FPGA Based Accelerators Design (Assignment-1)

Roll No: 20173071
Name: Aditya Saripalli

Q1. What is the FPGA used on the Amazon F1 instance? List down the important available hardware resources such as the number of LUTs, Flip Flops, DSPs, BRAM block, device memory size etc. Similarly list down the host CPU configuration like processor, clock frequency, main memory size, cache size etc.

Ans) Amazon EC2 F1 instances use FPGAs to enable delivery of custom hardware accelerations. F1 instances are easy to program and come with everything you need to develop, simulate, debug, and compile your hardware acceleration code, including an FPGA Developer AMI and supporting hardware level development on the cloud.

AWS EC2 F1 FPGA Features:

- Xilinx Virtex UltraScale+ VU9P FPGAs
- 64GB of ECC-protected memory on 4 x DDR4 RAMs
- Dedicated PCIe Gen3 x16 interface
- 2.5 million+ Logical Units
- 6,800 Digital Signal Processing (DSP) engines

There are 3 different AWS F1 FPGA instances that anyone can use. The details of which is given in the table below:

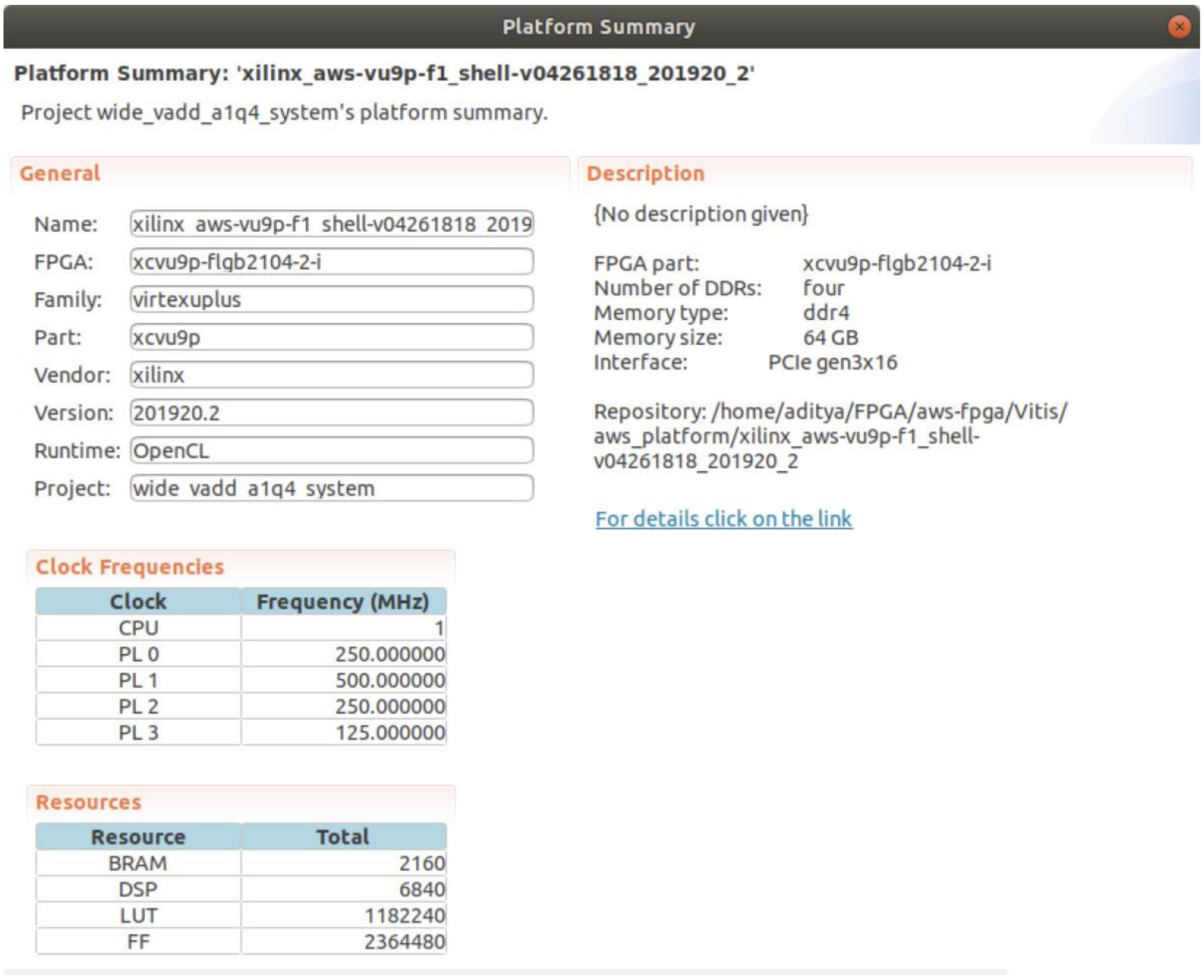
Instance	FPGAs	vCPU	Mem (GB)	SSD Storage (GB)	Networking (Gbps)
f1.2xlarge	1	8	122	470	up to 10
f1.4xlarge	2	16	244	940	up to 10
f1.16xlarge	8	64	976	4 x 940	25

For **f1.16xlarge** instances, the dedicated PCI-e fabric lets the FPGAs share the same memory space and communicate with each other across the fabric at up to 12 Gbps in each direction.

All the f1 instances have the following additional specs:

- 2.3 GHz (base) and 2.7 GHz (turbo) Intel Xeon E5-2686 v4 Processor
- Intel AVX / AVX2 (Advanced Vector Extension), Intel Turbo
- EBS (Elastic Block Store) Optimised
- Enhanced Networking

The actual count of the HW units of the AWS F1 FPGA can be seen from the "platform summary" of – xilinx_aws-vu9p-f1_shell-v04261818-201920_2, which we can get from Vitis, as shown below:


Platform Summary

Platform Summary: 'xilinx_aws-vu9p-f1_shell-v04261818_201920_2'

Project wide_vadd_a1q4_system's platform summary.

General

Name:	xilinx_aws-vu9p-f1_shell-v04261818_2019
FPGA:	xcvu9p-flgb2104-2-i
Family:	virtexuplus
Part:	xcvu9p
Vendor:	xilinx
Version:	201920.2
Runtime:	OpenCL
Project:	wide_vadd_a1q4_system

Description

{No description given}

FPGA part:	xcvu9p-flgb2104-2-i
Number of DDRs:	four
Memory type:	ddr4
Memory size:	64 GB
Interface:	PCIe gen3x16

Repository: /home/aditya/FPGA/aws-fpga/Vitis/aws_platform/xilinx_aws-vu9p-f1_shell-v04261818_201920_2

For details click on the link

Clock Frequencies

Clock	Frequency (MHz)
CPU	1
PL 0	250.000000
PL 1	500.000000
PL 2	250.000000
PL 3	125.000000

Resources

Resource	Total
BRAM	2160
DSP	6840
LUT	1182240
FF	2364480

On the Host side we use the **"m5.2xlarge"** instance for our development activities (like, writing code, SW/HW Emulations, Hardware creation, AFI creation etc). Its configuration details are as mentioned below:

m5.2xlarge runs with:

- 3.1 GHz Intel Xeon® Platinum 8175M processors with new Intel Advanced Vector Extension (AVX-512) instruction set.
- 8 vCPU cores.
- Each vCPU is a thread of either an Intel Xeon core or an AMD EPYC core.
- 32 GB RAM.
- Storage EBS only (up to 4,750 Mbps).
- Network bandwidth (up to 10 Gbps).

AWS m5 instances are widely used for small and mid-size databases, data processing tasks that require additional memory, caching fleets, and for running backend servers for SAP, Microsoft SharePoint, cluster computing, and other enterprise applications.

Q2. Find the PCIe bandwidth to the FPGA device on the F1 instance using the XRT command "xbutil"

Ans) Xilinx Board Utility (xbutil) is a standalone command line utility that is included with the Xilinx Run Time (XRT) installation package. It includes multiple commands to validate and identify the installed card(s), along with additional card details including DDR, PCIe, shell name (DSA), and system information. This tool can be used for both card administration and application debugging. The **xbutil** command line format is:

```
xbutil <command> [options]
```

To start with run the command **"xbutil query -d 0"** to get the complete information regarding the connected devices. An excerpt of the output is shown below:

```
[ec2-user@ip-172-31-88-240]$ xbutil query -d 0
...
...
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Shell                           FPGA                            IDCode
xilinx_aws-vu9p-f1_shell-v04261818_201920_2AWS VU9P                        0x0
Vendor          Device          SubDevice       SubVendor       SerNum
0x1d0f          0xf010          0x1d51          0xfedd
DDR size        DDR count       Clock0          Clock1          Clock2
64 GB           4               250             250             500
PCIe            DMA chan(bidir) MIG Calibrated  P2P Enabled     OEM ID
GEN 3x16        4               false           false           (N/A)
DNA                             CPU_AFFINITY    HOST_MEM size   Max HOST_MEM
                                0-7             0 Byte          0 Byte
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
...
INFO: xbutil query succeeded.
[ec2-user@ip-172-31-88-240]$
```

The above output shows that the Host is connected to the FPGA device using **PCIe Gen3 x16** link with **4 DMA channels**.

The PCIe bandwidths for DMA read/write transfers to the DDR RAM(s) can be found by running the following command:

```
[ec2-user@ip-172-31-88-240]$ xbutil dmatest -d 0 -b 0x800
INFO: Found total 1 card(s), 1 are usable
INFO: DMA test on [0]: xilinx_aws-vu9p-f1_shell-v04261818_201920_2
Total DDR size: 65536 MB
Buffer Size: 2 MB
Reporting from mem_topology:
Data Validity & DMA Test on bank0
Host -> PCIe -> FPGA write bandwidth = 7326.544593 MB/s
Host <- PCIe <- FPGA read bandwidth = 12175.438075 MB/s
Data Validity & DMA Test on bank2
Host -> PCIe -> FPGA write bandwidth = 7225.986737 MB/s
Host <- PCIe <- FPGA read bandwidth = 12177.790992 MB/s
INFO: xbutil dmatest succeeded.
[ec2-user@ip-172-31-88-240]$
```

Q3. For the vadd, wide_vadd programs from the Vitis Tutorials repository, plot the following metrics in the form of graph.

a) For increasing vector sizes (N = 2^{10}, 2^{11}, 2^{12}, …), find the kernel computation time and total communication time. Plot CPU vector addition time with FPGA vector addition time (include both computation and communication cost).
b) Repeat the above, by replacing the add operation with floating-point multiplication.

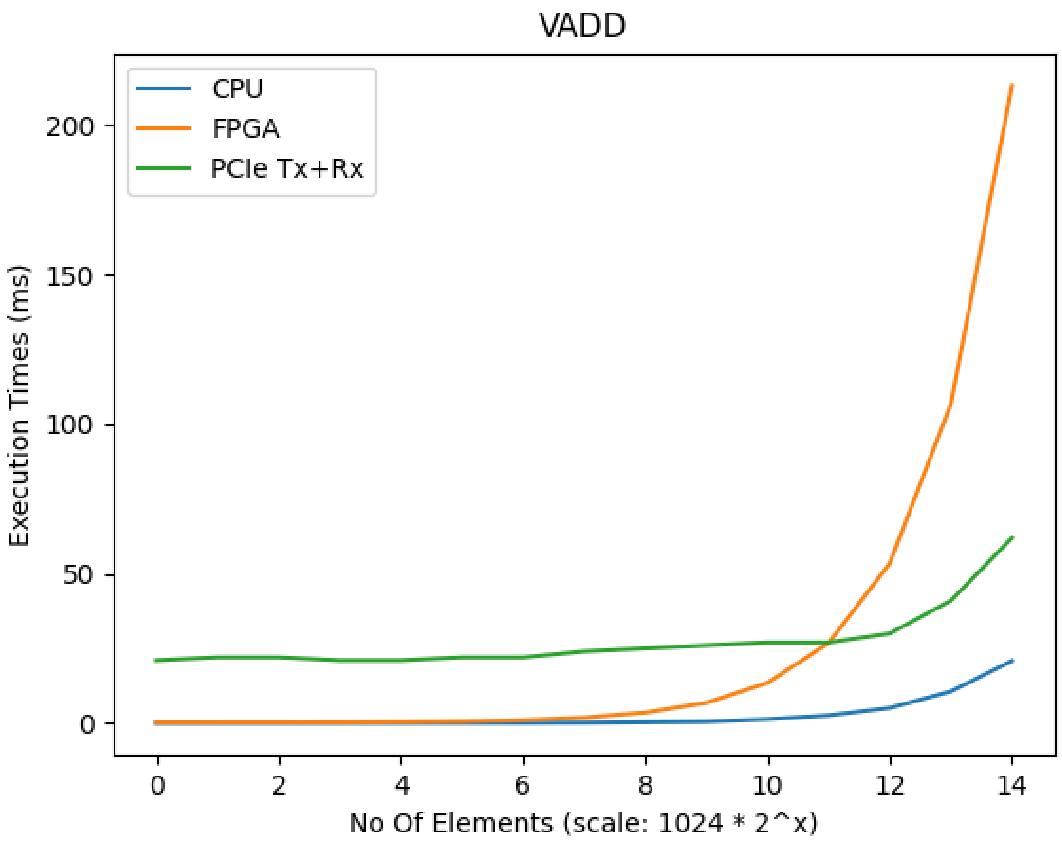
Ans 3(a) The "VADD" version of the program is the vanilla version of the vector addition program. We perform normal vector addition using CPU (using normal for loop) and add a kernel with 2 input vectors and an output vector for storing the result. The data vectors need to be transferred to and from the DDR banks in the FPGA domain. The Host and Kernel code snippets for VADD are as shown below.

