XUP Vitis Labs (2019.2)

Improving Performance

Introduction

In GUI Flow 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. Source Vitis environment and set environment variables

   If you have opened a new terminal window then execute following two shell scripts and set LIBRARY_PATH

   ```
   source ~/aws-fpga/vitis_setup.sh
   source ~/aws-fpga/vitis_runtime_setup.sh
   export LIBRARY_PATH=/usr/lib/x86_64-linux-gnu
   ```

2. Launch Vitis GUI

   Invoke GUI by executing the following command:

   `vitis &`

   Continue with the workspace you have used in previous lab

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

You should see `xilinx_aws-vu9p-f1_shell-v04261818_201920_1` as one of the platforms if you are continuing with previous lab, otherwise add it from `~/aws-fpga/Vitis/aws_platform`

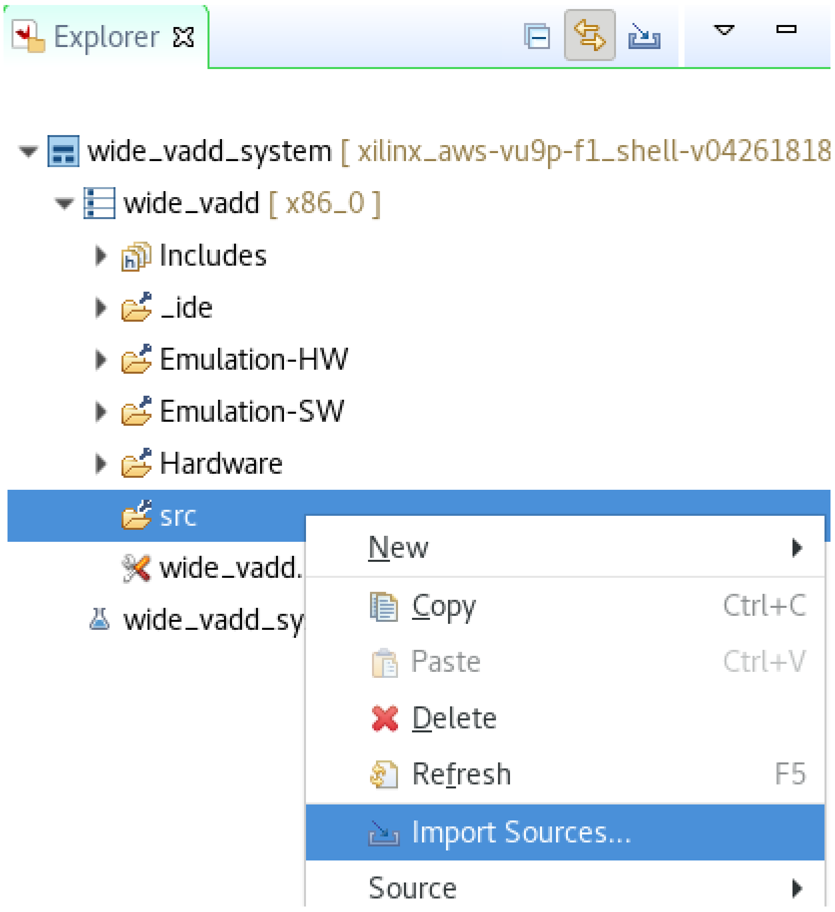
4. Select `Empty Application` as the template and click **Finish**

   The project is generated

5. Import provided source files from `~/sources/improving_performance_lab/wide_vadd` sub-directories

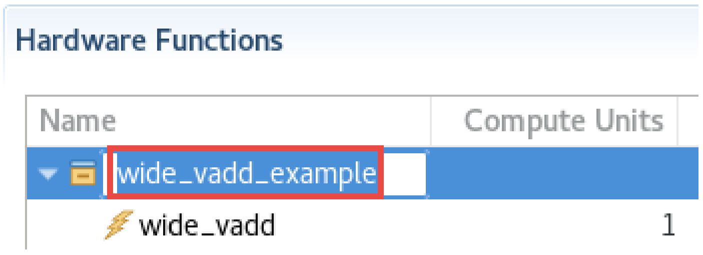
   Select `src` directory in Explorer view, right click and select `Import Sources...`

   - Import `improving_performance_lab/wide_vadd/hw_src/wide_vadd_krnl.cpp` for hardware accelerator
   - Import all `*.cpp` and `*.hpp` files in `improving_performance_lab/wide_vadd/sw_src/` for host code application



6. Within *Project Editor* view click on ⚡ in the *Hardware Functions* window and add `wide_vadd` function as a *Hardware Function* (kernel)

7. Change the binary container name to `wide_vadd_example` by clicking on it once and typing over. This is necessary as the name is hard-coded in `wide_vadd.cpp` on line 69



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 62-67 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 at lines 78-80 where local memories are defined of the same data type and width (512 bits)

