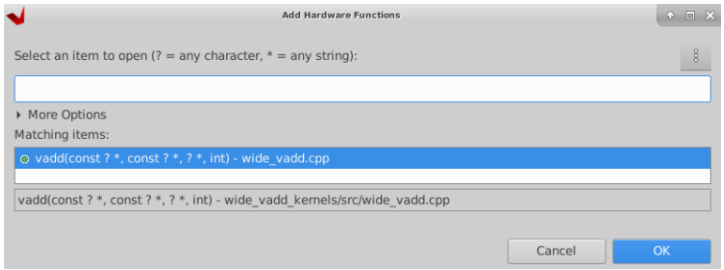
Select **vadd --> src** directory in Explorer view, right click and select Import Sources... in order to add the host files mport all `*.cpp` and `*.hpp` files in `/mnt/data2/Vitis-Tutorials/wide_vadd/sw_src/` for **host code** application and press **Finish**





6. The project is generated. Double Click on wide_vadd_kernels.prj and click on  in order to add **vadd** as a *Hardware Function* (kernel).





Analyze the kernel code

The DDR controller natively has a 512-bit wide interface internally. If we parallelize the dataflow in the accelerator, we will be able to process 16 array elements per clock tick instead of one. So, we should be able to get an instant 16x computation speed-up by just vectorizing the input

1. Double-click `wide_vadd.cpp` to view its content

   Look at lines 49-53 and note wider (512 bits) kernel interface

   `uint512_dt` is used in stead of `unsigned int` here for input, output and internal variables for data storage.

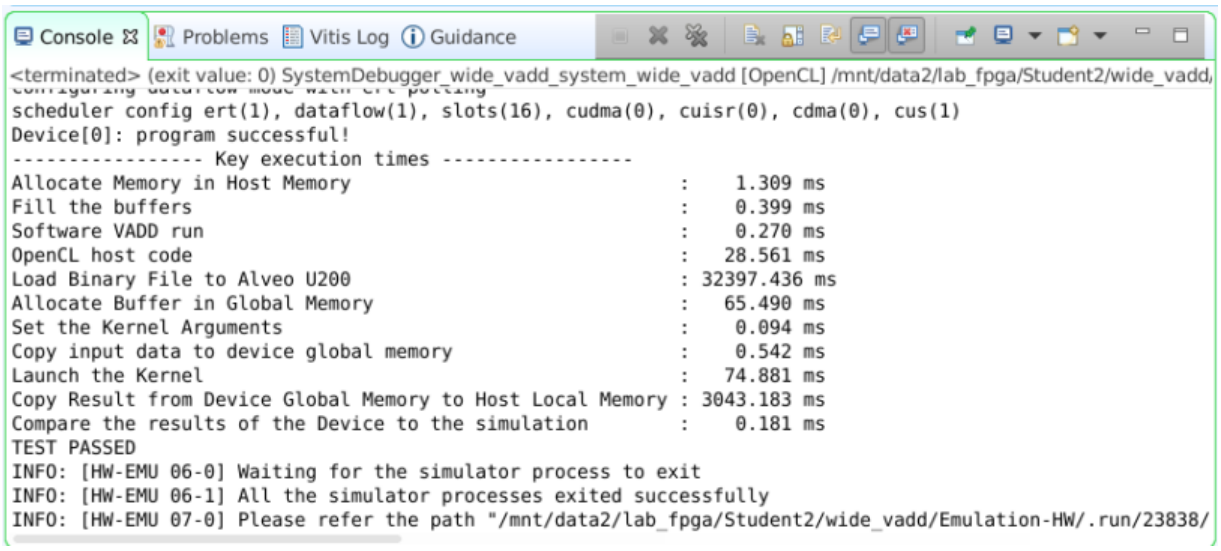
```
void wide_vadd(
  const uint512_dt *in1, // Read-Only Vector 1
  const uint512_dt *in2, // Read-Only Vector 2
  uint512_dt *out,       // Output Result
  int size               // Size in integer
)
```

