**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