Zynq SoC Based High Speed Data Transfer Using PCIe: A Device Driver Based Approach

Pramod Kumar Tanwar[1], Om Prakash Thakur[1], Kritik Bhimani[2], Gaurav Purohit[1], Vipin Kumar[1], Sanjay Singh[1], Kota Solomon Raju[1]
[1]Cyber Physical Systems, CSIR-CEERI, Pilani, India
[2]BITS-Pilani, Goa Campus, Goa, India

Idaku Ishii, Sushil Raut
Department of System Cybernetics
Hiroshima University
Hiroshima, Japan

***Abstract*— High Speed Data Transfers is a typical requirement of data intensive applications like image and video processing. Speed efficiency can be ensured by handling the data transfers at both Hardware and Software level. A complete system has been developed by implementing the hardware architecture on FPGA and writing corresponding Software device driver to perform the speedy data transfers from endpoint to a root complex device using PCIe interface. This paper describes the approach to design and verify this system. The speed of data transfer achieved practically for PCIe (2.0) x4 is 4Gib/s. The developed hardware architecture is resource constrained and low power. The hardware is implemented on Xilinx Zynq device. This work has a good potential in the field of image and video processing and can be used to perform large data transfer operations at high speed.**

Keywords*— *PCIe, Zynq, XMD, High Speed, DMA

I. Introduction

Image and video processing has revolutionized the world with its high performance digital cameras having flexible interface to achieve high throughput. The camera captured images are put to hardware accelerators to perform filtering, processing and displaying operations. There are huge number of computations performed on the captured images using FPGA boards. FPGA devices are used to create reconfigurable hardware architectures to perform the above said operations at high speed. After applying the image and video processing algorithms on the captured images, these images are sent to Central Processing Unit (CPU) for displaying and further processing. In this paper, a hardware architecture is developed in Xilinx Vivado using IPs to send the processed images data from the FPGA board to CPU with very low latency. Peripheral Component Interconnect Express (PCIe) is used here as the high speed serial interface to create the communication link between FPGA board and the CPU. To create and manage the PCIe interface, a software device driver is written on Linux Ubuntu OS which performs the data transfer operations smoothly and speedily [1].

For a typical high speed vision requirement, frames are captured at more than 250 frames/sec, pre-processed and sent to CPU for further processing and usage. A high speed frame grabber which captures image at more than 250 frames/sec with resolution of (512*512 pixels), needs a high speed (Gib/s) interface through which the data could be sent to the host system for further processing. In order to fulfill the high throughput need of high-speed digital data processing and to achieve high-speed communication between digital front-ends and computer, PCIe is the best suitable interface now a days [2]. This interface has a number of versions and lanes which could be used as per the application requirement. In this paper, PCIe (2.0) x4 is used to analyze the capability and efficiency of this interface. The developed hardware architecture is having Zynq SoC, which has abundant logic cells and dual hard core ARM Cortex A9 processors to make the complex decisions based on the arrived data from the peripherals like high speed camera. The Zynq SoC is also having the Programmable Logic (FPGA) and is capable in processing large data, applying massively parallel algorithms and performing speedy computations. Fig. 1 describes the interaction between PCIe endpoint and root complex CPU. Xilinx ZC706 board having Zynq-7000 all programmable SoC (Z-7045) is used as a PCIe endpoint.