2. Scroll down further and look where local memories are defined of the same data type and width (512 bits)

```
uint512_dt v1_local[BUFFER_SIZE]; // Local memory to store vector1
uint512_dt v2_local[BUFFER_SIZE];
uint512_dt result_local[BUFFER_SIZE]; // Local Memory to store result
```

Build and run in hardware emulation mode

1. Set *Active build configuration:* to **Emulation-HW**
2. Build in Emulation-HW mode by selecting **wide_vadd_system** in the Explorer view and clicking on the build ( ) button. This will take about 10 minutes
3. After build completes, in the Assistant view select **wide_vadd_system** and click on the Run button then select **Launch HW Emulator. To be noticed that the following times depends on server workload and they can be changed. The simulated time is calculated in Profile Summary (see below).**

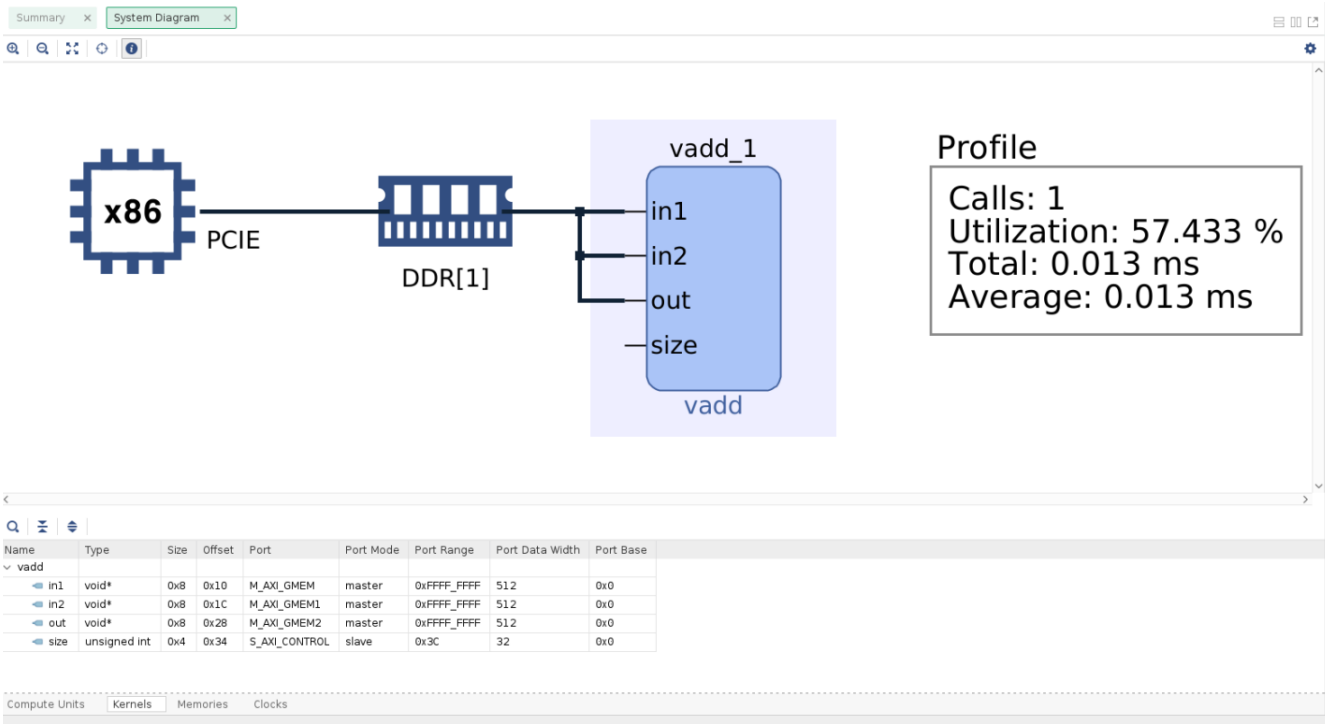


Analyze the generated design

1. In the Assistant view, double-clicking on *wide_vadd_system* > *wide_vadd* > *Emulation-HW* > *SystemDebugger_wide_vadd_system_wide_vadd* > *Run Summary (xclbin)*