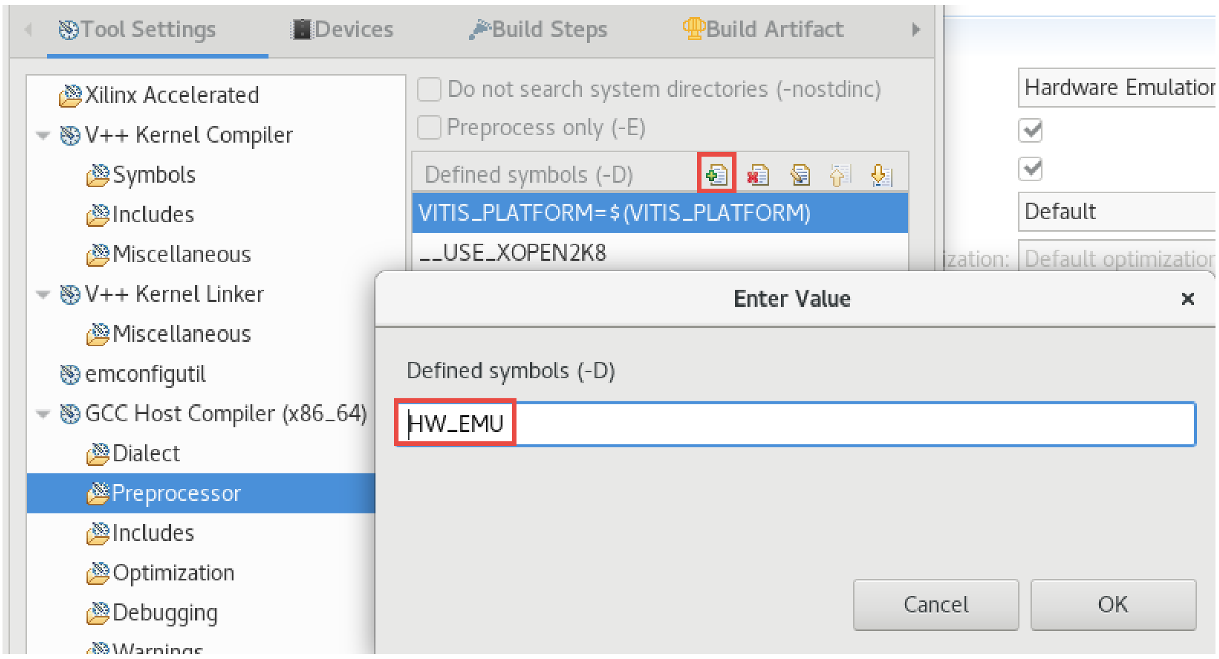
   ```
   uint512_dt v1_local[BUFFER_SIZE]; // Local memory to store vector1
   uint512_dt v2_local[BUFFER_SIZE];
   uint512_dt result_local[BUFFER_SIZE]; // Local Memory to store result
   ```

Setup Hardware Emulation

1. Set *Active build configuration:* to **Emulation-HW**

2. Set `HW_EMU` in the host code to use smaller data size (1,024 times smaller) as input to save emulation time (`wide_vadd.cpp` line 41)

   - Right click `wide_vadd` application in Explorer view, select `C/C++ Build Settings`
   - In the left-hand side view select `C/C++ Build > Settings`.
   - In `GCC Host Compiler (x86_64) > Preprocessor`, add `HW_EMU` in `defined symbols` window, and enter **OK**
   - Click **Apply and Close**
   - Click **Yes** to rebuild



Do not search system directories (-nostdinc)
Preprocess only (-E)
Defined symbols (-D)