HOST & FPGA:

```
void vadd_sw(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t size)
{
    for (uint32_t i = 0; i < size; i++)
    {
        c[i] = a[i] + b[i];
    }
}
void krnl_vadd(int *in1, int *in2, int *out_r, unsigned int size)
{
    int v1_buffer[BUFFER_SIZE];   // Local memory to store vector1

    //Per iteration of this loop perform BUFFER_SIZE vector addition
    for (unsigned int i = 0; i < size; i += BUFFER_SIZE)
    {
        #pragma HLS LOOP_TRIPCOUNT min=c_len max=c_len
        unsigned int chunk_size = BUFFER_SIZE;

        //boundary checks
        if ((i + BUFFER_SIZE) > size)
        {
            chunk_size = size - i;
        }

        read1: for (unsigned int j = 0; j < chunk_size; j++)
        {
            #pragma HLS LOOP_TRIPCOUNT min=c_size max=c_size
            v1_buffer[j] = in1[i + j];
        }

        //Burst read B and calc C, Burst writing to Global memory
        vadd_writeC: for (unsigned int j = 0; j < chunk_size; j++)
        {
            #pragma HLS LOOP_TRIPCOUNT min=c_size max=c_size
            //perform vector addition
            out_r[i+j] = v1_buffer[j] * in2[i+j];
        }
    }
}
```

Using this kernel and the host code the application run-metrics are as shown below:

	VADD			
	No Of Elements	CPU Execution Time (ms)	FPGA Execution Time (ms)	Data Transfer time (on PCIe) (ms)
0	1024	0.002	0.166	21
1	2048	0.003	0.205	22
2	4096	0.004	0.222	22
3	8192	0.007	0.254	21
4	16384	0.015	0.344	21
5	32768	0.033	0.55	22
6	65536	0.066	0.983	22
7	131072	0.126	1.834	24
8	262144	0.304	3.525	25
9	524288	0.492	6.828	26
10	1048576	1.288	13.484	27
11	2097152	2.594	26.814	27
12	4194304	5.086	53.466	30
13	8388608	10.626	106.748	33
14	16777216	20.838	213.362	56



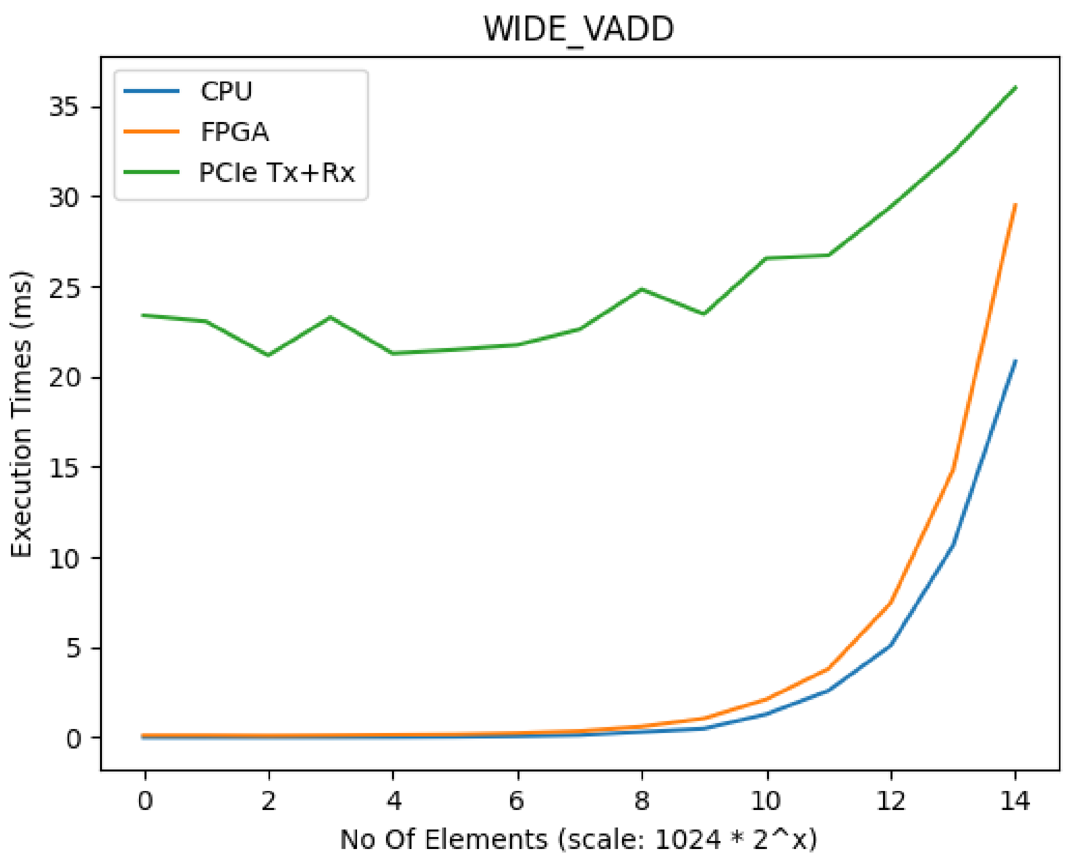
The performance of the kernel in this (VADD) version is almost 10 times slower than the CPU and not up to the mark. Now, we build a better version of VADD by using the full DDR bandwidth of 512bits, by using an "Arbitrary Precision" data type **"typedef ap_unit<512> uint512_t"**. We name it **"WIDE_VADD"**. The Kernel and Host code snippets are as follows.

HOST & FPGA:

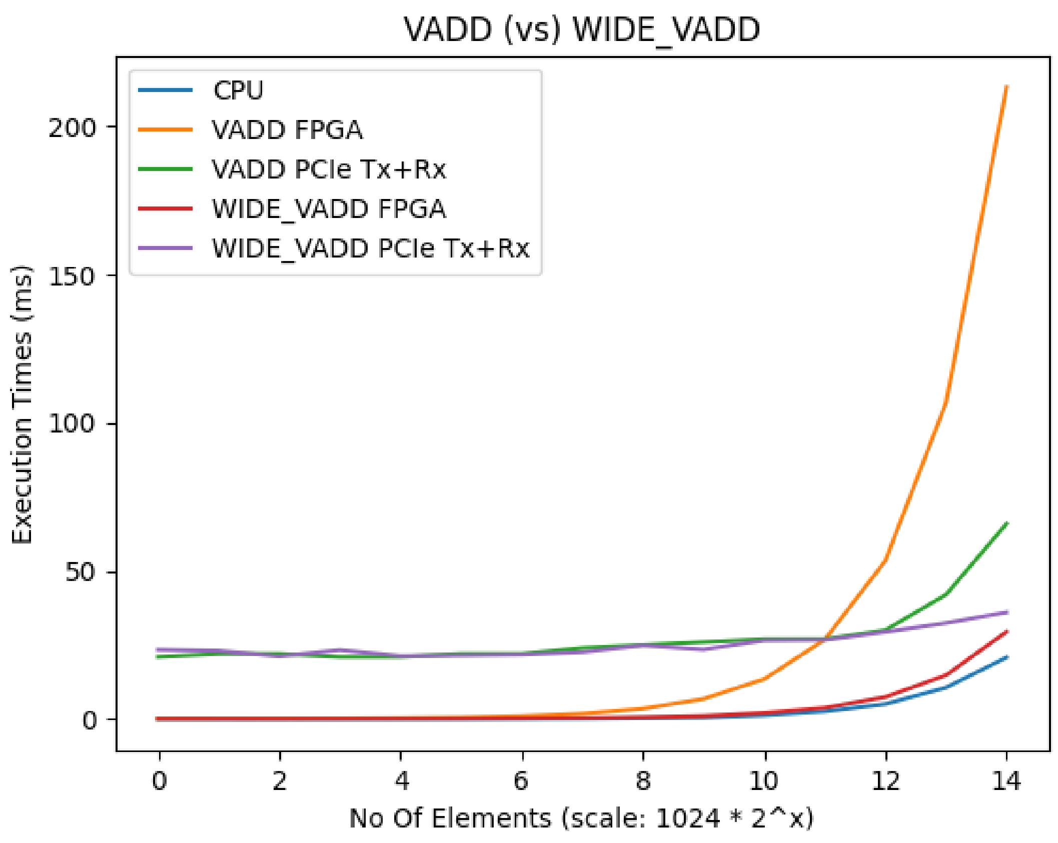
```
void wide_vadd_sw(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t size)
{
    for (uint32_t i = 0; i < size; i++)
    {
        c[i] = a[i] + b[i];
    }
}

void wide_vadd_krnl(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
{
#pragma HLS INTERFACE m_axi port=in1 max_read_burst_length=32 offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=in2 max_read_burst_length=32 offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi port=out max_write_burst_length=32 offset=slave bundle=gmem2
    #pragma HLS INTERFACE s_axilite port = in1 bundle = control
    #pragma HLS INTERFACE s_axilite port = in2 bundle = control
    #pragma HLS INTERFACE s_axilite port = out bundle = control
    #pragma HLS INTERFACE s_axilite port = size bundle = control
    #pragma HLS INTERFACE s_axilite port = return bundle = control

    uint512_dt v1_local[BUFFER_SIZE];
    uint512_dt v2_local[BUFFER_SIZE];
    int size_in16 = ((size - 1) / VECTOR_SIZE) + 1;

    for (int i = 0; i < size_in16; i += BUFFER_SIZE) {
        #pragma HLS DATAFLOW
        #pragma HLS stream variable = v1_local depth = chunk_size
        #pragma HLS stream variable = v2_local depth = chunk_size

        unsigned int chunk_size = ((i + BUFFER_SIZE) > size_in16) ?
                                   (size_in16 - i) : BUFFER_SIZE;
        v1_rd:
        for (int j = 0; j < chunk_size; j++) {
            #pragma HLS PIPELINE
            #pragma HLS LOOP_TRIPCOUNT min = 1 max = chunk_size
            v1_local[j] = in1[i + j];
            v2_local[j] = in2[i + j];
        }
        v2_rd_add:
        for (int j = 0; j < chunk_size; j++) {
            #pragma HLS PIPELINE
            #pragma HLS LOOP_TRIPCOUNT min = 1 max = chunk_size
            uint512_dt tmpV1 = v1_local[j];
            uint512_dt tmpV2 = v2_local[j];
            uint512_dt tmpV3 = 0;

            vec_sum:
            for (unsigned int s = 0; s < DATAWIDTH; s+= 32) {
                #pragma HLS UNROLL
                tmpV3(s + 31, s) = tmpV1(s + 31, s) + tmpV2(s + 31, s);
            }
            out[i + j] = tmpV3;
        }
    }
}
```

The application run-metrics are shown as below:

WIDE_VADD				
	No Of Elements	CPU Execution Time (ms)	FPGA Execution Time (ms)	Data Transfer time (on PCIe) (ms)
0	1024	0.002	0.116	23.394
1	2048	0.003	0.118	23.058
2	4096	0.004	0.101	21.18
3	8192	0.007	0.121	23.281
4	16384	0.015	0.15	21.291
5	32768	0.033	0.175	21.495
6	65536	0.066	0.24	21.754
7	131072	0.126	0.354	22.617
8	262144	0.304	0.603	24.835
9	524288	0.492	1.052	23.469
10	1048576	1.288	2.111	26.555
11	2097152	2.594	3.805	26.723
12	4194304	5.086	7.451	29.413
13	8388608	10.626	14.807	32.407
14	16777216	20.838	29.498	36.006



As we can clearly see there is a significant improvement in the Kernel run times for WIDE_VADD kernel (in comparison with VADD kernel). The plot below shows the comparison between VADD and WIDE_VADD on a single graph.



Ans 3(b) Now we would try to implement the similar kernels for VMUL and WIDE_VMUL based on the previous implementation but for floating point integers.
The host and the kernel code for "VMUL" is as shown below:

HOST:

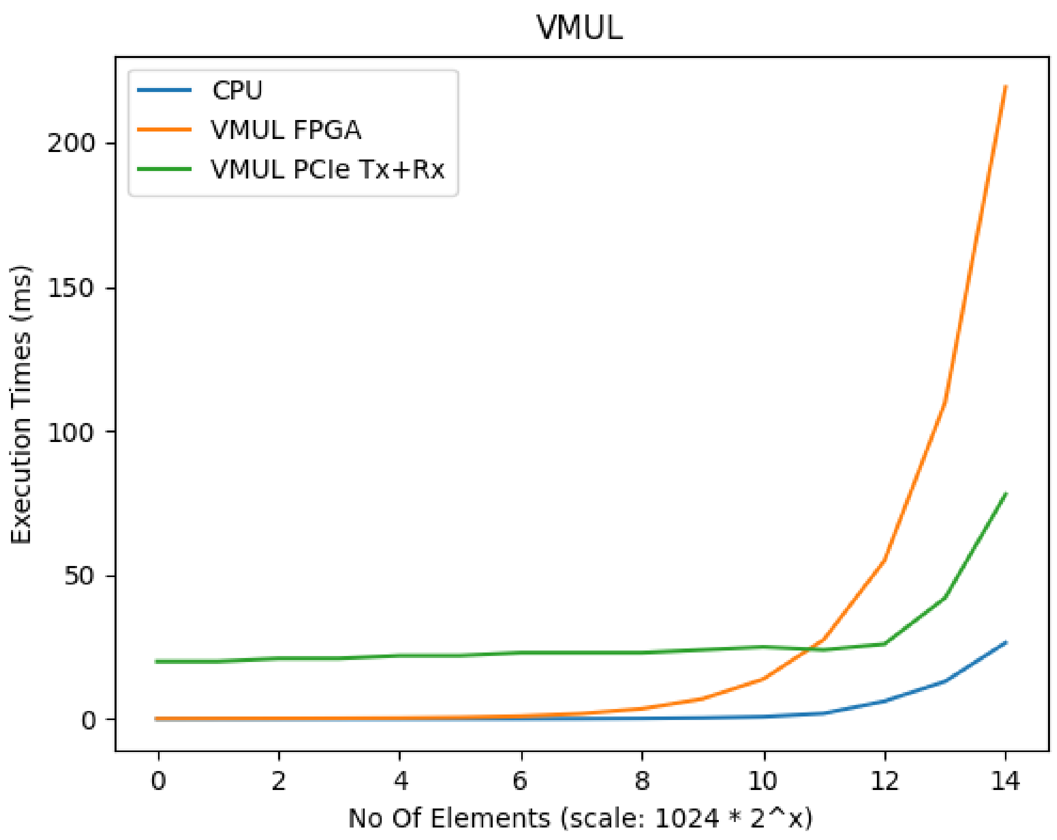
```
void vectors_init(float *buffer_a, float *buffer_b,
                  float *sw_results, float *hw_results,
                  unsigned int num_elements)
{
    std::default_random_engine engine;
    std::normal_distribution<float> dist(0, 50); // range 0 - 50

    for (size_t i = 0; i < num_elements; i++)
    {
        buffer_a[i] = dist(engine) * ((rand() % 2) ? 1 : -1);
        buffer_b[i] = dist(engine) * ((rand() % 2) ? 1 : -1);
        hw_results[i] = 0;
    }
}
void vadd_sw(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t size)
{
    for (uint32_t i = 0; i < size; i++)
    {
        sw_results[i] = buffer_a[i] + buffer_b[i];
    }
}
```