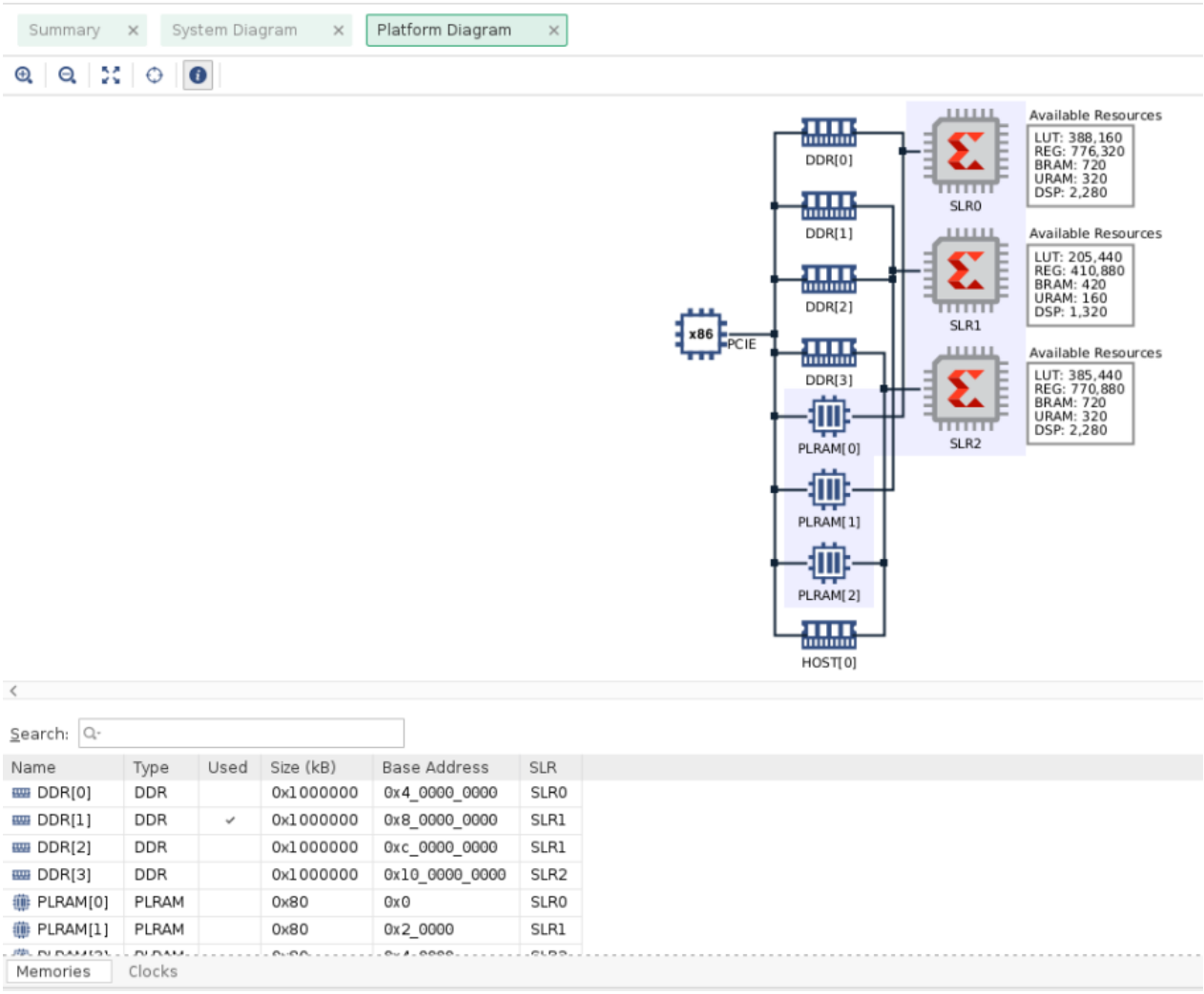
2. Check generated kernel interface

- Select **System Diagram**. Notice that all ports (in1, in2, and out) are using one bank
- Click **Kernels** tab
- Check the `Port Data Width` parameter. All input and output ports are 512 bits wide whereas size (scalar) port is 32 bits wide