VITIS_PLATFORM=$(VITIS_PLATFORM)
__USE_XOPEN2K8

Hardware Emulation
Default
Default optimization

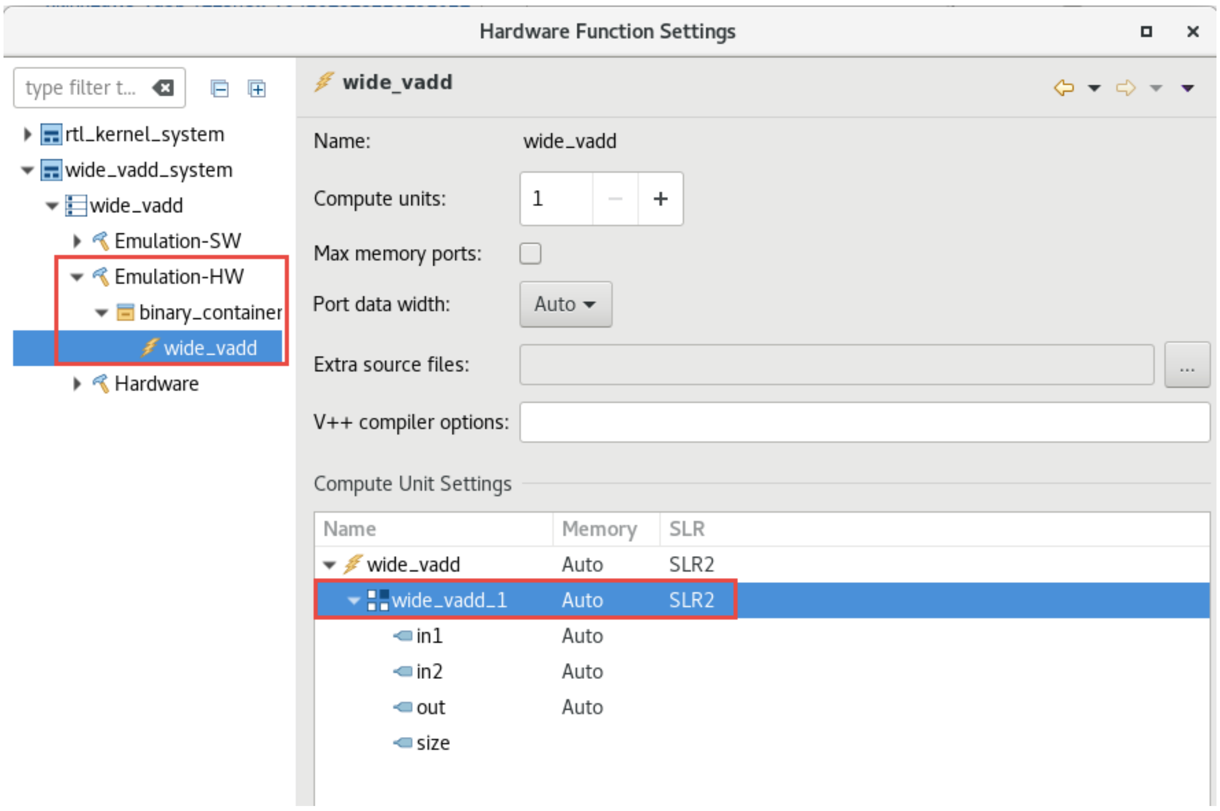
Enter Value

Defined symbols (-D)

HW_EMU

3. Set dedicated location of kernel and memory interface

   - Right click `wide_vadd > Emulation-HW` in *Assistant* view, select `Settings`
   - Navigate to *wide_vadd* kernel, adjust memory and SLR settings according to the screenshot below
   - Click Apply and Close



Name: wide_vadd
Compute units: 1
Max memory ports:
Port data width: Auto
Extra source files:
V++ compiler options:

Compute Unit Settings

Name	Memory	SLR
wide_vadd	Auto	SLR2
wide_vadd_1	Auto	SLR2
in1	Auto	
in2	Auto	
out	Auto	
size		

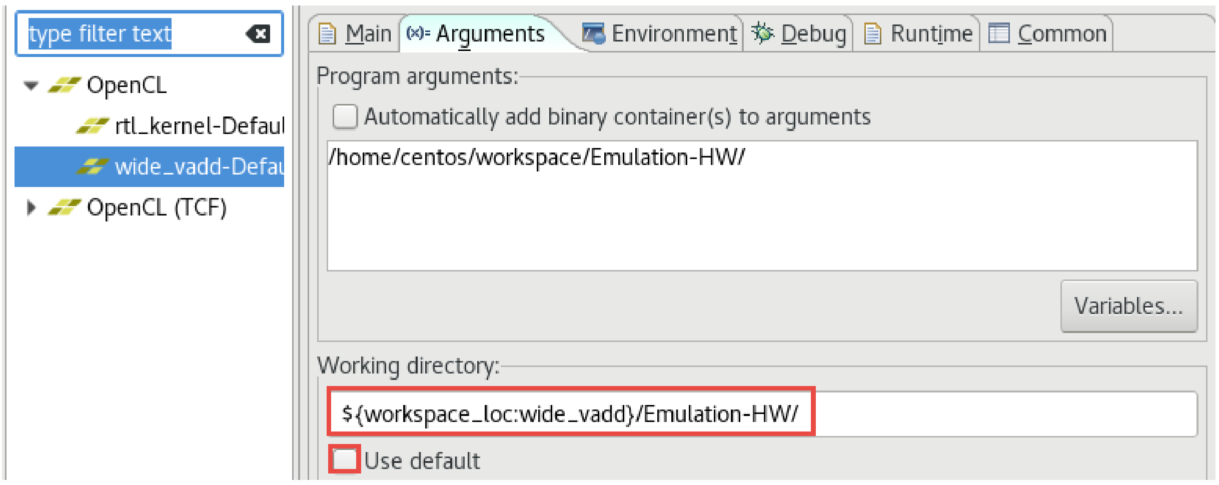
Build and run in hardware emulation mode

1. Build in Emulation-HW mode

   This will take about 10 minutes

2. After build completes, open *Run Configurations* window

3. Set `Working Directory` at Arguments tab (of Run Configurations) to `${workspace_loc:wide_vadd}/Emulation-HW/` by unchecking *Use default* check-box, and backspacing the directory path. This is because the xclbin location is hard coded in host source code