FPGA:

```
void vmul_krnl(float *in1, float *in2, float *out_r, unsigned int size)
{
    float v1_buffer[BUFFER_SIZE];

    for (unsigned int i = 0; i < size; i += BUFFER_SIZE)
    {
        #pragma HLS LOOP_TRIPCOUNT min=c_len max=c_len
        unsigned int chunk_size = BUFFER_SIZE;

        if ((i + BUFFER_SIZE) > size) {
            chunk_size = size - i;
        }

        read1:
        for (unsigned int j = 0; j < chunk_size; j++) {
            #pragma HLS LOOP_TRIPCOUNT min=c_size max=c_size
            v1_buffer[j] = in1[i + j];
        }

        // Burst reading and Burst writing
        vmul_writeC:
        for (unsigned int j = 0; j < chunk_size; j++) {
            #pragma HLS LOOP_TRIPCOUNT min=c_size max=c_size
            //perform vector multiplication
            out_r[i+j] = v1_buffer[j] * in2[i+j];
        }
    }
}
```

The run-metrics of the “VMUL” kernel against multiple input sizes is as shown below:

VMUL				
	No Of Elements	CPU Execution Time (ms)	FPGA Execution Time (ms)	Data Transfer time (on PCIe) (ms)
0	1024	0.001	0.143	20
1	2048	0.002	0.185	20
2	4096	0.004	0.228	21
3	8192	0.006	0.277	21
4	16384	0.012	0.35	22
5	32768	0.026	0.57	22
6	65536	0.048	0.999	23
7	131072	0.094	1.881	23
8	262144	0.186	3.575	23
9	524288	0.395	6.999	24
10	1048576	0.797	13.837	25
11	2097152	1.958	27.55	24
12	4194304	6.19	54.94	26
13	8388608	13.037	109.717	42
14	16777216	26.482	219.268	78



The FPGA execution times of VMUL is similar to the VADD timings. Now lets try to implement the WIDE_VMUL version. As in case of unsigned integers we don't have a similar 512-bit representation for floating point numbers, so we would try to run the kernel in batches and try to improve the data transfer timings. The code snippets for the WIDE_VMUL version are as shown below:

HOST:

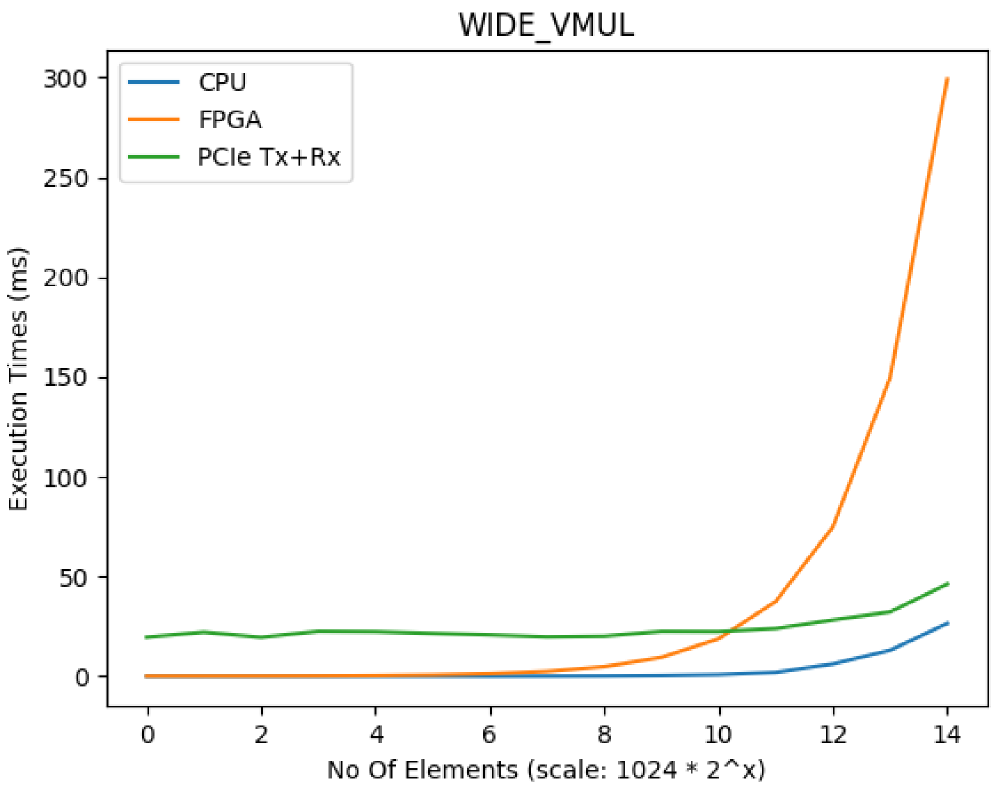
```
void vectors_init(float *buffer_a, float *buffer_b,
                  float *sw_results, float *hw_results,
                  unsigned int num_elements)
{
    std::default_random_engine engine;
    std::normal_distribution<float> dist(0, 50); // range 0 - 50

    for (size_t i = 0; i < num_elements; i++)
    {
        buffer_a[i] = dist(engine) * ((rand() % 2) ? 1 : -1);
        buffer_b[i] = dist(engine) * ((rand() % 2) ? 1 : -1);
        hw_results[i] = 0;
    }
}
void vmul_sw(float *a, float *b, float *c, uint32_t size)
{
    for (uint32_t i = 0; i < size; i++)
    {
        c[i] = a[i] * b[i];
    }
}
```

FPGA:

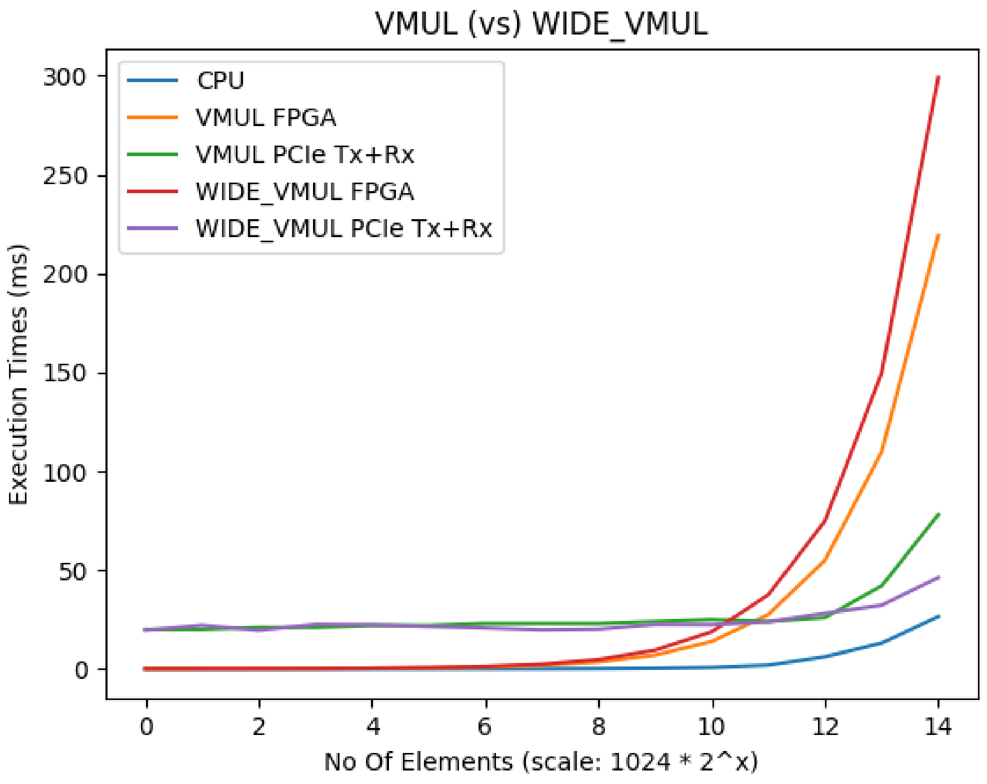
```
void wide_vmul_krnl(const float* in1, const float* in2,
                         float* out, unsigned int size)
{
#pragma HLS INTERFACE m_axi port=in1 max_read_burst_length=32  offset=slave bundle=gmem0
#pragma HLS INTERFACE m_axi port=in2 max_read_burst_length=32  offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi port=out max_write_burst_length=32 offset=slave bundle=gmem2
    #pragma HLS INTERFACE s_axilite port = in1 bundle = control
    #pragma HLS INTERFACE s_axilite port = in2 bundle = control
    #pragma HLS INTERFACE s_axilite port = out bundle = control
    #pragma HLS INTERFACE s_axilite port = size bundle = control
    #pragma HLS INTERFACE s_axilite port = return bundle = control

    float v1_local[BUFFER_SIZE]; // Local memory to store vector1
    float v2_local[BUFFER_SIZE]; // Local memory to store vector2

    for (unsigned int i = 0; i < size; i += BUFFER_SIZE) {
        #pragma HLS DATAFLOW
        #pragma HLS stream variable = v1_local depth = 64
        #pragma HLS stream variable = v2_local depth = 64

        unsigned int chunk_size = ((i + BUFFER_SIZE) > size) ?
                                   (size - i) : BUFFER_SIZE;
        v1_rd:
        for (unsigned int j = 0; j < chunk_size; j++) {
            #pragma HLS PIPELINE
            v1_local[j] = in1[i + j];
            v2_local[j] = in2[i + j];
        }
        v2_rd_mul:
        for (int j = 0; j < chunk_size; j++) {
            #pragma HLS PIPELINE
            out[i + j] = v1_local[j] * v2_local[j];
        }
    }
}
```

The run-metrics for “WIDE_VMUL” are as shown below:

WIDE_VMUL				
	No Of Elements	CPU Execution Time (ms)	FPGA Execution Time (ms)	Data Transfer time (on PCIe) (ms)
0	1024	0.001	0.108	19.604
1	2048	0.002	0.129	22.112
2	4096	0.004	0.146	19.543
3	8192	0.006	0.216	22.528
4	16384	0.012	0.383	22.356
5	32768	0.026	0.691	21.487
6	65536	0.048	1.282	20.726
7	131072	0.094	2.448	19.793
8	262144	0.186	4.807	20.11
9	524288	0.395	9.475	22.48
10	1048576	0.797	18.785	22.383
11	2097152	1.958	37.483	23.887
12	4194304	6.19	74.85	28.207
13	8388608	13.037	149.583	32.224
14	16777216	26.482	299.043	46.206



As we can see that the data transfer timings are improved however there is not much improvement on the side of kernel execution timings. The reason being we didn't have a unit512_t type of abstract precision representation for floating point numbers.



Q4. Repeat the above problem using the Vitis tutorial Example 05 (Vitis_Tutorials_Ex_5) wherein we overlap computation and communication. Also, for CPU vector addition parallelize the vector addition using OpenMP pragma. Compare FPGA performance with the parallelized CPU vector addition (refer Example 06 on the Tutorial).

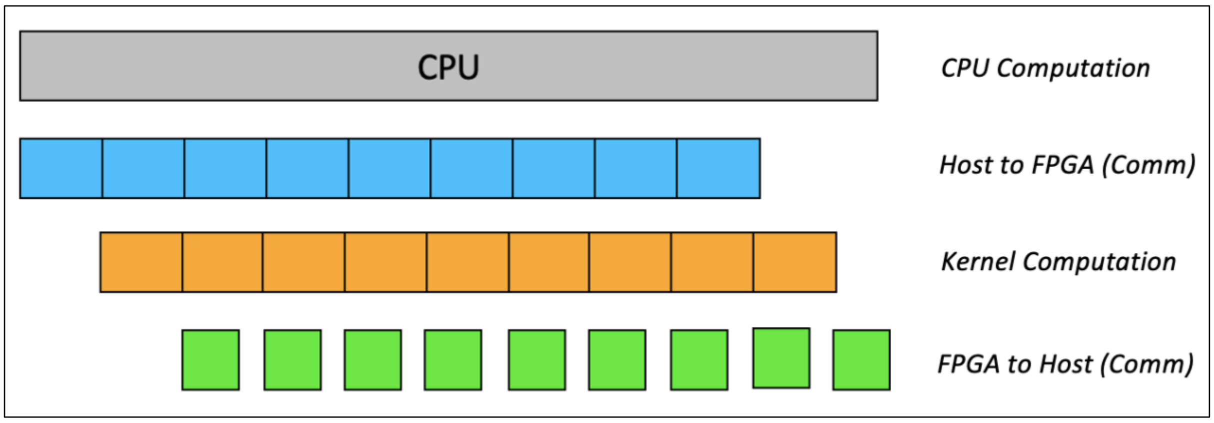
Ans) Recollect the WIDE_VADD results of Q3. We were able to improve the performance and timing of the VADD kernel significantly, however the results were still not at-par with the CPU execution timings. One reason for this is the example that we have considered is a simple naïve addition of two vectors, which would not show much difference in timings between CPU and FPGA, but still, we would try to improve the FPGA timings even better by making some architectural changes in the WIDE_VADD code.