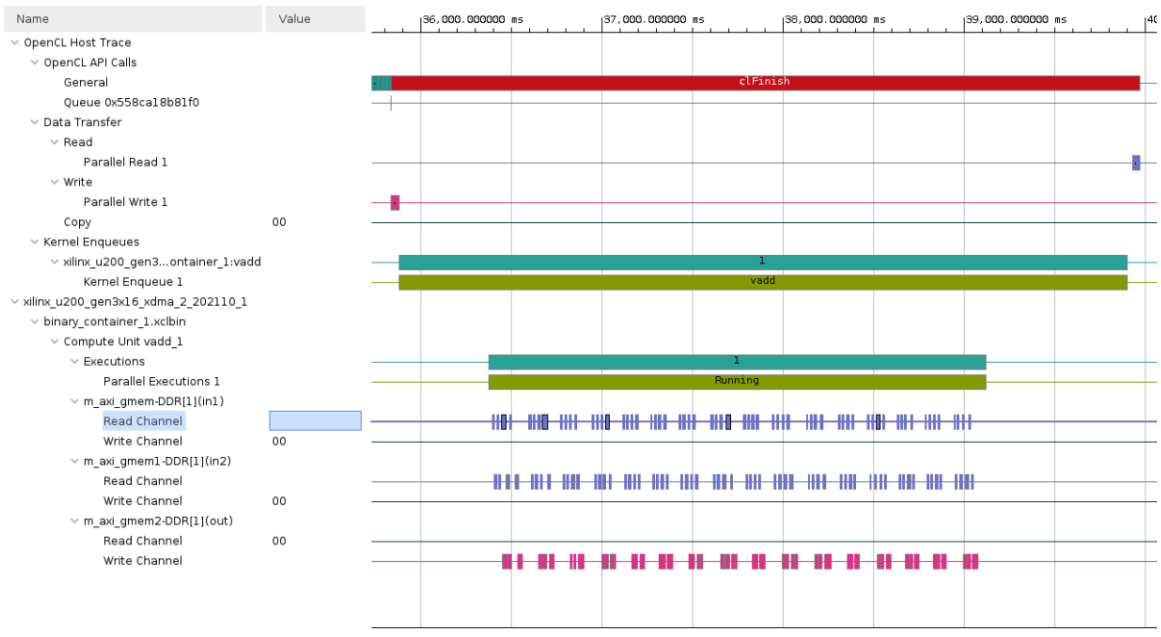


- Select **Platform Diagram** in the left panel

Observe that there are four DDR4 memory banks and three PLRAM banks. In this design, `bank1`, which uses SLR1, is being used for all operands.



3. *Click on* ***Timeline Trace***
   Scroll and zoom to find the data transfers. The three operands share the same AXI4-MM adapter, both inputs compete in read channel (resource contention).
   On the other hand, the write channel is independent but it is still mapped to the same memory bank.