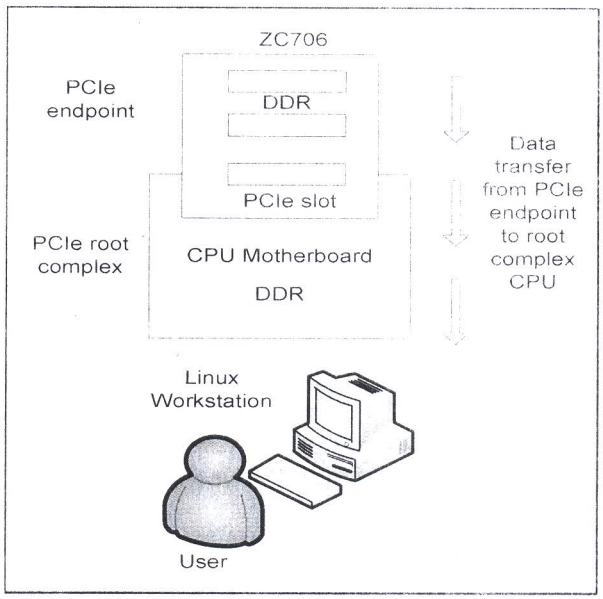

Fig. 1 Introduction to endpoint initiated data transfer through PCIe

ZC706 has PCIe (2.0) x4 interface, 1 GB DDR3 for processing systems (PS) and another 1 GB for Programmable Logic (PL). The hardware design is made using reconfigurable

hardware available in the PL part of this board. The developed hardware creates an interface between the endpoint and root complex CPU. To manage the data transfer operation, a software device driver is written on Linux core which could be run in online mode. The implemented software driver can work on all the Zynq SoC boards by modifying the configurations [3].

In the remaining sections of this paper, hardware design and software driver development are discussed. The developed hardware and software are put on the FPGA board and CPU, it performs the required data transfer operation and verifies the data buffers.

II. HARDWARE DESIGN DEVELOPMENT

Hardware design is realized using Xilinx Vivado tool and implemented on Xilinx Zynq-7000 SoC. The main IPs which are used in this project are: 1. AXI CDMA, 2. AXI memory mapped to PCI express, 3. Zynq-7000 processing system, 4. AXI BRAM generator, 5. Block memory generator. Other small IPs like binary counter, AXI interconnect, processing system reset, clock buffers and other glue logic IPs are also used in this design. The goal is to transfer data packets from ZC706 board DDR3 memory to Intel-i7 CPU using PCIe bus. PCIe is more like a network where different endpoints are connected to a switch or bridge and the switch or bridge is connected to root complex using dedicated paths. In the presented work, endpoint is directly connected to the root complex device. The endpoint which initiates the transaction is called requester and the responder is called completer. Here, the ZC706 Board is configured as endpoint and the CPU board is functional as root complex. When the PCIe device is connected to CPU then the configuration address spaces are filled. The PCIe device has three PCIEBARs (base address register) but only one is required here, which is used by the CPU to communicate with the ZC706 FPGA board. The source and destination addresses are of different bits (32 bits in ZC706 DDR and 64 bits in Intel-i7 CPU address space). It is necessary to use an address translator *i.e.* AXI memory mapped to PCI express block. To store the translation vectors, one Block Memory (BRAM) is needed and the controller of this memory is AXI BRAM controller [4]. Vivado 2015.1 is used to create the Zynq based hardware architecture.

Zynq processing system is required to execute the instructions. DDR controller sends the data buffers from Zynq DDR to CPU DDR. CDMA block manages data transfer and CPU usage. The remaining blocks like Processor system reset, counter, AXI Interconnect *etc.* are supporting IP blocks which are used to generate clocks, resets and interconnections between IP blocks. Some logical gates are also used to bring down the frequency of reference signals to act as debug monitors to check the proper working of the developed design on the hardware board. An implemented design has a low visibility to be looked into for debugging. A designer has to connect a JTAG debugger and uses Chipscope or other similar In Circuit Debugger (ICD) to dwell into the FPGA design once it is configured. We have used an old-school technique of LEDs is used to check the clocks and basic handshaking. Some user LEDs on the board are attached to show the clock and link up. Link up is the basic step towards the handshaking of PCIe interface. The block diagram in Fig. 2 shows the main IPs used to perform data transfer operations speedily. The Zynq-7000 processing system, AXI CDMA and AXI PCIe IPs configures each other. The Block memory generator IP is accessed by Zynq, AXI PCIe and AXI CDMA IPs to get the translated addresses during transfer operations. The DDR is interfaced with Zynq-7000 and data are accessed from it through its slave ports. The accessed data are put on the PCIe link after encoding and parallel to serial conversion. The PCIe link transfers the data on the CPU side and after serial to parallel conversion and decoding, the data buffers are received and displayed on the CPU.

Data is transferred in the form of standard packets. A data transfer packet consists of overheads which decreases the efficiency of transfer but increases the reliability. The data transfer packet looks like as shown in Fig. 3 [5]. PCIe protocol ensures the transfer of data payload from one device to other reliably.