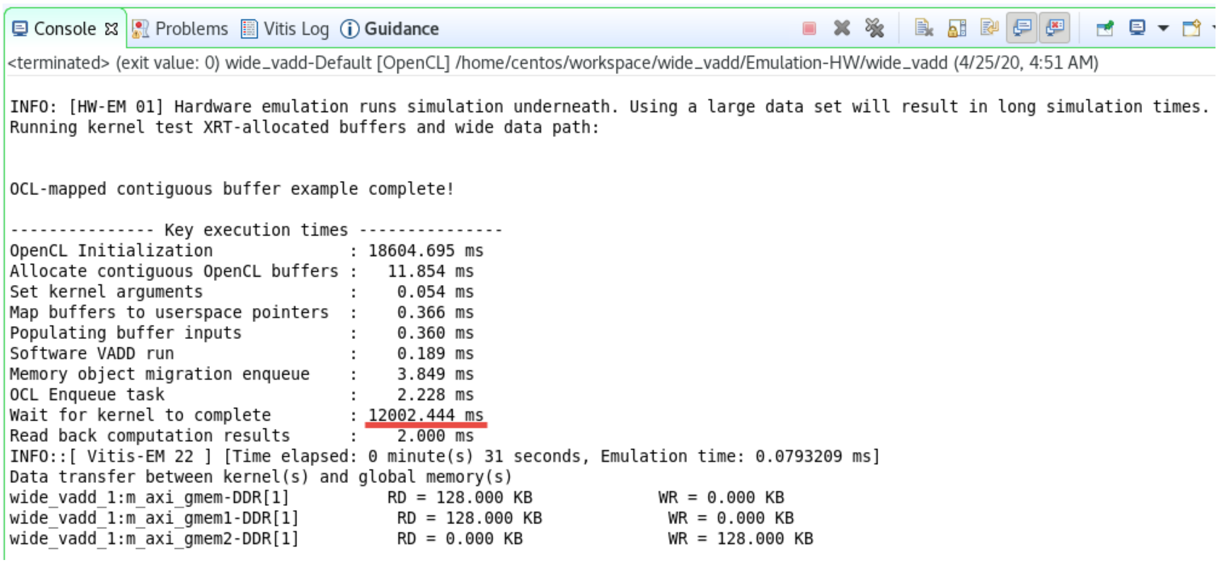


Program arguments:
Automatically add binary container(s) to arguments
/home/centos/workspace/Emulation-HW/

Working directory:
${workspace_loc:wide_vadd}/Emulation-HW/
Use default

4. Click **Apply** and then **Run**

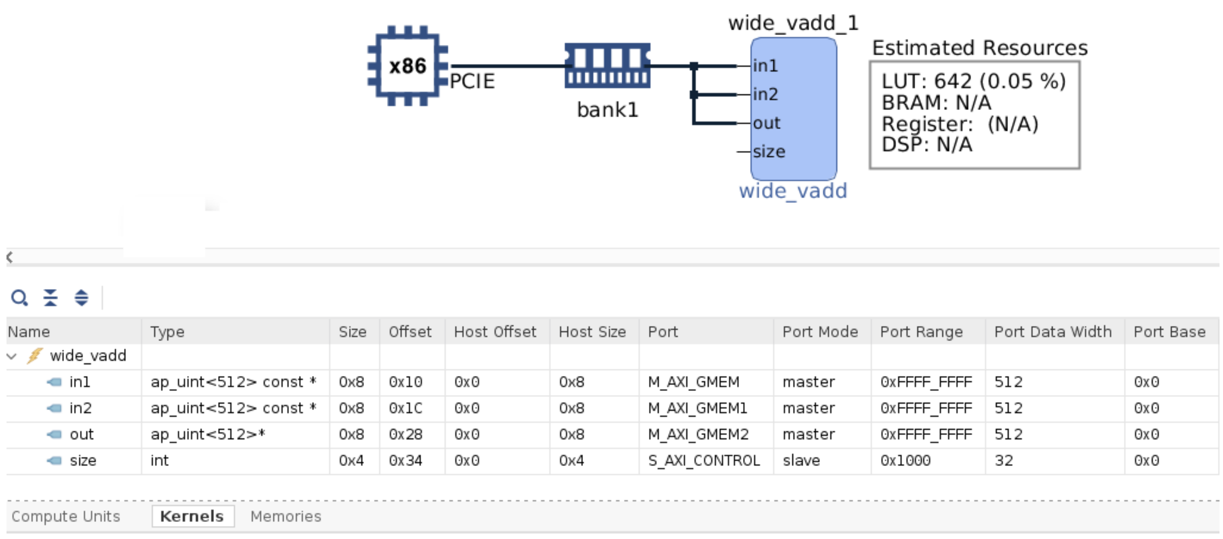
   Notice the kernel wait time is about 12 seconds.



```
<terminated> (exit value: 0) wide_vadd-Default [OpenCL] /home/centos/workspace/wide_vadd/Emulation-HW/wide_vadd (4/25/20, 4:51 AM)

INFO: [HW-EM 01] Hardware emulation runs simulation underneath. Using a large data set will result in long simulation times.
Running kernel test XRT-allocated buffers and wide data path:


OCL-mapped contiguous buffer example complete!

-------------- Key execution times --------------
OpenCL Initialization              : 18604.695 ms
Allocate contiguous OpenCL buffers :   11.854 ms
Set kernel arguments               :    0.054 ms
Map buffers to userspace pointers  :    0.366 ms
Populating buffer inputs           :    0.360 ms
Software VADD run                  :    0.189 ms
Memory object migration enqueue    :    3.849 ms
OCL Enqueue task                   :    2.228 ms
Wait for kernel to complete        : 12002.444 ms
Read back computation results      :    2.000 ms
INFO::[ Vitis-EM 22 ] [Time elapsed: 0 minute(s) 31 seconds, Emulation time: 0.0793209 ms]
Data transfer between kernel(s) and global memory(s)
wide_vadd_1:m_axi_gmem-DDR[1]          RD = 128.000 KB              WR = 0.000 KB
wide_vadd_1:m_axi_gmem1-DDR[1]          RD = 128.000 KB              WR = 0.000 KB
wide_vadd_1:m_axi_gmem2-DDR[1]          RD = 0.000 KB                WR = 128.000 KB
```