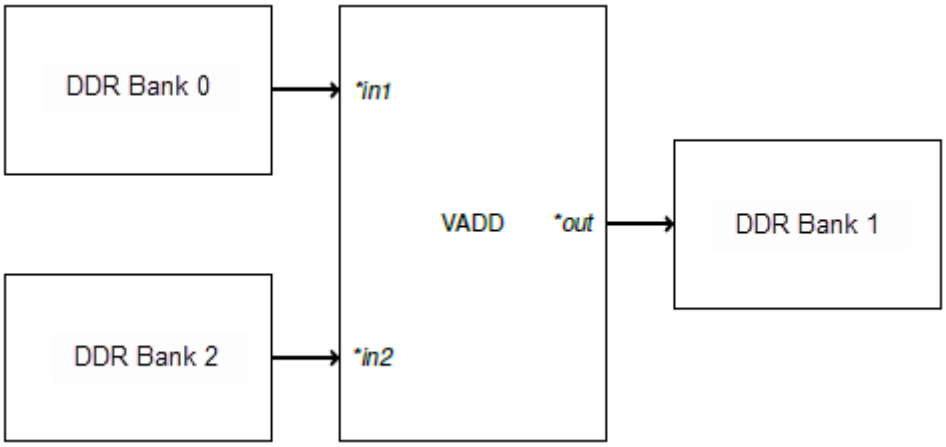


4. The **Profile Summary** reports that the kernel takes 0.022 ms to execute

5. Close the Vitis Analyzer.

Use multiple memory banks

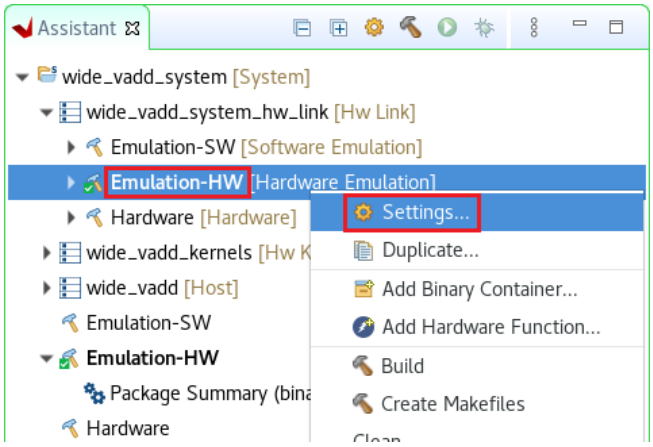
There are four DDR4 memory banks available on the accelerator card. In the previous section, we used only one bank. As we have three operands (two read and one write) it may be possible to improve performance if more memory banks are used simultaneously, providing maximize the bandwidth available to each of the interfaces. So it is possible to use the topology shown in following Figure.



This will provide the ability to perform high-bandwidth transactions simultaneously with different external memory banks. Remember, long bursts are generally better for performance than many small reads and writes, but you cannot fundamentally perform two operations on the memory at the same time.

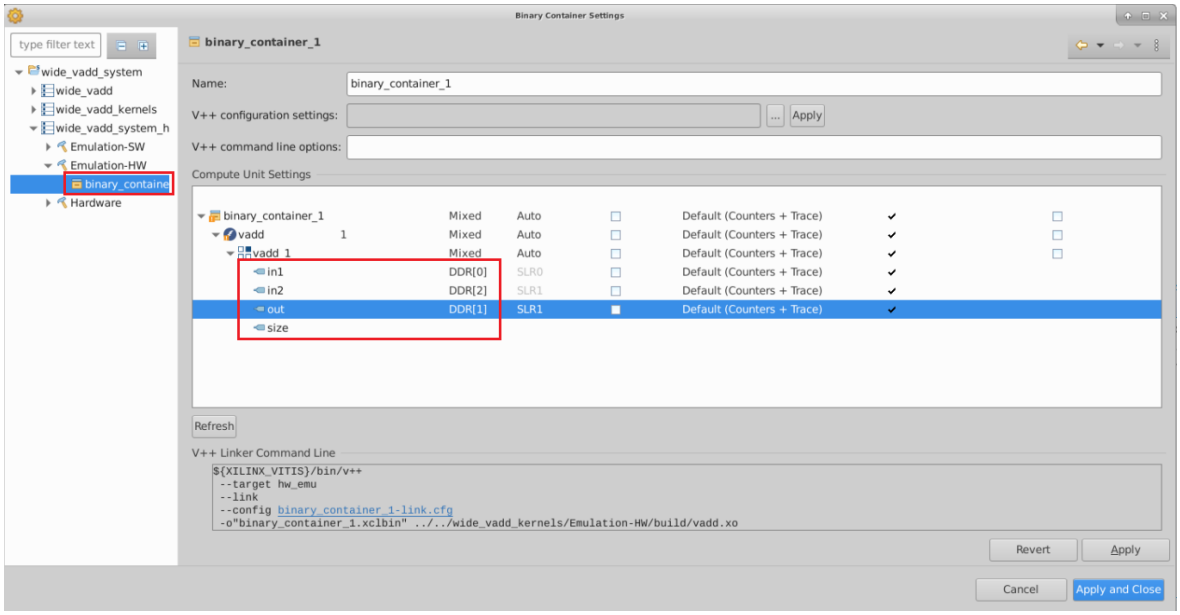
To connect a kernel to multiple memory banks, you need to assign each kernel's port to a memory bank.
Note that DDR controllers may be physically located in different SLRs (Super Logic Regions) on the FPGA. A kernel with routing that crosses SLR regions can be more difficult to build and meet timing. This should be taken into account in a real design, where multiple memory banks are located in different SLRs.