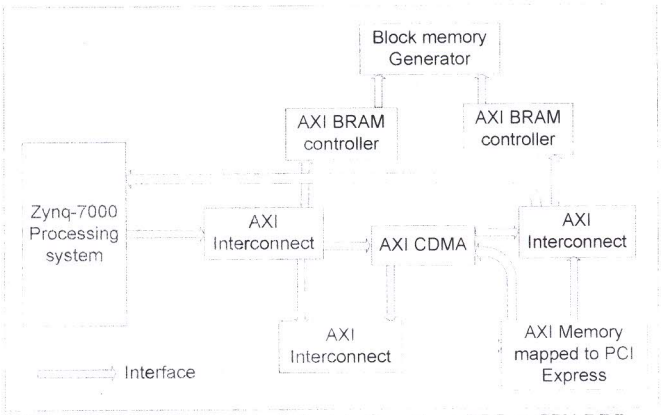

Fig. 2 Hardware design to transfer data from Zynq DDR to CPU DDR

For large data transfers, the whole data is divided into packets and sent in sequence, the error cyclic redundancy check (ECRC) and linear CRC are appended to the packet in transaction and data link layers. In the physical layer, start and stop bits are added and then 8/10b encoding is done. Data is sent bit by bit after conversion from parallel to serial [6].

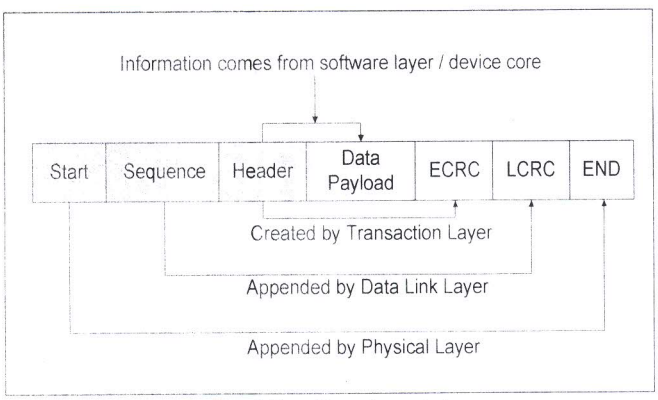

Fig. 3 Data Transfer packet

This hardware architectural design is then synthesized and bitstream file is generated, which is dumped to the ZC706 board. This design is now ready to create the detection link and initializing the interfaces which are necessary for data transfers. CPU should be able to detect the ZC706 board. The detection of ZC706 board indicates that this is ready to perform the data transfer operation. User LEDs as discussed above in this section indicates the clock and PCIe Link establishment with the CPU. To perform the required transfer, a software device driver is developed and inserted with Linux Ubuntu OS as explained in the next section.

III. SOFTWARE DEVICE DRIVER DEVELOPMENT

Device driver is a software program file which runs on a specific hardware target. It acts as an interface between firmware and host operating system. As we go from Industry Standard Architecture (ISA) bus to PCIe bus, the complexity of software program increases. PCIe driver is loosely based on PCI driver approach only. It makes use of the same library and header file structure as used in PCI. Only the addresses and Transaction Layer Packet (TLP) size are different than PCI. Writing device drivers is more like copying and applying the functions based on our application. But, it's not trivial. So, the main focus should be on understanding the header files and functions and knowing how to use and where to use these. In computers, a lot of processes run concurrently and do different tasks. Each process asks for different resources like computing power, memory, network connectivity, registers and other resources. The kernel is the fundamental and most important part of OS which manages all the requests made by different tasks. The kernel's role can be split into these: Process management, Memory management, Filesystems management, Device controls and Networking [7]. When any system call is occurred then the calls are sent to specific area to get managed. The kernel subsystems manage every call which are made by user space application programs. When the hardware devices are attached to peripheral ports then it gets detected by using device drivers and its configuration addresses and mount addresses are updated to specific proc files. Hardware is accessed using features like Virtual File Systems (VFS), Talk To You (TTYs). When the specific system calls are made, the kernel subsystems access the hardware devices and perform the desired tasks.