The main problem we faced with the earlier implementations is that the Kernel must wait for the data buffers of both the input vectors to be transferred to the DDR memory bank and only after that, it could start processing the data. Moreover, with the increasing size of the data input buffers the PCIe transfer latency also increased.

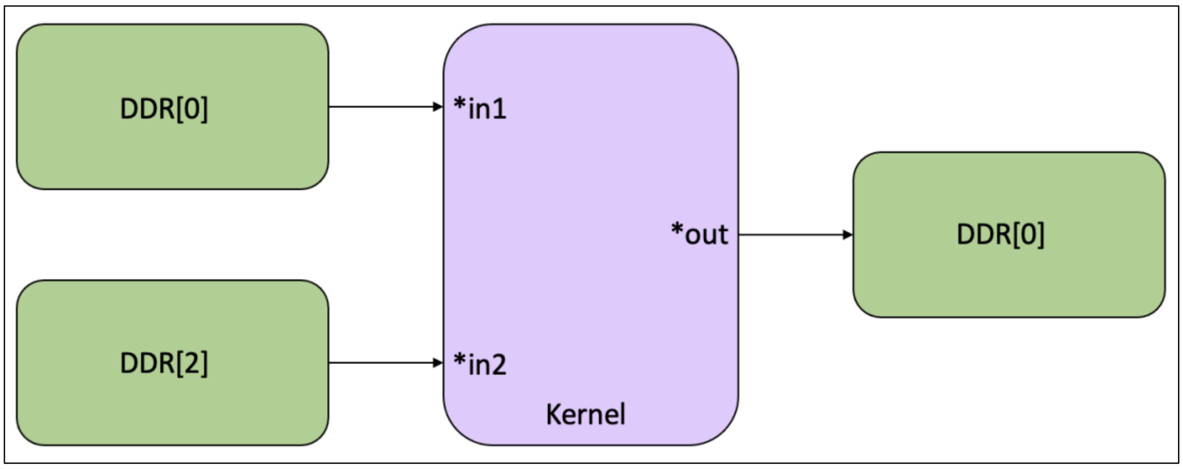
Now, to solve the above problem we would make 2 important architectural changes in the WIDE_VADD implementation, which would significantly improve the performance of the FPGA. They are:

1. Dividing the data buffers of the 2 input vectors and 1 output vector into sub-buffers which are aligned to the page size (4K) of the internal memory.
2. Using multiple DDR memory banks to store the input data vectors in such a manner that the kernel can simultaneously act on the DDR banks (in a ping-pong manner) to process the data and store the output.

By implementing the above-mentioned improvements, we would overlap the data-transfer (communication) with the data processing of the kernel (computation), see fig-1 below.



AWS F1 FPGA card has multiple DDR memory banks and we will be using the DDR[0] and DDR[2] banks to alternatively write input data buffers into them. Kernel would read from these 2 banks and perform the computations and write data back into the DDR[0]. This topology is as shown below:



Due to the above architectural changes, it would be difficult to individually measure the computation and communication timings. So, for this we would use a different metrics where we would combine the compute and communication timings as a single metric and compare with the same on the WIDE_VADD side.

HOST:

Code snippet showing how to map the user-space buffers to specific DDR banks):

```
int main (int argc, char* argv[])
{
...
        // Map our user-allocated buffers as OpenCL buffers
        cl_mem_ext_ptr_t bank0_ext = {0};
        cl_mem_ext_ptr_t bank2_ext = {0};
        bank0_ext.flags = 0 | XCL_MEM_TOPOLOGY;
        bank0_ext.obj   = NULL;
        bank0_ext.param = NULL;
        bank2_ext.flags = 2 | XCL_MEM_TOPOLOGY;
        bank2_ext.obj   = NULL;
        bank2_ext.param = NULL;

        cl::Buffer a_buf(xocl.get_context(),
                         static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                                   CL_MEM_EXT_PTR_XILINX),
                         num_elements * sizeof(uint32_t),
                         &bank0_ext,
                         NULL);
        cl::Buffer b_buf(xocl.get_context(),
                         static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                                   CL_MEM_EXT_PTR_XILINX),
                         num_elements * sizeof(uint32_t),
                         &bank2_ext,
                         NULL);
        cl::Buffer c_buf(xocl.get_context(),
                         static_cast<cl_mem_flags>(CL_MEM_READ_WRITE |
                                                   CL_MEM_EXT_PTR_XILINX),
                         num_elements * sizeof(uint32_t),
                         &bank0_ext,
                         NULL);
...
}
```

In addition, we also improved the CPU execution times by using OpenMP pragma.

```
void vadd_sw(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t size)
{
    #pragma omp parallel for
    for (uint32_t i = 0; i < size; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```

Subdividing the input/output buffers

```
int subdivide_buffer(std::vector<cl::Buffer> &divided_buf,
                     cl::Buffer buf_in,
                     cl_mem_flags flags,
                     unsigned int num_divisions)
{
    size_t size;
    size = buf_in.getInfo<CL_MEM_SIZE>();

    if (size  <= (num_divisions * PAGE_SIZE))
    {
        return FAILURE;
    }

    // Diving the buffer and aligning the sub-buffers to page size
    int num_pages = size / PAGE_SIZE;
    unsigned int num_pages_per_buffer = ((num_pages-1)/10) + 1;
    unsigned int buffer_size = (num_pages_per_buffer * PAGE_SIZE);
    unsigned int num_divs = (size-1)/(num_pages_per_buffer*PAGE_SIZE) + 1;

    cl_buffer_region region;
    int err;
    region.origin = 0;
    region.size   = buffer_size;

    for (unsigned int i = 0; i < num_divs; i++)
    {
       if (i == num_divs-1)
       {
           region.size = size - region.origin;
       }

       cl::Buffer buf = buf_in.createSubBuffer(flags,
                                               CL_BUFFER_CREATE_TYPE_REGION
                                               &region,
                                               &err);
        if (err != CL_SUCCESS)
        {
            return err;
        }

        divided_buf.push_back(buf);
        region.origin += region.size;
    }
    return SUCCESS;
}
```

Enqueue the sub-buffers and kernels into the task-queue:

```
int enqueue_subbuf_vadd(cl::CommandQueue &q,
                        cl::Kernel &krnl,
                        cl::Event &event,
                        cl::Buffer a,
                        cl::Buffer b,
                        cl::Buffer c)
{
    // Get the size of the buffer
    cl::Event k_event, m_event;
    std::vector<cl::Event> krnl_events;

    static std::vector<cl::Event> tx_events, rx_events;
    std::vector<cl::Memory> c_vec;
    size_t size;
    size = a.getInfo<CL_MEM_SIZE>();

    std::vector<cl::Memory> in_vec;
    in_vec.push_back(a);
    in_vec.push_back(b);

    q.enqueueMigrateMemObjects(in_vec, 0, &tx_events, &m_event);
    krnl_events.push_back(m_event);
    tx_events.push_back(m_event);

    if (tx_events.size() > 1)
    {
        tx_events[0] = tx_events[1];
        tx_events.pop_back();
    }

    krnl.setArg(0, a);
    krnl.setArg(1, b);
    krnl.setArg(2, c);
    krnl.setArg(3, (uint32_t)(size / sizeof(uint32_t)));

    q.enqueueTask(krnl, &krnl_events, &k_event);
    krnl_events.push_back(k_event);

    if (rx_events.size() == 1)
    {
        krnl_events.push_back(rx_events[0]);
        rx_events.pop_back();
    }

    c_vec.push_back(c);
    q.enqueueMigrateMemObjects(c_vec,
                               CL_MIGRATE_MEM_OBJECT_HOST,
                               &krnl_events,
                               &event);
    rx_events.push_back(event);

    return 0;
}
```

NOTE: The GitHub code (for subdividing the buffers) is wrong. I have fixed it by properly aligning the sub-buffers to the PAGE_SIZE (4K). In addition, after calling the `subdivide_buffer()` function for all the buffers, we need to reset sub-buffers count based on the actual number of divisions. The reason is, because of the PAGE alignment logic, in some cases the final count of sub-buffers will be less than requested size. Host code is updated to handle it accordingly.

FPGA:

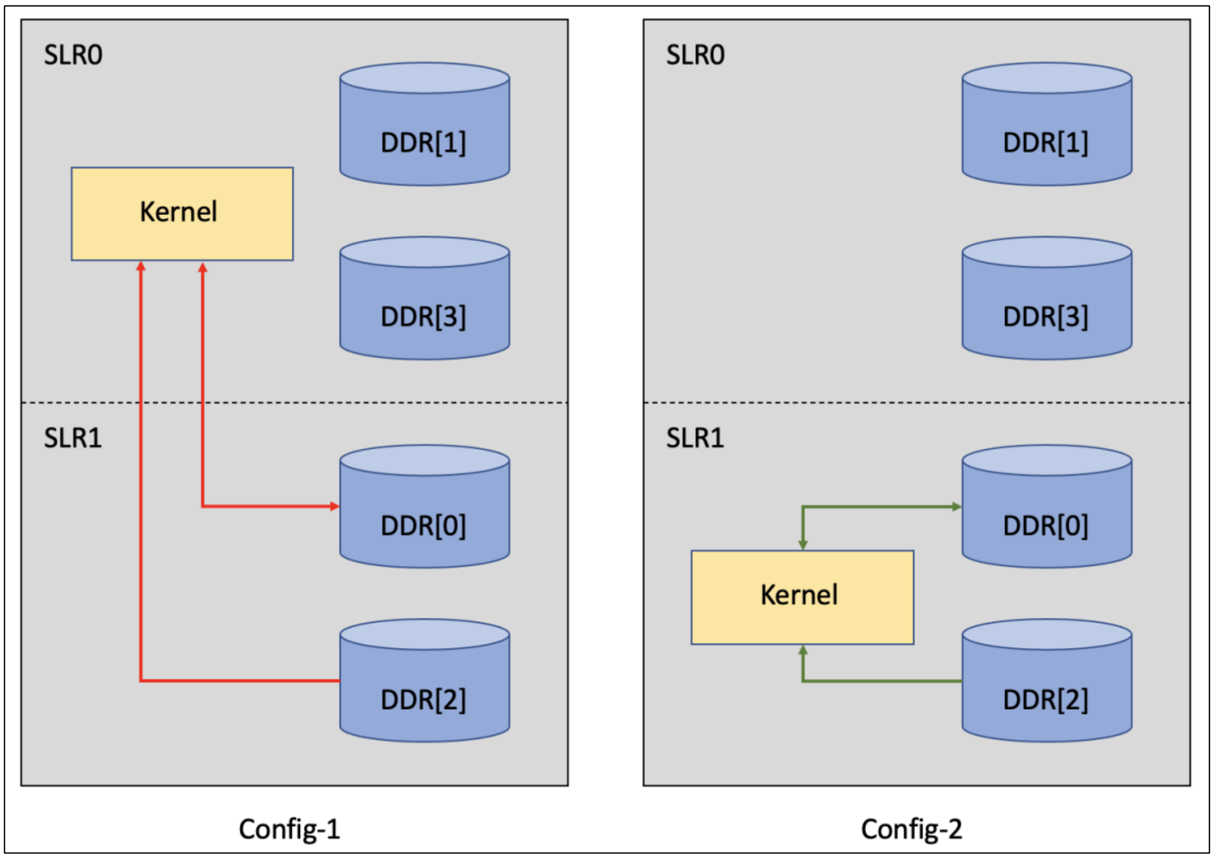
```
void wide_vadd_a1q4_krnl(const uint512_dt *in1,
                         const uint512_dt *in2,
                         uint512_dt *out,
                         int size)
{
#pragma HLS INTERFACE m_axi port=in1 max read burst length=32  offset=slave bundle=gmem0
#pragma HLS INTERFACE m_axi port=in2 max read burst length=32  offset=slave bundle=gmem1
#pragma HLS INTERFACE m_axi port=out max write burst length=32 offset=slave bundle=gmem0
    #pragma HLS INTERFACE s_axilite port = in1 bundle = control
    #pragma HLS INTERFACE s_axilite port = in2 bundle = control
    #pragma HLS INTERFACE s_axilite port = out bundle = control
    #pragma HLS INTERFACE s_axilite port = size bundle = control
    #pragma HLS INTERFACE s_axilite port = return bundle = control

    uint512_dt v1_local[BUFFER_SIZE];
    uint512_dt v2_local[BUFFER_SIZE];

    int size_in16 = ((size - 1) / VECTOR_SIZE) + 1;

    for (int i = 0; i < size_in16; i += BUFFER_SIZE)
    {
        #pragma HLS DATAFLOW
        #pragma HLS stream variable = v1_local depth = chunk_size
        #pragma HLS stream variable = v2_local depth = chunk_size

        unsigned int chunk_size = ((i + BUFFER_SIZE) > size_in16) ?
                                   (size_in16 - i) : BUFFER_SIZE;

        v1_rd:
        for (int j = 0; j < chunk_size; j++)
        {
            #pragma HLS PIPELINE
            #pragma HLS LOOP_TRIPCOUNT min = 1 max = chunk_size
            v1_local[j] = in1[i + j];
            v2_local[j] = in2[i + j];
        }
        v2_rd_add:
        for (int j = 0; j < chunk_size; j++)
        {
            #pragma HLS PIPELINE
            #pragma HLS LOOP_TRIPCOUNT min = 1 max = chunk_size
            uint512_dt tmpV1 = v1_local[j];
            uint512_dt tmpV2 = v2_local[j];
            uint512_dt tmpV3 = 0;