1. Assign memory banks as shown in figure below. In the **Assistant view**, right click on *wide_vadd_system* > *wide_vadd_system_hw_link* > *Emulation-HW* and then click Settings



2. In the Binary Container Settings windows, expand ***wide_vadd_system_hw_link > Emulation-HW*** and click ***binary_continer_1***

3. Assign the arguments of the **krnl_vadd** to the following memory banks
   - in1 to DDR[0]
   - in2 to DDR[2]
   - out to DDR[1]



The SLR column is automatically populated after a memory bank is selected

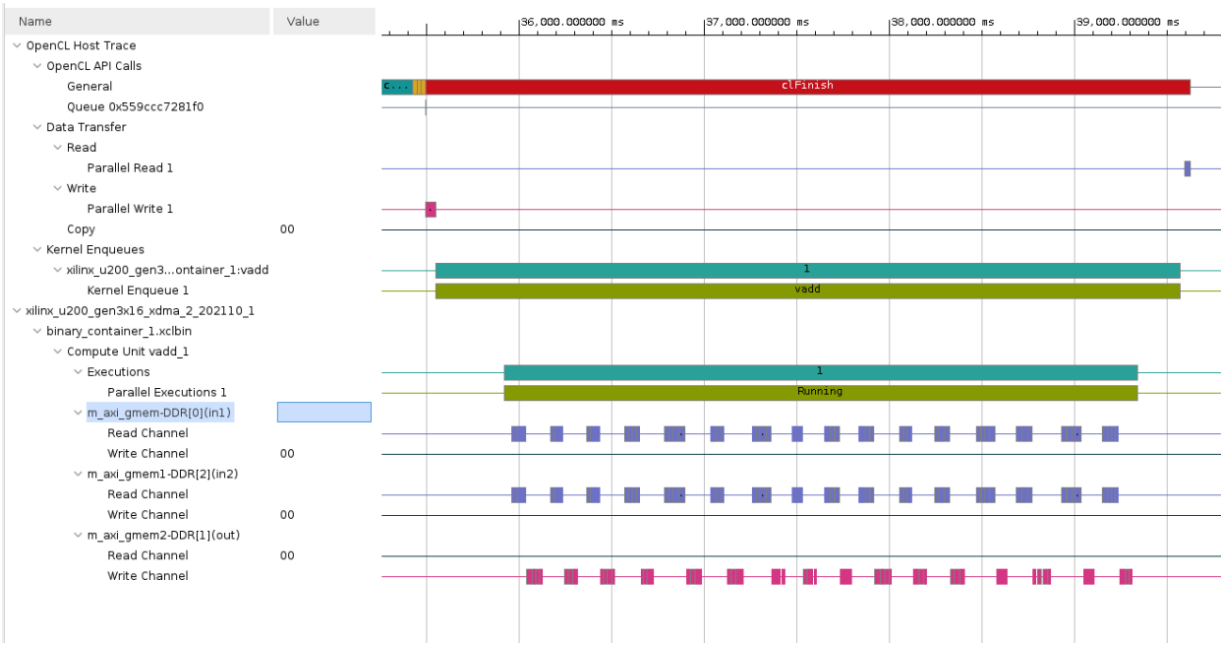
4. Click **Apply and Close**

5. **Rebuild** the project by selecting ***wide_vadd_system*** in the Explorer view and clicking on the build button