Linux OS is chosen to write the PCIe device driver, because of its open source nature and availability of community support, which is very helpful for achieving such an objective. Another advantage is that the device driver modules can be added to the kernel during runtime in Linux systems, which is not a case in MS Windows. In windows, the system needs to be rebooted after adding any driver modules. The driver modules are added and removed by using *insmod* and *rmmod* commands respectively. A char driver is written which can perform the burst mode Direct Memory Access (DMA) transfer in scatter gather mapping. The driver is written for kernel space and user space separately.

A. Kernel Space Driver Writing

Kernel space driver is OS specific. It is created to make a suitable environment for the user space applications to be executed. It sets file attributes and manages file system operations. In case of PCIe kernel space driver, it initializes, probes and removes PCIe device. There are a number of header files available to be used to write driver according to our task requirements in */usr/include* and */usr/include/linux* section of the OS files. The basic header files for char drivers are: *kernel.h*, *module.h* and *init.h*. Another header file *pci.h* is added for PCIe char driver. Except these, *kobject.h* is used to build the hierarchy seen in */sys*, *interrupt.h* to request and handle interrupts, *string.h* to change string to integer, *device.h* to use device id and many more to allocate character device regions. *device.h* also enables and disables pci. For file operations and signal attributes, *fs.h* and *signal.h* files are used. There is no *main()* function in kernel space. The functions are called only when any device is interfaced. The kernel space driver do the following tasks: 1. Gives major and minor number, 2. Sets file system attributes, 3. Manages device file systems opening, closing and memory mapping, 4. PCIe device probing, 5. Handles interrupts, 6. Sets DMA masking and coherent mapping, *etc.* This kernel space driver file is compiled with Linux kernel and it produces *.ko* (kernel object) file. Kernel object file is inserted first as a driver module which creates an environment for user space programs to execute. The user space driver file makes interface with the kernel module and use the hardware peripheral device to transfer required data from source address to destination using DMA in burst mode [8]. We have used Ubuntu 16.04, kernel version 4.6 to implement PCIe driver.

B. User Space Driver Writing

User space device driver contains the custom logic to transfer data from source to destination, handles interrupts and performs data translation operation. As the endpoint board ZC706 is attached to the CPU, the functions from kernel object modules are called; and virtual files are created in */dev* and */sys* directories which contains configuration addresses and memory mapped addresses of different PCIEBARs and AXIBARs. The user space program handles the data transfer operations using the offset addresses [4]. The Base Address Register (BAR) mapped in memory or I/O space is used to control registers and this is the address used by any root complex or CPU to communicate with the endpoint device. The driver allocates buffers in DDR memory and the address of these buffers are written in control registers. The driver accesses the control registers and performs read and write operations from the buffer via DMA. The DMA block makes an interrupt when the tasks get completed [4]. The kernel space program may also work for different PCIe hardware designs because it uses general functions which are called, as and when the device gets attached to CPU. It registers and unregisters the driver. The device name and id are filled in PCI table. BAR is mapped in memory or I/O space and driver allocates buffer in DDR. The kernel object (*.ko* file) has been created based on the value of kernel object attribute (read, write and other permissions of kernel object file). In this way, the correct kernel variable is filled into buffer

and then updated. These buffers are written in control registers and accessed by DMA. Based on the device major and minor number, PCIe device is probed. The memory mapped addresses are given to buffers allocated previously and DMA masking is done. Masking is a process of acknowledging the kernel that our system is capable of "x" bit DMA transfer. It can be noted here that not all the CPUs are configured of using full DMA transfer. So, as a practice it is recommended to query the DMA masking and coherent mapping. At the same, PCIe devices are capable of using 64 bit DMA addressing in 64 bit CPUs. Citing from the code, a signal named *siginfo* is used to store and transfer the information of signals to user space. This signal is used in user space after data transfer completion to indicate that an interrupt has been received by user space. After successfully transferring the data from endpoint to root complex using DMA in burst mode with scatter gather mapping, the PCIe driver is removed and buffers are unmapped. The character device region is also unregistered and at last PCIe is removed by running the command *rmmod* [7], [8].