            vec_sum:
            for (unsigned int s = 0; s < DATAWIDTH; s+= 32)
            {
                #pragma HLS UNROLL
                tmpV3(s + 31, s) = tmpV1(s + 31, s) + tmpV2(s + 31, s);
            }
            out[i + j] = tmpV3;
        }
    }
}
```

In addition to the changes made to the Host and the Kernel source code, the following configuration changes are required on the Vitis v++ compiler side to process the port bundling and DDR banks mapping with the ports accordingly. The below lines should be added to the file: `<project_name>_system_hw_link/Hardware/<kernel_name>-link.cfg`

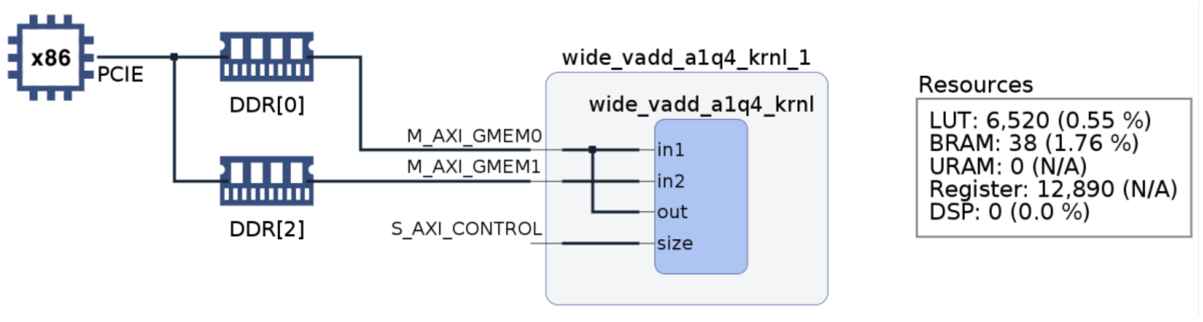
```
[Connectivity]
sp=wide_vadd_a1q4_krnl_1.m_axi_gmem0:DDR[0]
sp=wide_vadd_a1q4_krnl_1.m_axi_gmem2:DDR[2]
sp=wide_vadd_a1q4_krnl_1.m_axi_gmem0:DDR[0]
slr=wide_vadd_a1q4_krnl_1:SLR1
```

We can replace the `m_axi_gmem0` and `m_axi_gmem2` with the actual variable names (`in1, in2 & out`) too, but when in future if we want to change the variable names then we also need to modify these configurations settings too. Mapping the connectivity switch with the `m_axi` ports gives us more flexibility.

The reason behind choosing the DDR banks 0 and 2 is that both these banks belong to the same SLR region of the AWS F1 FPGA. So, assigning these two banks and mapping the kernel onto the same SLR region (last line in the config settings above), would optimize the kernel access timings to the DDR memory (see the figure below).



After creating the HW, Vitis analyzer shows the system design as follows:



The run-metrics of the host and kernel are as shown below:

WIDE_VADD_A1Q4					
	No Of Elements	CPU Execution Time (ms)	CPU Execution Time (OpenMP) (ms)	***WIDE_VADD*** FPGA Execution + Data Trfr time (ms)	***WIDE_VADD_DDR*** FPGA Execution + Data Trfr (ms)
0	1024	0.002	0.222	23.51	0.355
1	2048	0.003	0.245	23.176	0.426
2	4096	0.004	0.247	21.281	0.405
3	8192	0.007	0.241	23.402	0.413
4	16384	0.015	0.249	21.441	1.168
5	32768	0.033	0.249	21.67	1.175
6	65536	0.066	0.278	21.994	1.375
7	131072	0.126	0.318	22.971	1.518
8	262144	0.304	0.412	25.438	1.693
9	524288	0.492	0.586	24.521	2.006
10	1048576	1.288	0.917	28.666	2.225
11	2097152	2.594	1.555	30.528	3.593
12	4194304	5.086	2.911	36.864	6.747
13	8388608	10.626	5.85	47.214	13.107
14	16777216	20.838	11.426	65.504	25.842

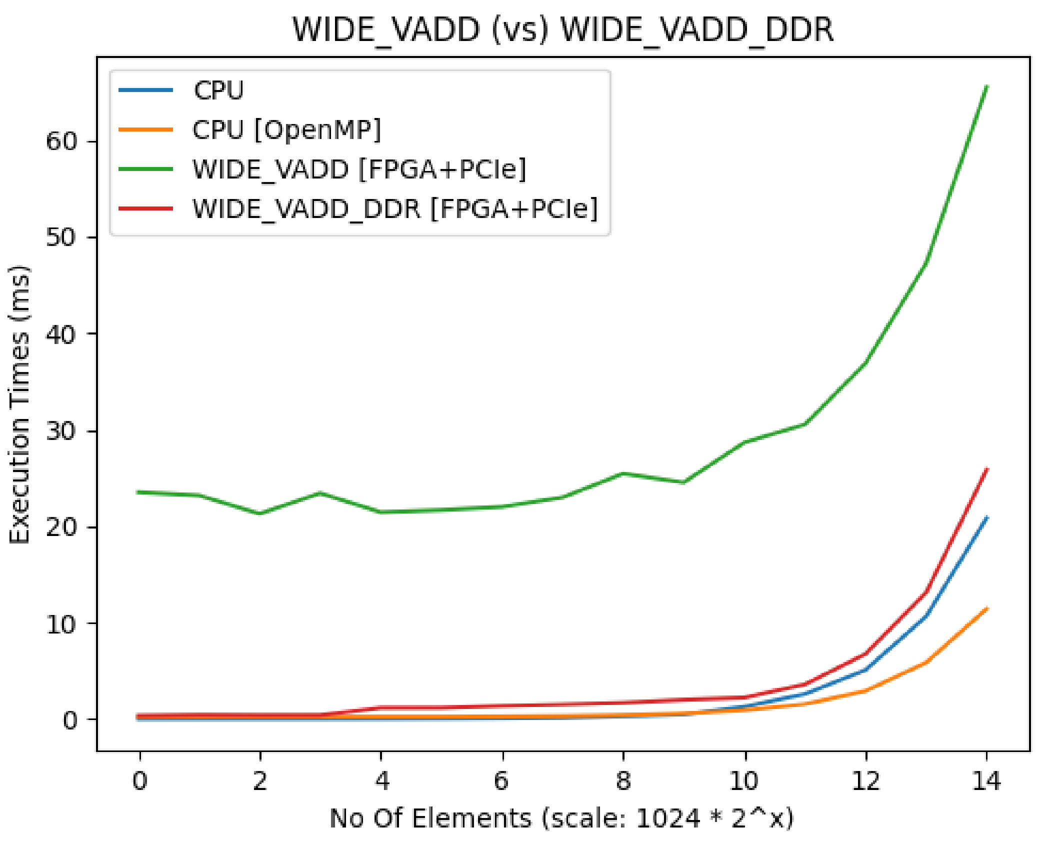
As we can see the [FPGA Execution + Data Transfer] timings of WIDE_VADD_DDR is significantly improved when compared to WIDE_VADD. Moreover, for larger input data the FPGA is performing at-par with the CPU.

NOTE: The [FPGA Execution + Data Transfer] metrics for WIDE_VADD_DDR is collected based on the following computations:

WIDE_VADD_DDR FPGA Execution + Data Trfr (ms)	Subdividing Buffers	Send/Execute/Receive Sub-Buffers	Wait for kernels to complete
0.355	0.004	0.137	0.214
0.426	0.009	0.171	0.246
0.405	0.005	0.145	0.255
0.413	0.005	0.147	0.261
1.168	0.019	0.344	0.805
1.175	0.017	0.531	0.627
1.375	0.021	0.54	0.814
1.518	0.018	1.094	0.406
1.693	0.02	0.742	0.931
2.006	0.021	0.456	1.529
2.225	0.022	0.468	1.735
3.593	0.023	0.533	3.037
6.747	0.025	0.466	6.256
13.107	0.029	0.487	12.591
25.842	0.029	0.573	25.24

[FPGA Execution + Data Transfer]
= (Subdividing Buffers + Send/Execute/Receive Sub-buffers + Wait for Kernels to complete)

Figure below shows the run-metrics of *WIDE_VADD_DDR* against *WIDE_VADD*:



Q5. Assume that you have a database of N vectors of size 256 each. Each element of the vector is a floating-point number. Given a query vector Q, we need to find the vector which is closest to the query vector with respect to the cosine similarity measure. Assume the all the vectors are normalized. Design a system which maximizes the query throughput. You should provide a detailed analysis of latency, throughput, hardware resource utilization etc.

Ans) The first part for this task was to create an appropriate dataset to work with. For that I took the following approach:

Dataset Creation (pre-processing):

- Read the no of vectors (N) as an input argument. Range tested is (2^8, 2^9, 2^{10}, … 2^{22}).
- Each vector is of size 256 floating point numbers.
- Created a data set of N * 256 random floating-point numbers with uniform real distribution.
- Randomly create a query vector (Q) of 256 elements with the same distribution.

HOST : DATASET CREATION:

```
void create_data_set(uint32_t n_vecs, float* q_vec, float* d_vecs)
{
    std::default_random_engine engine;
    std::uniform_real_distribution<float> dist(10, 25); // range 1 - 5
    srand(time(0));

    // create random normalized data vectors
    for (uint32_t i=0; i < n_vecs; i++)
    {
        float norm = 0.0;
        uint32_t offset = i * VECTOR_SIZE;
        for (uint32_t j=0; j < VECTOR_SIZE; j++)
        {
            d_vecs[offset + j] = dist(engine) * ((rand() % 2) ? 1 : -1);
            norm += d_vecs[offset + j] * d_vecs[offset + j];
        }
        // normalize
        for (uint32_t j=0; j < VECTOR_SIZE; j++)
        {
            d_vecs[offset + j] /= sqrt(norm);
        }
    }

    // create a random normalized query vector
    float norm = 0.0;
    for (uint32_t k=0; k<VECTOR_SIZE; k++)
    {
        q_vec[k] = dist(engine) * ((rand() % 2) ? 1 : -1);
        norm += q_vec[k] * q_vec[k];
    }
    // normalize
    for (uint32_t k=0; k<VECTOR_SIZE; k++)
    {
        q_vec[k] /= sqrt(norm);
    }
}
```

Now, we have 3 sets of data buffers – query vector, data vectors & buffer storing cosine values. All these vectors were of different sizes. Idea was to implement the design of **"sub-dividing the buffers and using multiple DDR banks"**, where we would divide only the data vectors and cosine values buffers and send to different DDR banks. The query vector would need not be divided into sub-buffers because it was of constant size (256 elements).

For that design to happen, I had to come up with a different logic of sub-dividing the data vectors buffer and the cosine values buffer. The number of sub-buffers should be same between the 2, but they are of different sizes.

HOST : SUB-DIVIDING DATA VECTORS BUFFER

```
int subdivide_data_buffer(std::vector<cl::Buffer> &divided_buf,
                          cl::Buffer buf_in,
                          cl_mem_flags flags,
                          uint32_t num_divisions,
                          uint32_t& num_vectors_per_buffer)
{
    // Get the size of the buffer
    size_t size = buf_in.getInfo<CL_MEM_SIZE>();
    if (size  <= (num_divisions * PAGE_SIZE))
    {
        return FAILURE;
    }

    uint32_t num_pages = size / PAGE_SIZE;
    uint32_t num_pages_per_buffer = ((num_pages-1)/num_divisions) + 1;
    uint32_t sub_buffer_size = (num_pages_per_buffer * PAGE_SIZE);
    uint32_t num_divs = (size-1) / (sub_buffer_size) + 1;
    num_vectors_per_buffer = sub_buffer_size / (VECTOR_SIZE*sizeof(float));

    cl_buffer_region region;
    region.origin = 0;
    region.size = sub_buffer_size;

    for (uint32_t i = 0; i < num_divs; i++)
    {
        if (i == num_divs-1)
        {
            region.size = size - region.origin;
        }
        int err;
        cl::Buffer buf=buf_in.createSubBuffer(flags,
                                               CL_BUFFER_CREATE_TYPE_REGION,
                                               &region,
                                               &err);
        if (err != CL_SUCCESS)
        {
            return err;
        }
        divided_buf.push_back(buf);
        region.origin += region.size;
    }
    return SUCCESS;
}
```

HOST : SUB-DIVIDING COSINE VALUES BUFFER

```
int subdivide_cosine_buffer(std::vector<cl::Buffer> &divided_buf,
                            cl::Buffer buf_in,
                            cl_mem_flags flags,
                            uint32_t num_buffers,
                            uint32_t num_vectors_per_buffer)
{
    if (num_buffers == 0)
    {
        return FAILURE;
    }