5. Check generated kernel interface

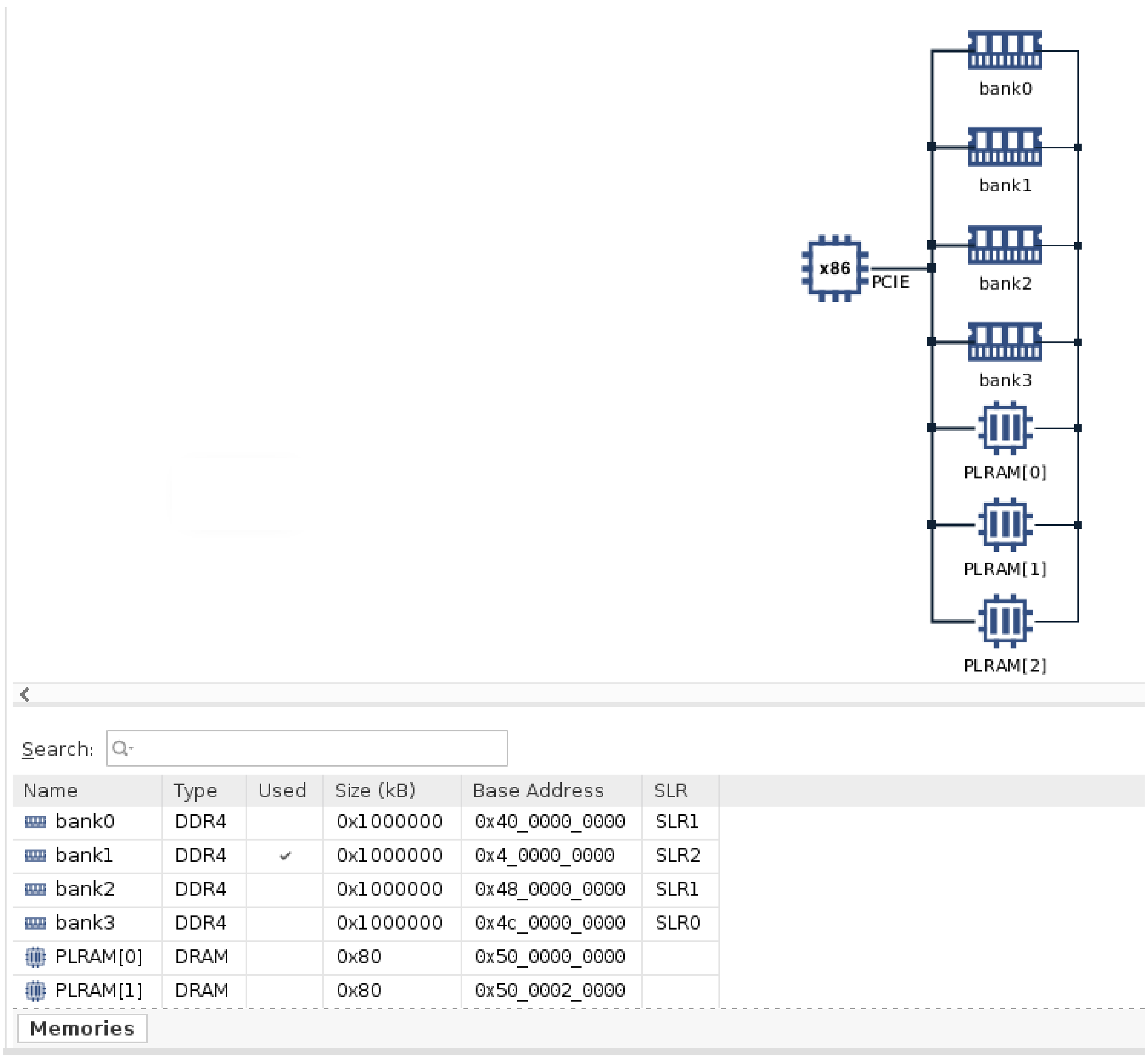
- Open Link Summary with Vitis Analyzer by expanding `Emulation-HW > wide_vadd_example` and double-clicking `Link Summary`
- Select **System Diagram**. Notice that all ports (in1, in2, and out) are using one bank
- Click **Kernels** tab
- Check the `Port Data Width` parameter. All input and output ports are 512 bits wide whereas size (scalar) port is 32 bits wide



Name	Type	Size	Offset	Host Offset	Host Size	Port	Port Mode	Port Range	Port Data Width	Port Base
wide_vadd										
in1	ap_uint<512> const *	0x8	0x10	0x0	0x8	M_AXI_GMEM	master	0xFFFF_FFFF	512	0x0
in2	ap_uint<512> const *	0x8	0x1C	0x0	0x8	M_AXI_GMEM1	master	0xFFFF_FFFF	512	0x0
out	ap_uint<512>*	0x8	0x28	0x0	0x8	M_AXI_GMEM2	master	0xFFFF_FFFF	512	0x0
size	int	0x4	0x34	0x0	0x4	S_AXI_CONTROL	slave	0x1000	32	0x0

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

Observe that there are four DDR4 memory banks and two PLRAM banks. In this design, `bank1`, which uses SLR2, is being used for all operands. Also notice that `bank0` and `bank2` are connected to SLR1