IV. DATA TRANSFER OPERATION IN DETAIL

A number of descriptors are created and DMA is used in burst mode with scatter gather mapping to perform the data transfer operation. The descriptors contain the next descriptor address to continue the chain of execution and it has also the source address, destination address and data bytes value to be transferred. A sample of a descriptor is shown in Table I [4].

TABLE I A SAMPLE DESCRIPTOR

Store descriptor at this address	Next descriptor address	Source address	Destination address	Data bytes value

These descriptors are created in application layer of PCIe devices. As per the maximum payload size of the peripheral PCIe and host PCIe devices, the data packets are formed in the subsequent three layers (as described above in Fig. 3) and then the data are transferred bit by bit following the serial transfer protocols of PCIe devices. Descriptors are of two types: 1. Address translation descriptor, 2. Data transfer descriptor. The translation descriptor increases the address offsets according to the size of transfer after every data transfer descriptor execution. It also changes the destination address and is appropriately set before the execution of data transfer descriptor. The target data transfer descriptor takes data from source address and throws it to the destination address. The destination address is changed by the AXI memory mapped PCIe as shown in Fig. 4.

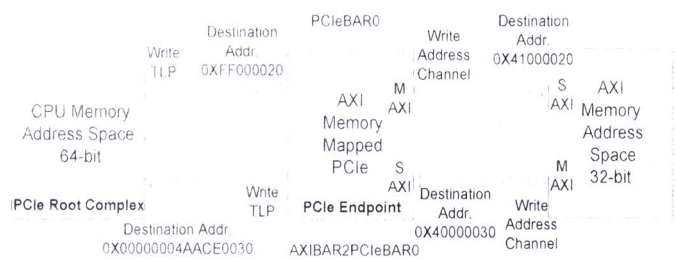

Fig. 4 Address translation from 32-bit to 64-bit

The AXI memory mapped PCIe block translates the 32-bit address to 64-bit and vice-versa. Fig. 4 is an example of address translation. The actual translation differs from it. A group of descriptors are executed in DMA burst mode with scatter gather mapping until the required data is transferred. The transferred data is accessed using AXIBARs and the data buffers addresses are printed on terminal [4].

V. RESULTS

We have developed the hardware architecture design in Vivado and PCIe device driver software. First of all, the Xilinx ZC706 board is inserted to the PCIe slot of Intel-i7 5960x CPU. When the board is inserted to the PCIe slot, it is not getting detected by CPU. The developed hardware design is dumped to the ZC706 board through JTAG master booting mode. This dumped design configures the FPGA, Zynq PS and initializes the interfaces. Fig. 5 shows the assembled hardware.

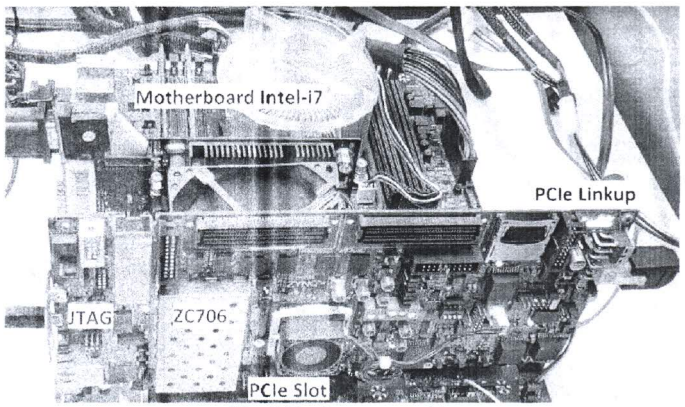

Fig. 5 Hardware assembly

The LEDs glow as the PCIe linkups. The PCIe device is searched using *lspci* on Linux OS terminal and it shows that a new PCIe device has been detected.

The designed hardware architecture in Vivado requires very less percentage of available resources like Flip-flops, LUTs, Global buffers and transceivers *etc.* as described in Fig. 6. Only 15% of available LUTs, 7% of Flip-Flops, 20% of Transceivers and 34% of Global buffers have been used. The power requirements are also very low and the total power consumption is around 4W.