    // Get the size of the buffer
    size_t size = buf_in.getInfo<CL_MEM_SIZE>();
    size_t element_size = sizeof(float); // 4 bytes
    uint32_t sub_buffer_size = (num_vectors_per_buffer * element_size);

    cl_buffer_region region;
    region.origin = 0;
    region.size = sub_buffer_size;

    for (uint32_t i = 0; i < num_buffers; i++)
    {
        if (i == num_buffers-1)
        {
            region.size = size - region.origin;
        }
        int err;
        cl::Buffer buf=buf_in.createSubBuffer(flags,
                                              CL_BUFFER_CREATE_TYPE_REGION,
                                              &region,
                                              &err);
        if (err != CL_SUCCESS)
        {
            return err;
        }
        divided_buf.push_back(buf);
        region.origin += region.size;
    }
    return SUCCESS;
}
```

`subdivide_data_buffer()` will be called before `subdivide_cosine_buffer()`. In that call we would make a note of number of vectors per each sub-buffer and use that data in the subsequent call to `subdivide_cosine_buffer()`. That way we would maintain proper synchronization between the two buffers and have same no of divisions.

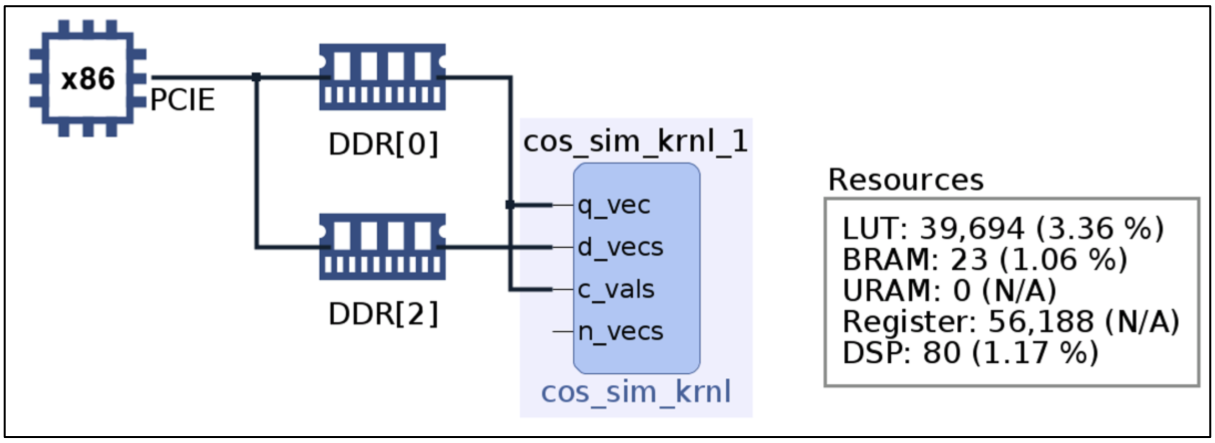
Moreover we have mapped the query vector and the cosine vector to the same DDR memory bank (DDR[0]), because (during kernel computations) we would read the query vector only once and after that we would only access that memory bank to write cosine values – hence there will be no contention. We will map the data vector to a different DDR bank (DDR[2]), but in the same SLR region (SLR1). Finally map the kernel also to the SLR1 region for fast computations.

The mapping done in the host code is as show below:

HOST : DDR MEMORY MAPPING

```
// Map our user-allocated buffers as OpenCL buffers
cl_mem_ext_ptr_t bank0_ext = {0};
cl_mem_ext_ptr_t bank2_ext = {0};
bank0_ext.flags = 0 | XCL_MEM_TOPOLOGY;
bank0_ext.obj   = NULL;
bank0_ext.param = NULL;
bank2_ext.flags = 2 | XCL_MEM_TOPOLOGY;
bank2_ext.obj   = NULL;
bank2_ext.param = NULL;

cl::Buffer q_vec_buf(xocl.get_context(),
                     static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                               CL_MEM_EXT_PTR_XILINX),
                     VECTOR_SIZE * sizeof(float),
                     &bank0_ext,
                     NULL);
cl::Buffer d_vec_buf(xocl.get_context(),
                     static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                               CL_MEM_EXT_PTR_XILINX),
                     num_elements * sizeof(float),
                     &bank2_ext,
                     NULL);
cl::Buffer c_val_buf(xocl.get_context(),
                     static_cast<cl_mem_flags>(CL_MEM_READ_WRITE |
                                               CL_MEM_EXT_PTR_XILINX),
                     num_vectors * sizeof(float),
                     &bank0_ext,
                     NULL);
```

The associated architectural design created in the HW is as shown below:



LUT: 39,694 (3.36 %)
BRAM: 23 (1.06 %)
URAM: 0 (N/A)
Register: 56,188 (N/A)
DSP: 80 (1.17 %)

Now that all the pre-processing steps are done, we need to enqueue the data transfer tasks (for the query vector and the data vector sub-buffers), kernel computations task and the cosine data collection tasks in an overlapping manner so that the kernel starts computations immediately once it sees the first data sub-buffer. The code for this is shown below:

HOST: ENQUEUE DATA TRANSFER & COMPUTATION TASKS

```
int enqueue_subbuf_vadd(cl::CommandQueue &q,
                        cl::Kernel &krnl,
                        cl::Event &event,
                        cl::Buffer q_buf,
                        cl::Buffer d_buf,
                        cl::Buffer c_buf)
{
    // Get the size of the buffer
    cl::Event k_event, m_event;
    std::vector<cl::Event> krnl_events;
    static std::vector<cl::Event> tx_events, rx_events;

    std::vector<cl::Memory> in_vec;
    in_vec.push_back(q_buf);
    in_vec.push_back(d_buf);

    q.enqueueMigrateMemObjects(in_vec, 0, &tx_events, &m_event);
    krnl_events.push_back(m_event);
    tx_events.push_back(m_event);

    if (tx_events.size() > 1)
    {
        tx_events[0] = tx_events[1];
        tx_events.pop_back();
    }

    size_t size;
    size = c_buf.getInfo<CL_MEM_SIZE>();
    uint32_t n_vectors = size / sizeof(float);
    krnl.setArg(0, q_buf);
    krnl.setArg(1, d_buf);
    krnl.setArg(2, c_buf);
    krnl.setArg(3, n_vectors);

    q.enqueueTask(krnl, &krnl_events, &k_event);
    krnl_events.push_back(k_event);

    if (rx_events.size() == 1)
    {
        krnl_events.push_back(rx_events[0]);
        rx_events.pop_back();
    }

    std::vector<cl::Memory> c_vec;
    c_vec.push_back(c_buf);
    q.enqueueMigrateMemObjects(c_vec,
                               CL_MIGRATE_MEM_OBJECT_HOST,
                               &krnl_events,
                               &event);
    rx_events.push_back(event);

    return 0;
}
```

Now comes the kernel side of the computations. Here, we read the query vector as a burst read operation (only once) from the DDR[0] bank, and save it in the local FPGA memory. I have used **HLS UNROLL** pragma to speed up the operations.

For computing the dot-product I used HLS PIPELINE pragma to pipeline the MAC operations (Multiply and Accumulate) operations.

FPGA: COMPUTING COSINE SIMILARITY

```
void cos_sim_krnl(float* q_vec,         // Query Vector : size = 256
                  float* d_vecs,        // Data Vectors : size = batch-size
                  float* c_vals,        // store cosine values
                  unsigned int n_vecs)  // number of vectors
{
   #pragma HLS INTERFACE m_axi port=q_vec max_read_burst_length=32 offset=slave bundle=gmem0
   #pragma HLS INTERFACE m_axi port=d_vecs max_read_burst_length=32 offset=slave bundle=gmem2
   #pragma HLS INTERFACE m_axi port=c_vals max_write_burst_length=32 offset=slave bundle=gmem(
    #pragma HLS INTERFACE s_axilite port = q_vec bundle = control
    #pragma HLS INTERFACE s_axilite port = d_vecs bundle = control
    #pragma HLS INTERFACE s_axilite port = c_vals bundle = control
    #pragma HLS INTERFACE s_axilite port = n_vecs bundle = control
    #pragma HLS INTERFACE s_axilite port = return bundle = control

    float query_buffer[VECTOR_SIZE];

    // Burst reading the query vector into local memory
    // we need to do this only once
    read_query_vector:
    for (unsigned int j = 0; j < VECTOR_SIZE; j++)
    {
        #pragma HLS UNROLL
        query_buffer[j] = q_vec[j];
    }

    //Per iteration of this loop perform VECTOR_SIZE vector MAC operations
    for (unsigned int i = 0; i < n_vecs; i++)
    {
        #pragma HLS PIPELINE
        float dot_product = 0.0;

        compute_dot_product:
        for (unsigned int j = 0; j < VECTOR_SIZE; j++)
        {
            #pragma HLS PIPELINE
            dot_product += query_buffer[j] * d_vecs[i*VECTOR_SIZE +j];
        }
        c_vals[i] = dot_product;
    }
}
```

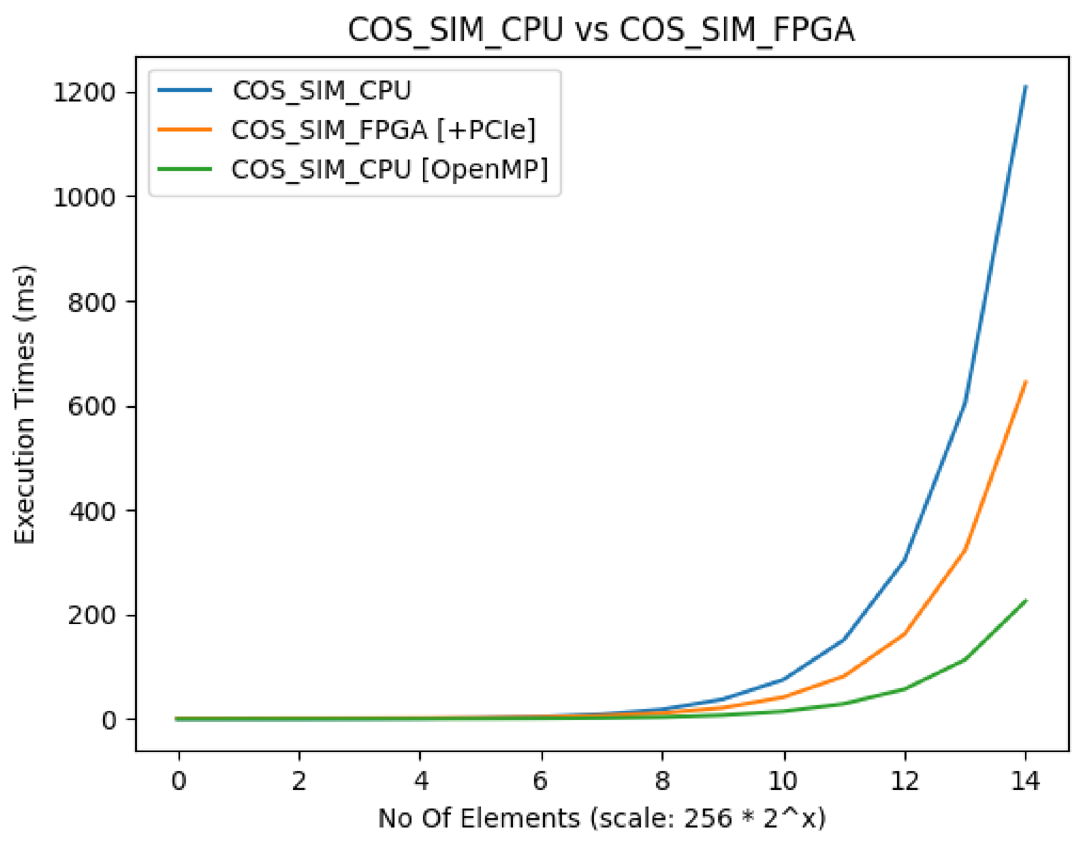
With this implementation I was able to outperform the traditional CPU based computations for computing the dot product of floating-point vectors. However, the OpenMP version of the CPU based computations performs better than FPGA.

The results of FPGA computations against CPU based computations (for both the versions with and without OpenMP) are as shown in the statistics and plot below:

COS_SIM			
No Of Vectors (Vector Size = 256)	CPU Execution Time (ms)	CPU Execution Time (OpenMP) (ms)	FPGA Execution + Data Trfr (ms)
256	0.071	0.229	1.449
512	0.143	0.279	1.461
1024	0.279	0.295	1.638
2048	0.58	0.374	1.704
4096	1.113	0.467	2.014
8192	2.237	0.65	2.607
16384	4.47	1.101	3.799
32768	9.129	2.049	6.393
65536	18.432	3.844	11.448
131072	37.828	7.533	21.595
262144	75.35	14.768	41.767
524288	151.513	29.239	81.985
1048576	303.707	57.088	162.485
2097152	603.885	113.469	322.873
4194304	1208.934	225.995	644.157

[FPGA Execution + Data Transfer]
= (Subdividing Buffers + Send/Execute/Receive Sub-buffers + Wait for Kernels to complete)

FPGA Execution + Data Trfr (ms)	Subdividing Buffers	Send/Execute/Receive Sub-Buffers	Wait for kernels to complete
1.449	0.014	0.479	0.956
1.461	0.016	0.43	1.015
1.638	0.016	0.44	1.182
1.704	0.016	0.47	1.218
2.014	0.016	0.688	1.31
2.607	0.019	0.405	2.183
3.799	0.016	0.399	3.384
6.393	0.022	0.475	5.896
11.448	0.024	0.474	10.95
21.595	0.024	0.476	21.095
41.767	0.024	0.57	41.173
81.985	0.023	0.538	81.424
162.485	0.023	0.57	161.892
322.873	0.02	0.538	322.315
644.157	0.017	0.509	643.631

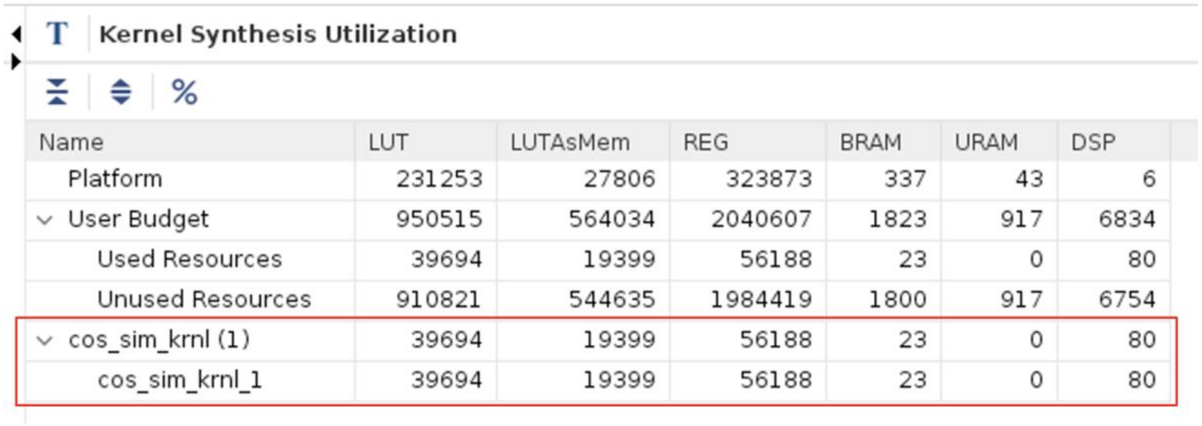


A typical execution run for this task is as shown below: (with N = 2^{22} = 4194304)

