Shared Vector Register of RISC-V for the Future Hardware Acceleration

Tomoaki Tanaka
Tokyo University of Agriculture and Technology
Tokyo, Japan
s229356t@st.go.tuat.ac.jp

Ryosuke Higashi
Tokyo University of Agriculture and Technology
Tokyo, Japan
s208653w@st.go.tuat.ac.jp

Hidetaro Tanaka
Tokyo University of Agriculture and Technology
Tokyo, Japan
s227891v@st.go.tuat.ac.jp

Takefumi Miyoshi
WasaLabo, LLC.
Tokyo, Japan
miyo@wasa-labo.com

Yasunori Osana
University of the Ryukyus
Okinawa, Japan
osana@eee.u-ryukyu.ac.jp

Jubee Tada
Graduate School of Science and Engineering, Yamagata University
Yamagata, Japan
jubee@yz.yamagata-u.ac.jp

Kiyofumi Tanaka
Japan Advanced Institute of Science and Technology
Ishikawa, Japan
kiyofumi@jaist.ac.jp

Hironori Nakajo
Tokyo University of Agriculture and Technology
Tokyo, Japan
nakajo@cc.tuat.ac.jp

Abstract

In this study, we present a vector register sharing mechanism that directly shares vector registers inside the processor with the acceleration circuitry. Because this mechanism can share the value of a vector register at a given time, high-speed communication is expected, particularly in SoC FPGAs. To validate this mechanism, this study designs and implements a processor with vector registers to obtain a preliminary estimation. The RISC-V's RV64IMV and proprietary instructions are adopted for the instruction set of the proposed processor. As a preliminary evaluation of our proposed architecture, we measured the CPI, maximum operating frequency and resource usage with and without vector extension instructions for the processor. The evaluation shows that the proposed processor can transfer data at a maximum of 787.2 [MByte/s] with the vector register sharing mechanism.

CCS Concepts: • **Hardware → Hardware accelerators**.

Keywords: RISC-V, Vector Extension, Hardware Acceleration, Vector Register

1 Introduction

In recent years, acceleration has been introduced not only in data centers or supercomputers but also in embedded devices to offload regular processing from a processor, thereby increasing the overall system speedup. In particular, GPUs, that perform graphics processing operations on the fastest systems, listed in the TOP500, are used for artificial intelligence and large-scale simulations by taking advantage of the large number of arithmetic operation units inside. However, GPUs consume large amounts of power, and although they are effective for SIMD operations, it is difficult for them to handle other complex processing operations. On the other hand, FPGAs, which allow flexible reconfiguration of acceleration circuits, are attracting attention as a hardware acceleration technology. FPGA acceleration is more flexible and consumes less power than GPU acceleration. In general, FPGA acceleration is implemented by connecting the FPGA board to an I/O bus, such as PCIe, and supplying/-collecting data between the processor and the accelerator through the bus, bringing in high data transfer latency. To overcome such delays, SoC FPGA devices, which incorporate a processor and FPGA logic on a single die, have been developed by Intel Corporation and AMD Xilinx, Inc. SoC FPGAs allow the acceleration circuitry to communicate with the processor via an internal bus, which enables communication with lower data transfer latency than when using PCIe. Data for acceleration is transferred from external memory to the FPGA's internal memory, processed by the accelerator, and then transferred back to external memory or loaded by the processor for subsequent processing. Typically, DMA is used for the data transfer to reduce the load on the processor. While DMA is more effective for large data transfers, the bandwidth of the SoC chip's internal bus limits processing performance. Therefore, transfer delays are significant for sequential processing of small amounts of data, such as streaming processing, thus performance improvement is hindered. Consequently, as a new data transfer method between a processor and a accelerator, we focus on sharing a processor's internal registers directly with an acceleration circuit to transfer data using load/store instructions to external memory. However, the size of processor registers in an ordinary SoC FPGA is limited to 32 or 64 bits, and cannot be effective for large data transfers. Therefore, we propose a vector

register sharing mechanism that enables high-bandwidth data transfer instead of sharing ordinary integer registers. To indicate the effectiveness of the vector register sharing mechanism, we have designed and implemented a processor that shares a part of vector registers using the RISC-V vector extension instruction (RVV). In this paper, we present the vector register sharing mechanism, implementation of the mechanism based on a RISC-V vector extension as well as the results of preliminary evaluation.

Section 2 describes related work on SoCs and accelerators. Section 3 shows our proposed vector register sharing mechanism. The instruction set and configuration of the proposed processor are shown in Section 4. Section 5 shows the performance of the processor as a preliminary evaluation. Finally, Section 6 indicates the results and future prospects.

2 Related Work

Rjabov et al. have studies a method to communicate with a host computer via PCIe using a device with Zynq-7000, one of the SoC FPGAs [8].

A few years later, Tanwar et al. focus on communication not between the main processors but between other peripheral PCIe-enabled devices, and achieves a 4-Gib/s transfer rate at PCIe (2.0) x4 between accelerators using PCIe in Zynq SoCs [9].

