**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,** whichwe can get from Vitis, as shown below:

Table

Description automatically generated

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.**

1. **For increasing vector sizes (N = 210, 211, 212, …), 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).**
2. **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:**

|  |
| --- |
|  |

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

Table

Description automatically generated

Chart, line chart

Description automatically generated

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:**

|  |
| --- |
|  |

The application run-metrics are shown as below:

Table

Description automatically generated

Chart, line chart

Description automatically generated

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.

Chart, line chart

Description automatically generated

**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:**

|  |
| --- |
|  |

**FPGA:**

|  |
| --- |
|  |

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

Table

Description automatically generated

Chart, line chart

Description automatically generated

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:**

|  |
| --- |
|  |

**FPGA:**

|  |
| --- |
|  |

The run-metrics for “WIDE\_VMUL” are as shown below:

Table

Description automatically generated

Chart, line chart

Description automatically generated

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.

A picture containing chart

Description automatically generated

**Q4. Repeat the above problem using the Vitis tutorial Example 05 (**[**Vitis\_Tutorials\_Ex\_5**](https://github.com/%20Xilinx/Vitis-Tutorials/blob/2021.1/Hardware%20Acceleration/Introduction/)**) 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.

Chart, bar chart

Description automatically generated

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:

Diagram

Description automatically generated

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)**:**

|  |
| --- |
|  |

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

|  |
| --- |
|  |

Subdividing the input/output buffers

|  |
| --- |
|  |

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

|  |
| --- |
|  |

**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:**

|  |
| --- |
|  |

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).

Diagram

Description automatically generated

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

Diagram

Description automatically generated

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

Table

Description automatically generated

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:

Table

Description automatically generated

[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***:

Chart, line chart

Description automatically generated

**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 (28, 29, 210, … 222).
* 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:**

|  |
| --- |
|  |

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**

|  |
| --- |
|  |

**HOST : SUB-DIVIDING COSINE VALUES BUFFER**

|  |
| --- |
|  |

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**

|  |
| --- |
|  |

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

Diagram

Description automatically generated

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**

|  |
| --- |
|  |

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**

|  |
| --- |
|  |

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:

Table

Description automatically generated

[FPGA Execution + Data Transfer]

= (Subdividing Buffers + Send/Execute/Receive Sub-buffers + Wait for Kernels to complete)

Table

Description automatically generated

Chart

Description automatically generated

A typical execution run for this task is as shown below: (with **N = 222 = 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):**

Table

Description automatically generated

**Device Utilization (Post Synthesis):**

A picture containing chart

Description automatically generated

**Device Map (on FPGA):**

Chart, treemap chart

Description automatically generated

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](https://github.com/aws/aws-fpga/blob/master/Vitis/docs/Alveo_to_AWS_F1_Migration.md).

Thus, the **Peak floating-point performance** = 4000 \* 250 \* 106 = **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.

Diagram

Description automatically generated

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.

A picture containing chart

Description automatically generated

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 \* 106) = **24 GFlops**

Chart, line chart

Description automatically generated

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:

Chart, line chart

Description automatically generated