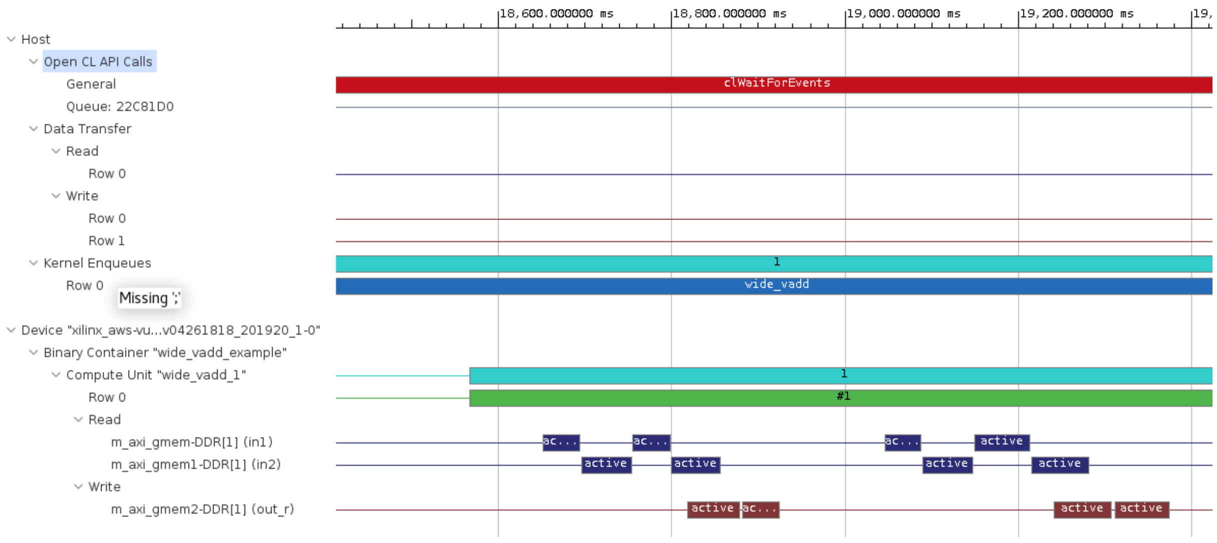
Name	Type	Used	Size (kB)	Base Address	SLR
bank0	DDR4		0x1000000	0x40_0000_0000	SLR1
bank1	DDR4	✓	0x1000000	0x4_0000_0000	SLR2
bank2	DDR4		0x1000000	0x48_0000_0000	SLR1
bank3	DDR4		0x1000000	0x4c_0000_0000	SLR0
PLRAM[0]	DRAM		0x80	0x50_0000_0000	
PLRAM[1]	DRAM		0x80	0x50_0002_0000	

1. Close the Vitis Analyzer.

2. Run Vitis Analyzer on timing trace

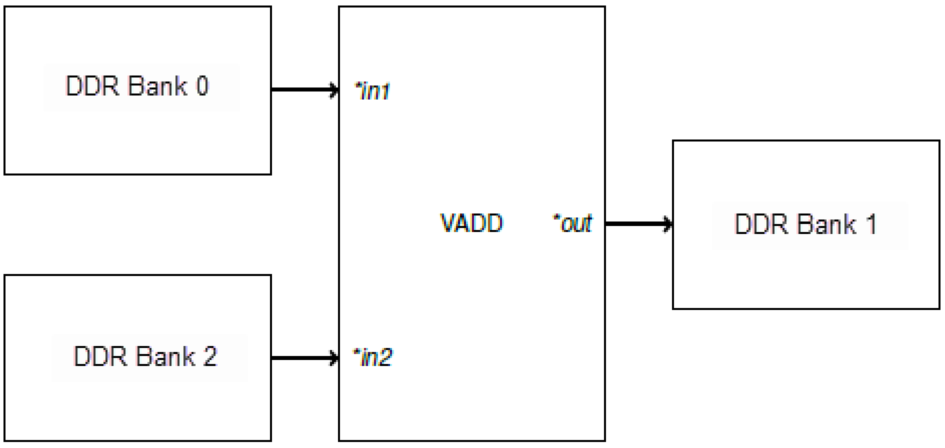
```
cd /home/centos/workspace/wide_vadd/Emulation-HW
vitis_analyzer timeline_trace.csv
```

3. Zoom into area where data transfer on various ports of the kernel is taking place and observe the sequential data transfer between two input operands and a result since only one memory controller is being used



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.

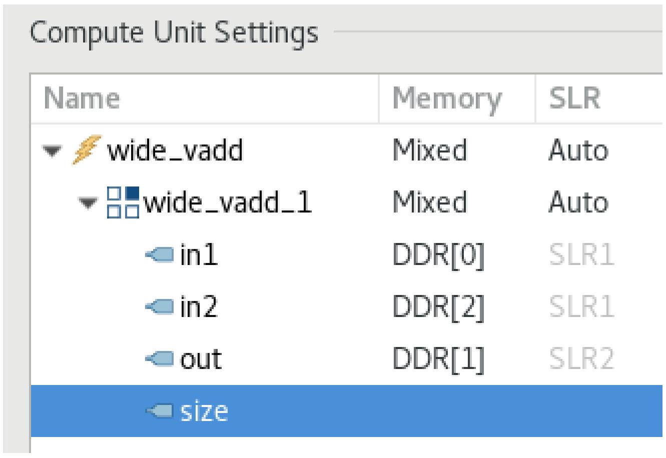
To connect a kernel to multiple memory banks, you need to

1. Assign the kernel's interface to a memory controller
2. Assign the kernel to an SLR region

Please note that since the DDR controllers are constrained to different SLR (Super Logic Region), the routing between two SLR may have some challenges in timing closure when the design is compiled for bitstream. This technique is valuable in the cases where one SLR has two DDR controllers.