6. After build completes, in the **Assistant view** select **wide_vadd_system** and click on the **Run button** and then select SystemDebugger_wide_vadd_system (System Project Debug)

7. In the Assistant view, double-clicking on *wide_vadd_system* > *wide_vadd* > *Emulation-HW* > *SystemDebugger_wide_vadd_system_wide_vadd* > *Run Summary (xclbin)*

8. *Click on **Timeline Trace***

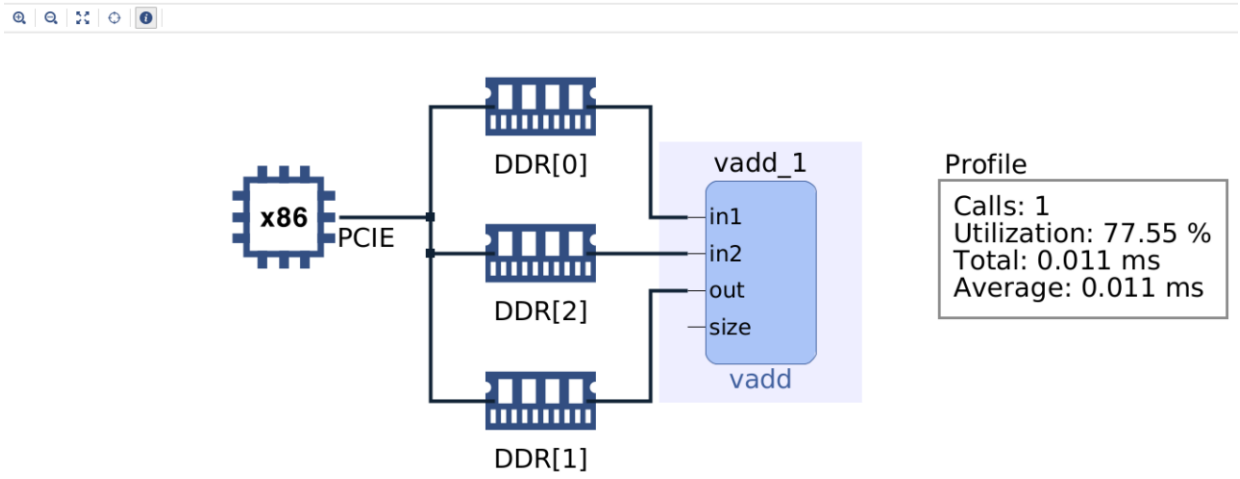


Note that now each argument is mapped to a different memory bank and there is overlap in the read operations.

9. Click on **Profile Summary** and get the Kernel execution time (0.0147 ms).

10. Open **System Diagram**

Notice all ports (in1, in2, and out) are using different memory banks



Name	Type	Size	Offset	Port	Port Mode	Port Range	Port Data Width	Port Base
vadd								
in1	void*	0x8	0x10	M_AXI_GMEM	master	0xFFFF_FFFF	512	0x0
in2	void*	0x8	0x1C	M_AXI_GMEM1	master	0xFFFF_FFFF	512	0x0
out	void*	0x8	0x28	M_AXI_GMEM2	master	0xFFFF_FFFF	512	0x0
size	unsigned int	0x4	0x34	S_AXI_CONTROL	slave	0x3C	32	0x0

11. Close **Vitis Analyzer**

Conclusion

From a simple vadd application, we explored steps to optimize kernel and system performance by:

- Using Vitis HLS directives to use multiple AXI4-MM adapters, widen the kernel buses and unrolling the computation loop.
- Assign dedicated memory controller for the arguments
- Use Vitis Analyzer to view the result

The kernel was highly optimized as well as the system. However, the data movement is dominant. To truly achieve acceleration the application has to have a high compute intensity.

The compute intensity ratio is defined as:

compute intensity = compute operations / memory accesses

The bigger this number is, the more opportunities to achieve acceleration.