```
[ec2-user@ip-172-31-88-240 Hardware]$
[ec2-user@ip-172-31-88-240 Hardware]$ ./cos_sim cos_sim.awsxclbin 4194304
[INFO] No Of Vectors: 4194304
[INFO] No Of Elements: 1073741824
[INFO] Loading cos_sim.awsxclbin to program the board.
[INFO] Cosine Similarity experiment completed successfully!
[INFO] Max cosine value [SW]: [3354417] : 0.307137
[INFO] Max cosine value [HW]: [3354417] : 0.307137
--------------- Key execution times ---------------
[ET] Create Data Set                            : 29516.469 ms
[ET] Compute Cosine Similarity on CPU           : 1208.934 ms
[ET] Compute Cosine Similarity on CPU [OpenMP]  :  225.995 ms
[ET] Subdividing Buffers                        :    0.017 ms
[ET] Send/Execute/Receive sub buffers           :    0.509 ms
[ET] Wait for kernels to complete               :  643.631 ms
[ec2-user@ip-172-31-88-240 Hardware]$
[ec2-user@ip-172-31-88-240 Hardware]$
```

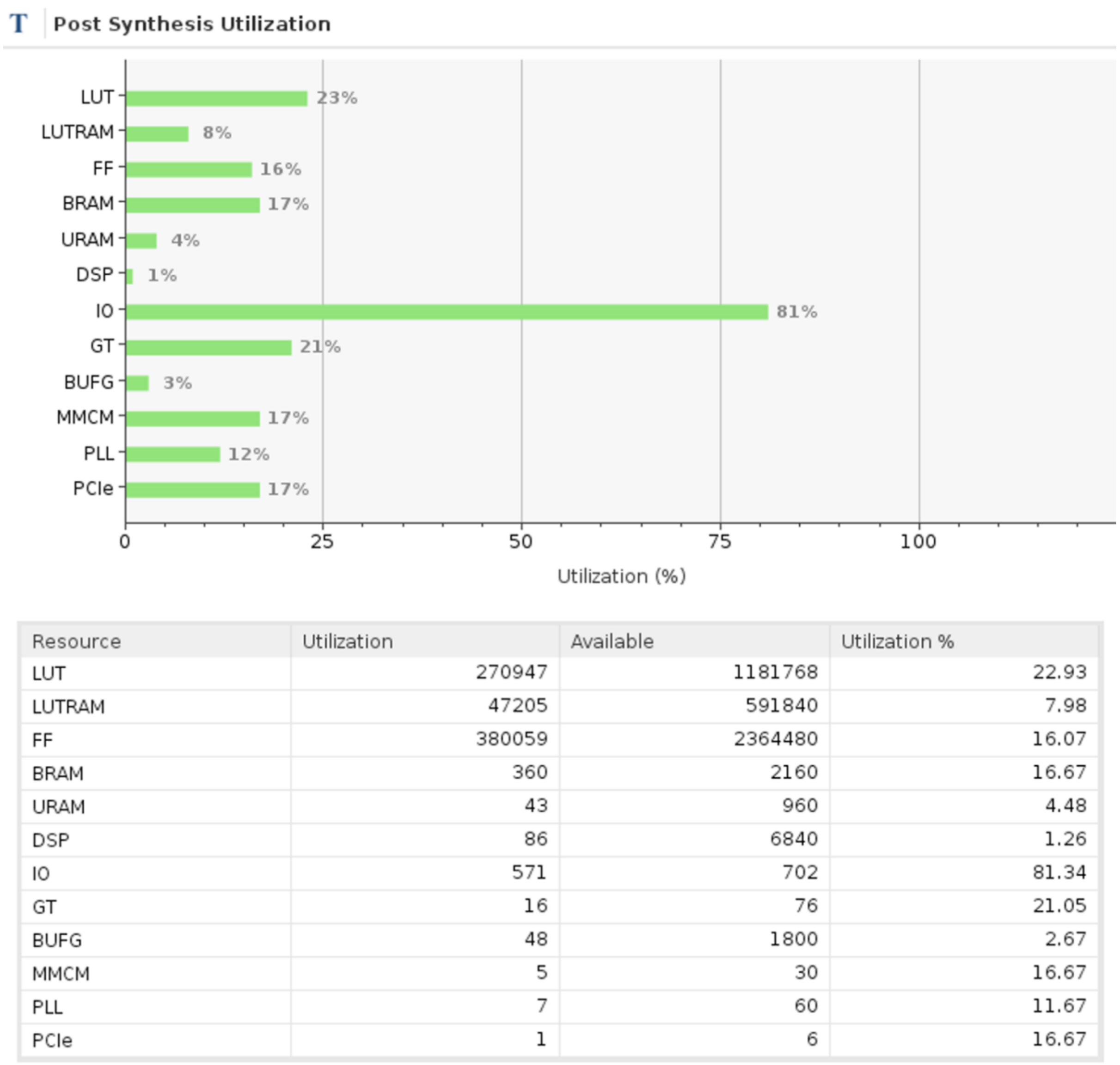
The Hardware utilization numbers are as shown below:

Accelerator utilization (Kernel Synthesis):


Kernel Synthesis Utilization

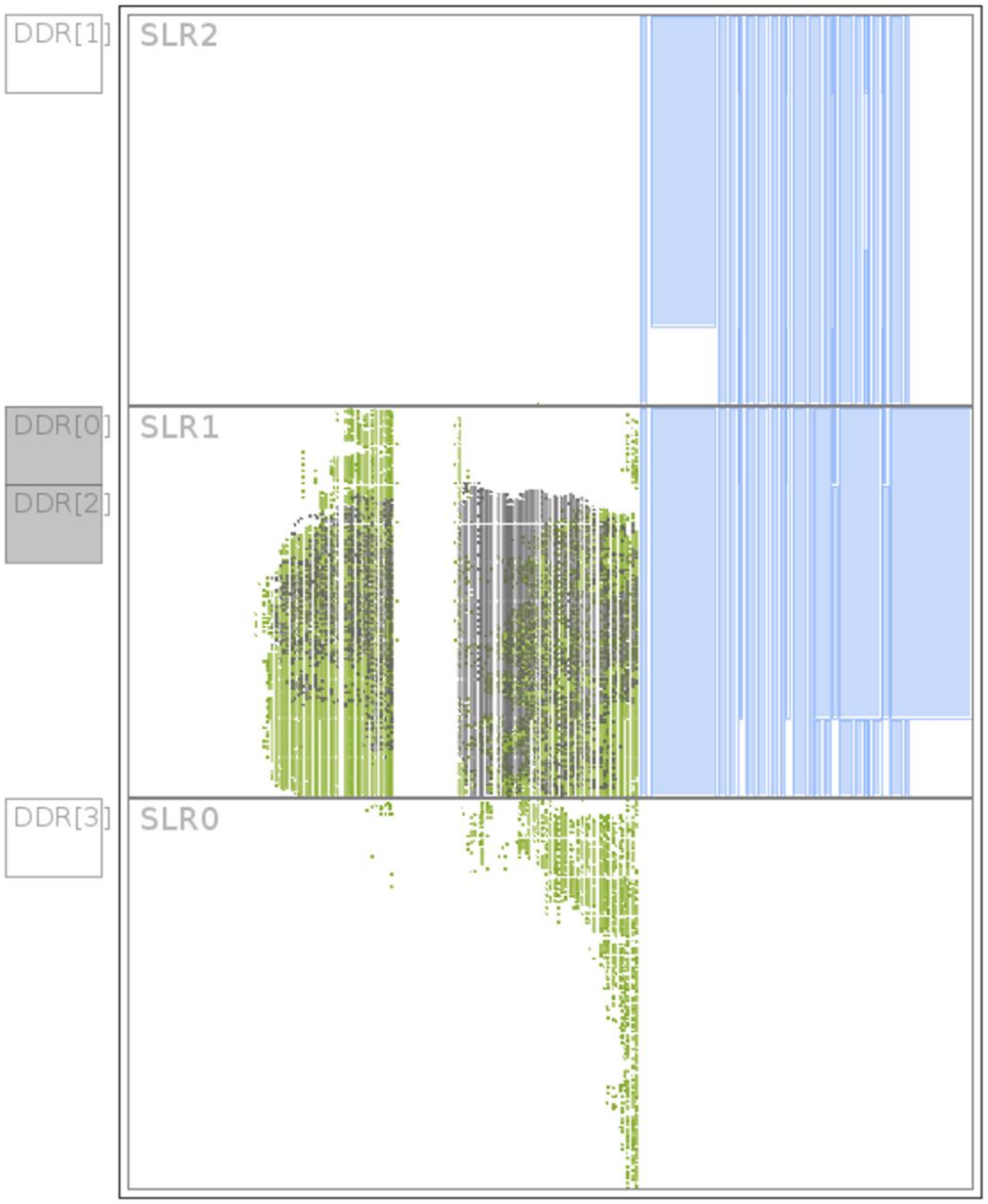
Name	LUT	LUTAsMem	REG	BRAM	URAM	DSP
Platform	231253	27806	323873	337	43	6
User Budget	950515	564034	2040607	1823	917	6834
Used Resources	39694	19399	56188	23	0	80
Unused Resources	910821	544635	1984419	1800	917	6754
cos_sim_krnl (1)	39694	19399	56188	23	0	80
cos_sim_krnl_1	39694	19399	56188	23	0	80

Device Utilization (Post Synthesis):


Post Synthesis Utilization

Resource	Utilization	Available	Utilization %
LUT	270947	1181768	22.93
LUTRAM	47205	591840	7.98
FF	380059	2364480	16.07
BRAM	360	2160	16.67
URAM	43	960	4.48
DSP	86	6840	1.26
IO	571	702	81.34
GT	16	76	21.05
BUFG	48	1800	2.67
MMCM	5	30	16.67
PLL	7	60	11.67
PCIe	1	6	16.67

Device Map (on FPGA):



In conclusion the cosine similarity operation is memory bound as we can see there are lot of resources on the FPGA that are unused, and the computations mostly depend on the memory bandwidth & data transfer rates. The detailed Roofline Analysis is given below:

Roofline Analysis:

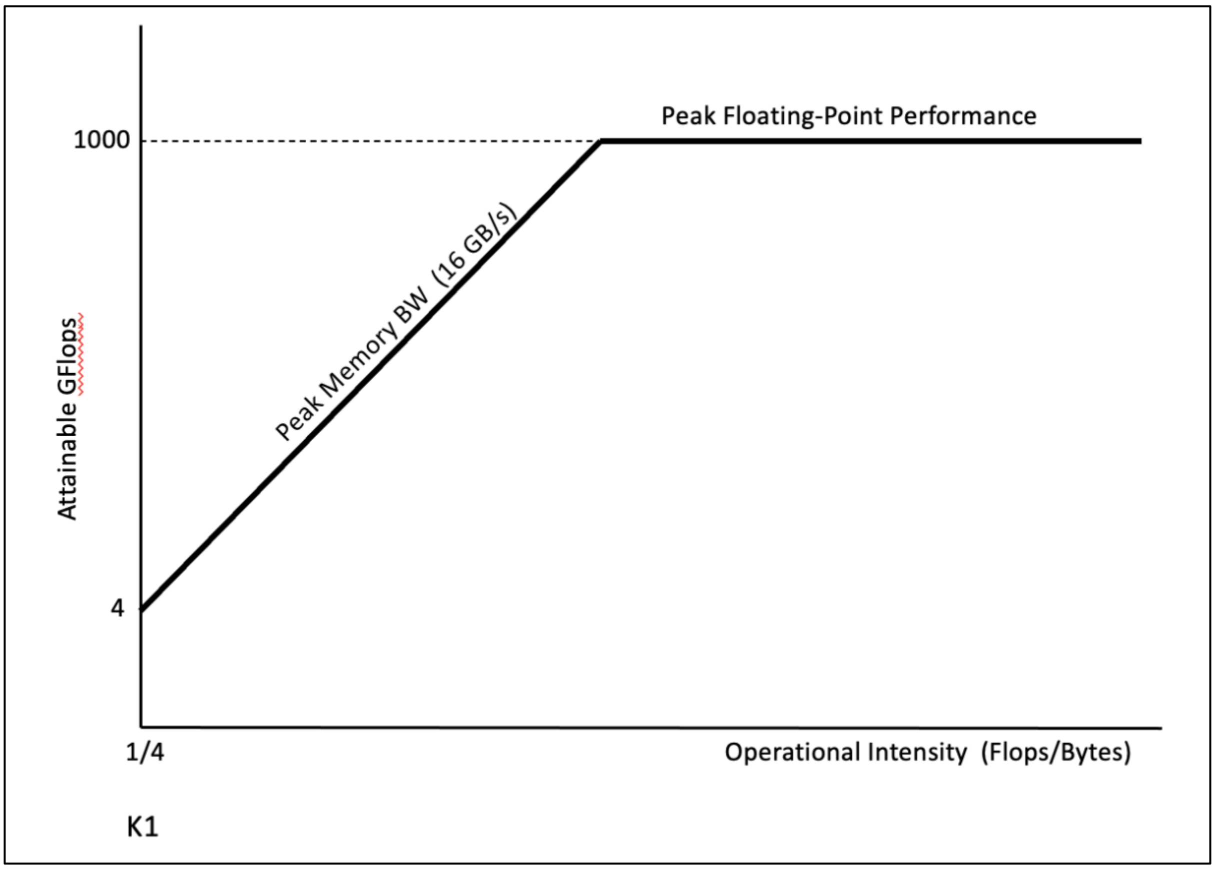
Now let's do some Roofline analysis on the results.
For this we would consider the number of DSPs for floating point computations (MAC operations). So, in the AWS F1 instance there are 6800 DSPs, but using the 100% of them is not possible as the place and route would never happen with such utilization. Based on several test runs performed on different kernels we observe that one can use around 4000 DSPs for the maximum performance. Hence we have **4000 DSPs** for the peak usage and **250 MHz** of operating frequency of AWS F1 instance – based on the reference here.

Thus, the **Peak floating-point performance** = 4000 * 250 * 10^6 = **1000 GFlops**.
The DDR4 memory bandwidth = **16 GB/s**
The **Operational Intensity** = (No of Operations / No of Bytes Involved)