1. Assign memory banks as shown in figure below

    - Right click `wide_vadd > Emulation-HW` in *Assistant* view, select `Settings`
    - Navigate to `wide_vadd` kernel, and adjust memory and SLR settings
    - Click Apply and Close



Compute Unit Settings

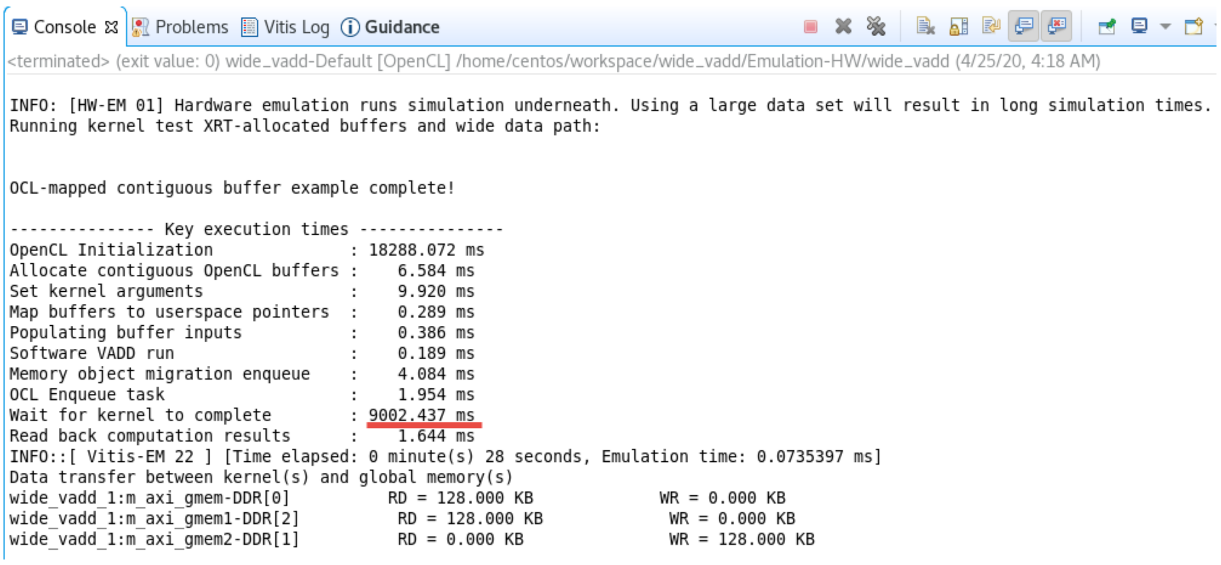
Name	Memory	SLR
wide_vadd	Mixed	Auto
wide_vadd_1	Mixed	Auto
in1	DDR[0]	SLR1
in2	DDR[2]	SLR1
out	DDR[1]	SLR2
size		

2. Build Emulation-HW

    This will take about 10 minutes. After build completes, open Run Configurations window

3. Click **Run**

    Notice that the kernel wait time has reduced from about 12 seconds (single memory bank) to 9 seconds (two memory banks) indicating performance improvement



```
<terminated> (exit value: 0) wide_vadd-Default [OpenCL] /home/centos/workspace/wide_vadd/Emulation-HW/wide_vadd (4/25/20, 4:18 AM)

INFO: [HW-EM 01] Hardware emulation runs simulation underneath. Using a large data set will result in long simulation times.
Running kernel test XRT-allocated buffers and wide data path:


OCL-mapped contiguous buffer example complete!

--------------- Key execution times ---------------
OpenCL Initialization              : 18288.072 ms
Allocate contiguous OpenCL buffers :    6.584 ms
Set kernel arguments               :    9.920 ms
Map buffers to userspace pointers  :    0.289 ms
Populating buffer inputs           :    0.386 ms
Software VADD run                  :    0.189 ms
Memory object migration enqueue    :    4.084 ms
OCL Enqueue task                   :    1.954 ms
Wait for kernel to complete        : 9002.437 ms
Read back computation results      :    1.644 ms
INFO::[ Vitis-EM 22 ] [Time elapsed: 0 minute(s) 28 seconds, Emulation time: 0.0735397 ms]
Data transfer between kernel(s) and global memory(s)
wide_vadd_1:m_axi_gmem-DDR[0]          RD = 128.000 KB              WR = 0.000 KB
wide_vadd_1:m_axi_gmem1-DDR[2]          RD = 128.000 KB              WR = 0.000 KB
wide_vadd_1:m_axi_gmem2-DDR[1]          RD = 0.000 KB                WR = 128.000 KB
```