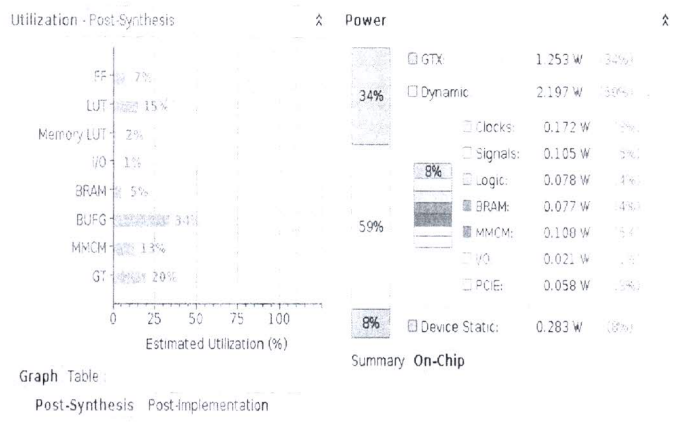

Fig. 6 Resource utilization and Power requirements

After PCIe endpoint detection by the CPU, the device driver software is inserted to the kernel. The kernel space driver creates an environment for the user space applications to be executed and manages the operations. To perform the data transfer operation from endpoint to root complex device, the user space driver software is compiled and run on the terminal. This initiates the scatter gather DMA operation and transfers data buffers from ZC706 DDR to CPU DDR through PCIe.

As an example, 40MiB data are sent using 10 data transfer descriptors of 4MiB each and 4 data translation descriptors of 8 Bytes each. Large data can be sent using more number and different sizes of descriptors. The transfer operation is performed for different sizes of data using different sizes of descriptors and a comparison table is prepared as shown below in Table II. The speed of transfer is calculated using transfer size and elapsed time during transfer.

TABLE II. DESCRIPTOR SIZE VS TRANSFER SPEED

Size (KiB)	Transfer speed (Gib/s)
128	3.70
256	3.78
512	3.85
1024	3.92
2048	3.96
4096	4.00
5120	4.01
6144	4.02
7168	4.03
<8192	4.03

Increase in descriptor size reduces the number of interrupts to be handled. This increases the speed of data transfer. It cannot be further increased beyond 8MiB due to a limitation from Xilinx IP which says 8MiB or more data cannot be sent using one descriptor [4]. The address translation descriptor is required after every 8MiB data transfer. Transferred data buffers are verified by checking at source address and the destination address using Xilinx Microprocessor Debugger (XMD) and device driver software respectively.

This implementation proposes an optimal way of achieving PCIe transfers in an efficient way. Data transfer is validated at both source and destination. The average speed of transfer has been experimentally proven as around 4 Gib/s.

VI. DISCUSSIONS

This work has good potential in the field of high speed image and video processing. It could be used to develop a Zynq SoC based high speed image frame grabber, where image processing algorithms can be easily applied at high frame rate. It uses less resources and offers low latency interface to transfer processed data from FPGA board to CPU. We expect Zynq SoC based futuristic high speed vision frame grabber systems soon in market, which could give better performance as compared to the available ones. There are major industries like Xilinx, Altera, Photron, *etc.*, working on the same to achieve faster response and also trying to get the efficient architectural designs to perform the data transfer and algorithmic processing operations. PCIe is leading in the area of high speed data transfer application interfaces. Now a days, high speed storage systems also use PCIe interface to transfer and store data speedily. The developed interface has some limitations:

A. Payload Size in Data Packets

The PCIe specification allows payload size up to 4096 bytes per packet. But, the peripheral devices and motherboards don't support the same. In this paper, PCIe gen2 v2.6 IP, Zynq SoC and Intel-i7 5960x CPU are used. The default payload size in this case is 256 bytes and it needs to be increased to speed up the transfer rate and efficiency. This can be done by modifying the PCIe configuration registers of the motherboard and the HDL configuration of the PCIe IP. As per the default payload size, we get 92% [(256 / (256+20)) * 100] efficient transfer where 20 bytes are considered as the headers and error redundancy bytes [9].

B. PL Clock Frequency

The transfer speed directly depends upon the PL fabric clock and it could be configured up to 250MHz in Vivado tool. However, PCIe clock and PL fabric clock can be increased to more than 250MHz by using external PLL (Phase Locked Loop) circuits [10]. We will try to overcome these limitations in future.