= (2 / 8) = **1/4**.

NOTE: Here no of operations is 1 addition and 1 multiplication per every 2*4 = 8 bytes of data.

Based on the above data the **Attainable GFlops** = (Operational Intensity) * (DDR B/W)
= (1/4 Flops/Byte) * (16 GB/s)
= **4 GFlops**

Hence the basic version of the Kernel (say version ***K1***) has a very poor performance of 4 GFlops, in comparison with the maximum attainable GFlops of 1000 GFlops.

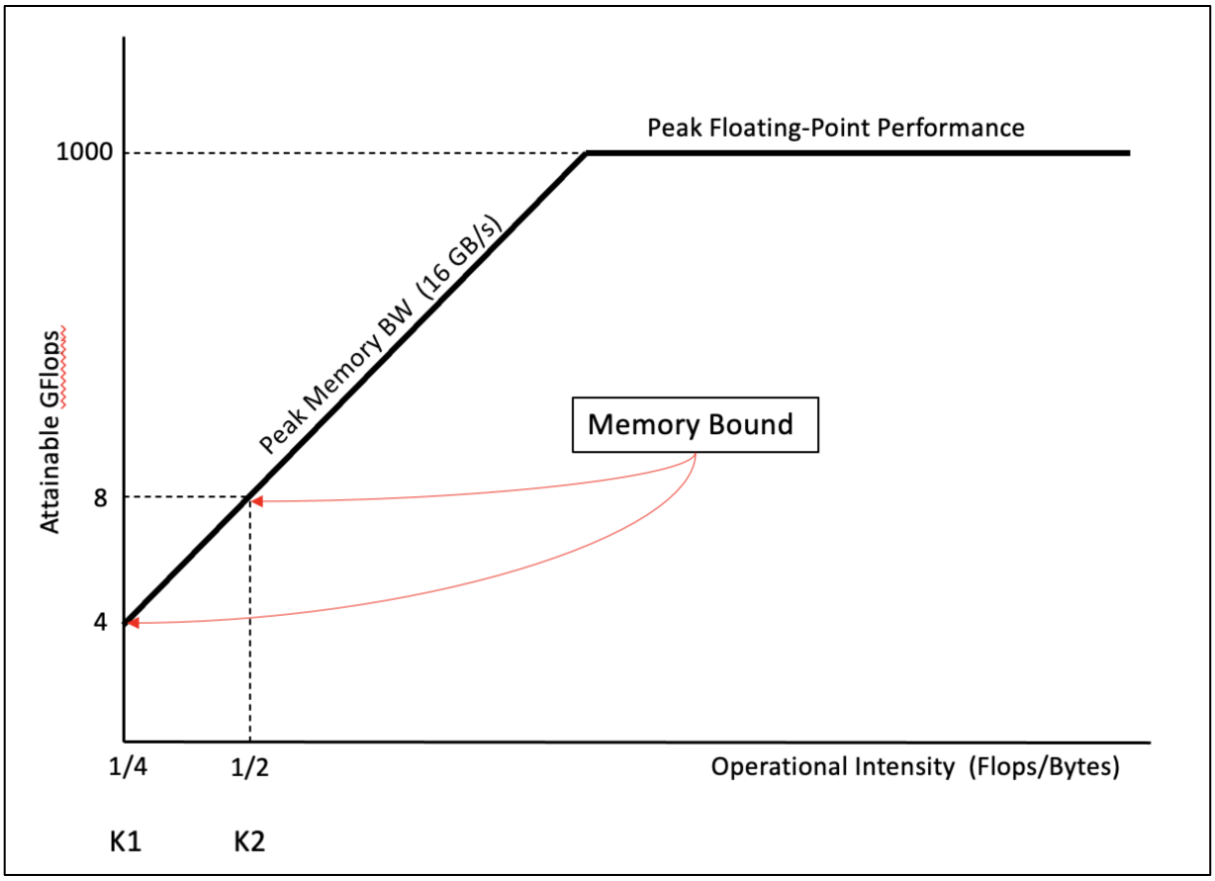


Now how to Improve the compute performance?

Now consider copying the query vector into FPGA local memory before the MAC computations. As the vector size is constant, we can ignore the copy time, and we only need to do it once.

Now the new **Operational Intensity** = (No of Operations / No of Bytes Involved)
= (2 / 4)
= **1/2**

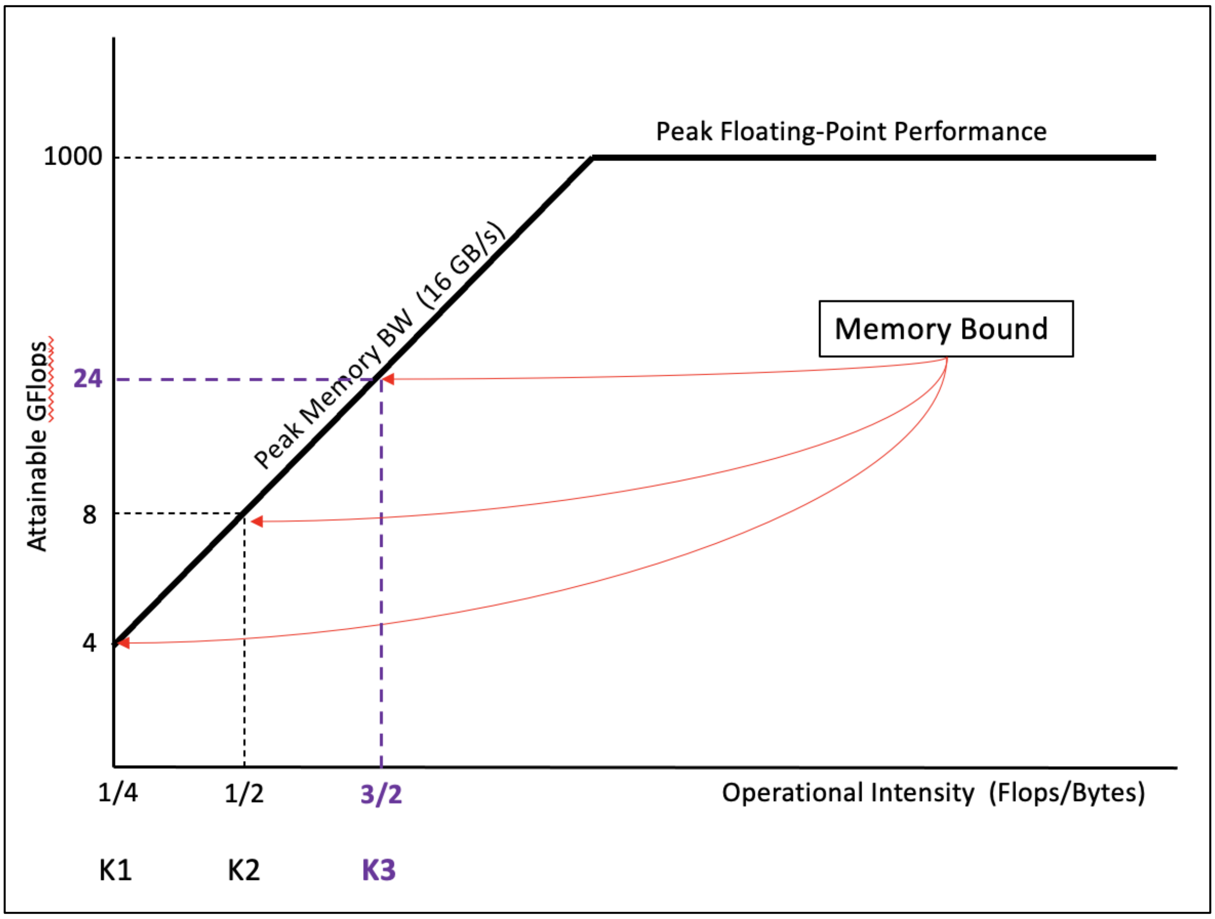
Hence with this version (K2) of kernel, we attain a performance of = 16 * 1/2 = **8 GFlops.**
A slight improvement over version K1. But still very far away from Peak performance.



Both the version K1 and K2 are memory bound. Now let’s try to improve the performance further.
Now considering the version K3 with the following 2 improvements:

- Using double-buffering technique (ping-pong buffers), using 2 DDR banks, From SLR1 region.
- Sub-dividing the input buffers to perform batch-wise computations.

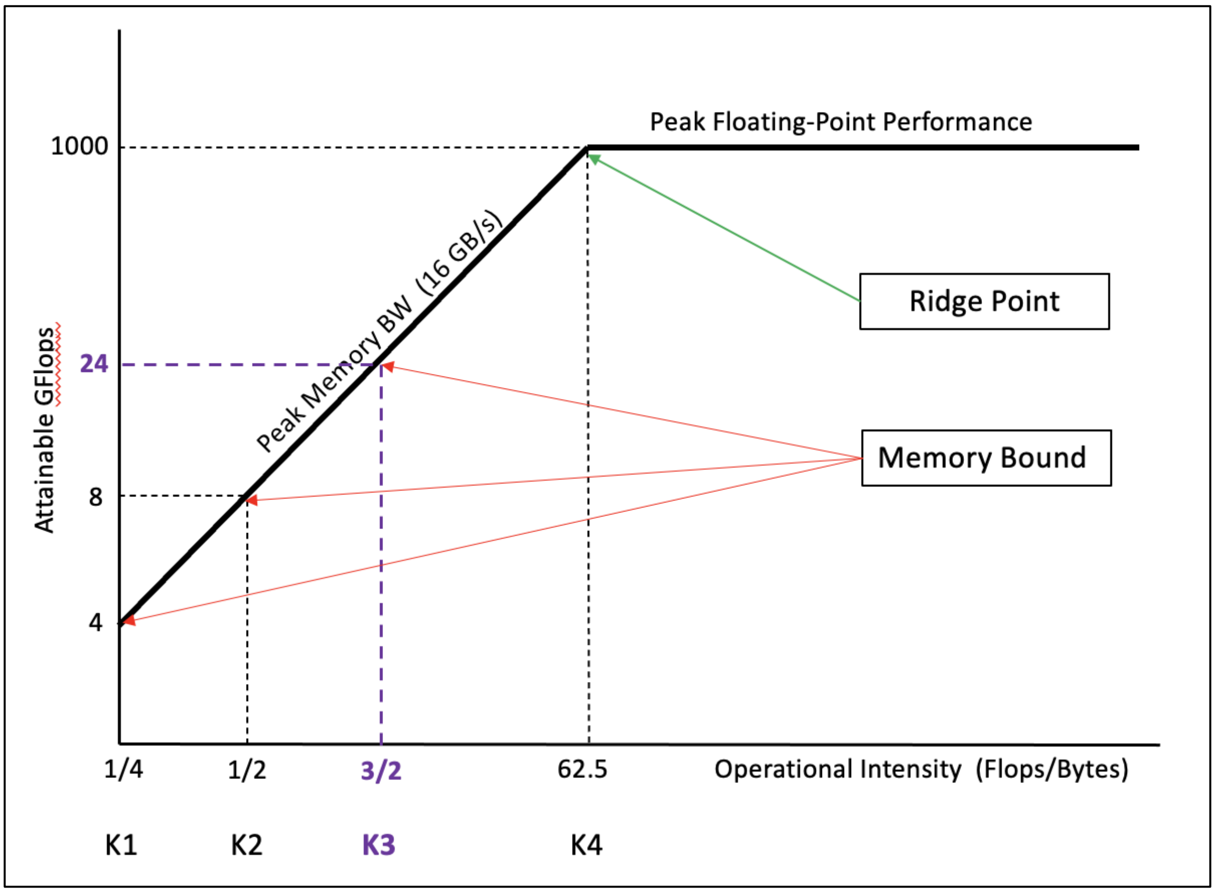
With the above 2 improvements (by keeping the **number of batches** of input data buffers to be fixed at **10**), I was able to use around **80 DSPs** for the Floating-Point computations. Now, we need to compute the performance using a different metric.
With 80 DSPs used for floating point computations we have, the Compute Performance as
= (80) * (300 * 10^6) = **24 GFlops**



The Roofline performance of the current implementation version of the application is highlighted in purple. Still the application is Memory Bound.

Now to attain the peak performance of 1000 GFlops, we need an operating intensity of 62.5 (1000 / 16 = 125/2 = 62.5). So, if we somehow increase the Operational Intensity to 62.5, we need to increase the number of operations. One way to achieve that is to increase the number of queries (let say Q queries) and perform the MAC operations for these Q queries in parallel – this will give another version of Kernel (say K4).

Then we get the Operational Intensity as Q * (2/4) = Q/2
Thus, to attain max performance we have Q/2 = 62.5 = 125/2 => **Q = 125**
Hence if we have substantial amount of input queries to batch 125 queries and perform computations for them in parallel then there is a possibility to attain the maximum peak Floating Point performance as shown below:


Peak Floating-Point Performance
1000
Ridge Point
Peak Memory BW (16 GB/s)
24
Memory Bound
8
4
Attainable GFlops
1/4
1/2
3/2
62.5
Operational Intensity (Flops/Bytes)
K1
K2
K3
K4