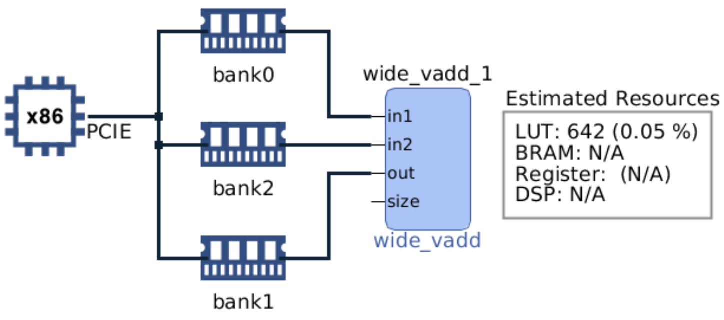
4. Check generated kernel interface

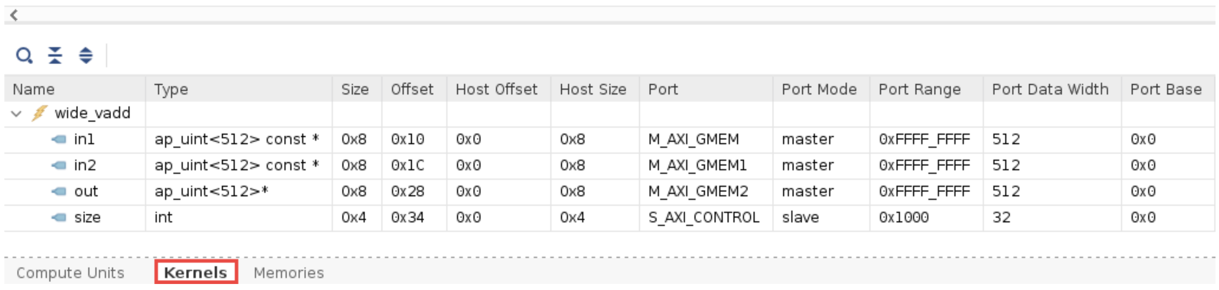
- Open Link Summary with Vitis Analyzer by expanding `Emulation-HW > wide_vadd_example` and double-clicking `Link Summary`

- Select System Diagram

- Click *Kernels* tab

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



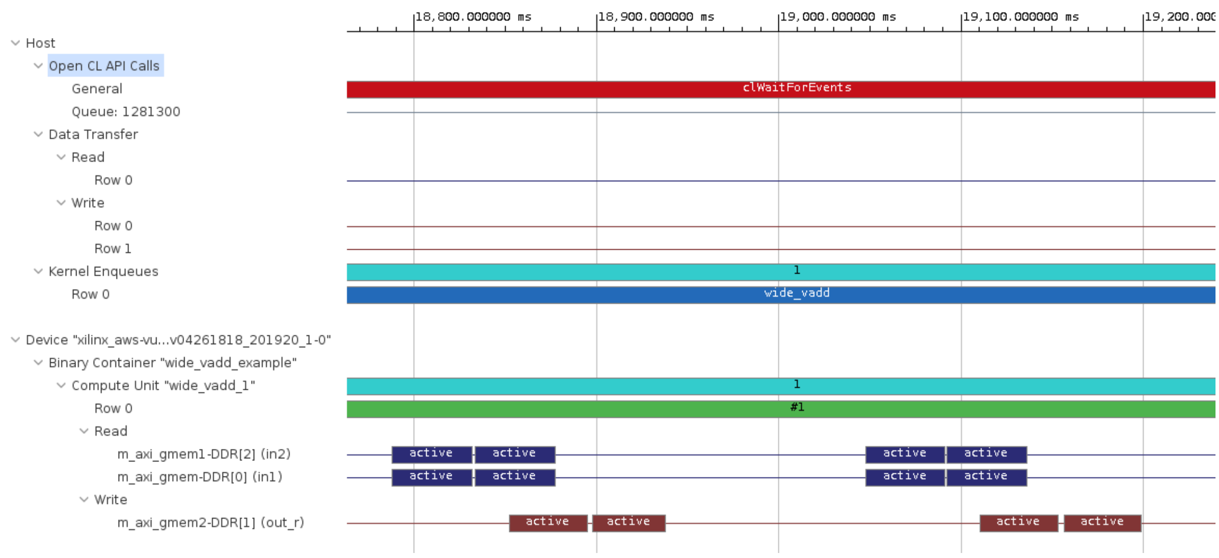


Name	Type	Size	Offset	Host Offset	Host Size	Port	Port Mode	Port Range	Port Data Width	Port Base
wide_vadd										
in1	ap_uint<512> const *	0x8	0x10	0x0	0x8	M_AXI_GMEM	master	0xFFFF_FFFF	512	0x0
in2	ap_uint<512> const *	0x8	0x1C	0x0	0x8	M_AXI_GMEM1	master	0xFFFF_FFFF	512	0x0
out	ap_uint<512>*	0x8	0x28	0x0	0x8	M_AXI_GMEM2	master	0xFFFF_FFFF	512	0x0
size	int	0x4	0x34	0x0	0x4	S_AXI_CONTROL	slave	0x1000	32	0x0

1. Run Vitis Analyzer on timing trace

```
cd /home/centos/workspace/wide_vadd/Emulation-HW
vitis_analyzer timeline_trace.csv
```

2. Zoom into area where data transfer on various ports of the kernel is taking place and observe that data fetching is taking place in parallel and result is written overlapping the fetching data, increasing the bandwidth



3. Close Vitis Analyzer

Conclusion

From a simple vadd application, we explored several steps to increase system performance:

- Expand kernel interface width
- Assign dedicated memory controller
- Use Vitis Analyzer to view the result

Start the next lab: Optimization Lab