C. Descriptor Transfer Size

The supported transfer size for a single descriptor is 8MiB. Xilinx has limited the transfer size of descriptor to less than 8MiB [4].

In this paper, PCIe (2.0) x4 has been used to transfer data from endpoint to root complex device based on the developed hardware architecture design implemented on ZC706 board and created device driver software. This work can be used as an analogy in developing higher form factor PCIe interfaces [4]. DMA technique is used to transfer data from source to destination at very low latency. The transferred data buffers are verified using XMD and the device driver software. Data transfer speed is calculated based on the data buffer transfer size and time elapsed in the transfer operation. The developed hardware architecture is very simple and uses less resources and power at the same time.

VII. CONCLUSIONS

The large size data buffers are transferred from Xilinx Zynq ZC706 (DDR) to Intel-i7 5960x CPU (DDR) through PCI Express interface. The practical speed of data transfer is around 4Gib/s without using any extra clocking circuits. The speed of transfer is comparable with the available alternatives and the developed system is better in the sense of hardware design flexibility, software abstraction and resource utilization. Zynq device is capable of processing and transferring images data at high speed. The speed of transfer can be increased by overcoming the limitations. However, this hardware architecture design uses very less hardware resources to transfer data at high speed. The speed of data transfer directly depends on PL fabric clock and CPU clock. The frequencies of

these clocks can be increased by adding external PLL circuits and SMA connectors. To achieve the 100 Gib/s speed, greater form factor and newer version PCIe connector can be used [11]. The device driver is capable to work with different Zynq boards with minor modifications.

This work has a very good scope in high speed object tracking, stampede surveillance, Robotics vision and in image frame grabbers. It could also work as a high speed data storage system.

ACKNOWLEDGEMENT

This work has been partially sponsored by Department of Science & Technology (DST), New Delhi, India.

REFERENCES

[1] H. Kavianipour, S. Muschter, and C. Bohm, "High Performance FPGA-Based DMA Interface for PCIe," IEEE Trans. Nucl. Sci., vol. 61, no. 2, pp. 745–749, Apr. 2014.

[2] Idaku Ishii, Tetsuro Tatebe, Qingyi Gu, Yuta Moriue, Takeshi Takaki and Kenji Tajima, "2000 fps Real-time Vision System with High-frame-rate Video Recording", 2010 IEEE International Conference on Robotics and Automation, pp. 1536-1541, May, 2010.

[3] Artjom Rjabov, Alexander Sudnitson, Valery Sklyarov, Iouliia Skliarova, "Interactions of Zynq7000 Devices with general purpose computers through PCI-express: a case study", Proceedings of the 18th Mediterranean Electrotechnical Conference MELECON-2016, pp. 1-4, April, 2016.

[4] PCI Express Endpoint-DMA Initiator Subsystem, "XAPP1171 Xilinx Document," November, 2013.

[5] PCI Express Layers, available at http://www.veriten.com/pcie_primer.htm. Accessed on February 12, 2017.

[6] PCI Express topology and link performance, available at https://en.wikipedia.org/wiki/PCI_Express. Accessed on March 10, 2017.

[7] Jonathan Corbet, Greg Kroah Hartman and Alessandro Rubini, "Linux Device Drivers", Third Edition, Publisher: O'Reilly Media, Chapter-1, February, 2005.

[8] Jonathan Corbet, Greg Kroah Hartman and Alessandro Rubini, "Linux Device Drivers", Third Edition, Publisher: O'Reilly Media, Chapter-2, 3, 12, 15, February, 2005.

[9] Understanding Performance of PCI Express Systems, "WP350 Xilinx Document", October, 2014.

[10] Xilinx, Zynq-7000 All programmable SoC Technical Reference Manual, available at http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[11] M. Caselle, S. Chilingaryan, A. Herth, A. Kopmann, U. Stevanovic, M. Vogelgesang, M. Balzer, and M. Weber, "Ultrafast streaming camera platform for scientific applications," IEEE Trans. Nucl. Sci., vol. 60, no. 5, pp. 3669–3677, Oct. 2013.