Several studies on DMA-based transfers are subsequently conducted. Kavianipour et al. have studies FPGA-based DMA transfer using PCIe. They achieve up to 748 [MByte/s] (read) and 784 [MByte/s] (write) by communicating between the FPGA and the memory on the host PC via PCIe [3].

In addition,when the processor operating frequency reaches its peak, research has been conducted on P2P DMA, which communicates directly between PCIe devices without using any processors or a DMA controller [5].

NVIDIA has achieved direct communication between different GPUs via PCIe using a technology called GPUDirect. Products using this technology have already been developed [6].

As mentioned above, most research on data transfer and DMA focus on devices with PCIe slots. However, edge computers, such as those installed on small devices, are not sufficiently large to use PCIe. Recently, SoC FPGAs have attracted considerable attention as edge-side computers, and there have been many related studies.

Jridi has examined the effectiveness of SoC FPGAs on the Internet of Multimedia Things (IoMT). In this study, the author envisions wireless image transmission and display functions as applications, as well as the ability to compress and encrypt multiple images simultaneously. In this paper, power efficiency is improved by a factor of 7.7 by incorporating many of these functions in FPGAs as hardware [2]. Zhu et al. have focused on the fact that SoC FPGAs are often used in edge-side computers in edge computing, and achieve efficiency through dynamic task scheduling [10]. Qingqing et al. have focused on the large amount of information from sensors and other sources in mobile robots and offload the hard processing to the FPGA side by using a SoC FPGA for LiDAR odometry (light-based self-position estimation) of a multi-robot, thereby increasing speed [7].

SoC FPGAs are expanding in edge computing. However, communication between the processor and FPGA in SoC FPGAs is performed via memory using an AXI bus [4]. This results in a low transfer rate when the transfer size is large because multiple memory accesses occur, and the bus is occupied for a long time.

3 Vector Register Sharing Mechanism

3.1 Key idea of the proposed mechanism

The vector register sharing mechanism is developed to bring high-speed data transfer between a processor and an accelerator in a SoC FPGAs by sharing the processor's internal vector registers directly with the accelerator, as shown in Figure 1.

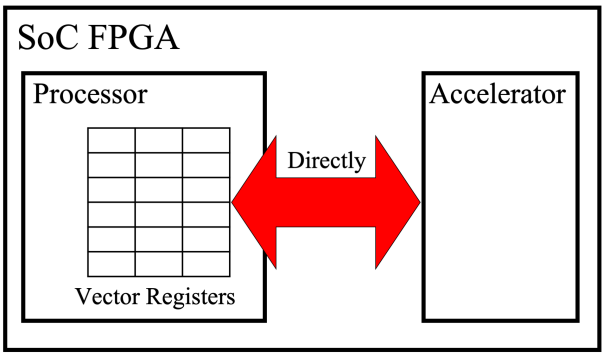

Figure 1. Vector Register Sharing Mechanism.

The transfer rate of the proposed mechanism is shown by comparing it with Xilinx's AXI DMA as an example. Because AXI can transfer data up to 1024[bit] wide, this example assumes n[bit] data larger than 1024[bit].

In Figure 2, the following procedure is used to perform the transfer.

1. The processor transfers the data to memory. (memory delay + n/1024 [clock])
2. Processor sends the address to DMAC (1[clock])
3. DMAC loads data from memory (memory delay + n/1024 [clock])
4. DMAC transfers data to the FPGA (n/1024[clock])

The following number of clocks are required in total.

- (n/1024)×3 + 2[clock]
- memory delay× 2[clock]

The transfer with our proposed mechanism is shown in Figure 3. Since the vector registers are directly shared, the

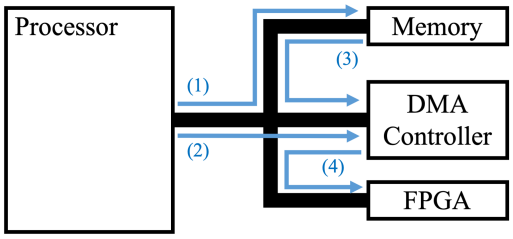

Figure 2. DMA for processor to the FPGA data transfer.

transfer can be performed in one clock regardless of the memory delay. Even if this mechanism is implemented on the AXI bus, the transfer can be done in (n/1024) [clock]. The comparison shows that the proposed mechanism is faster than DMA.

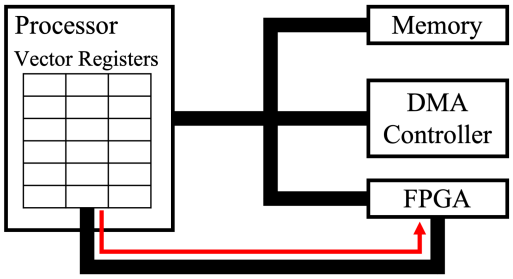

Figure 3. Data transfer from processor to the FPGA using vector register sharing mechanism.

Consider the case where the CPU is used for preprocessing or post-processing before and after offloading processing to the accelerator. For example, one may want to filter the data to be offloaded according to parameters calculated by the CPU. In this case of a transfer using AXI, the flow is as shown in Figure 4.

1. Memory transfers data to processor (memory delay + n/1024 [clock])
2. Processor runs preprocess
3. Processor transfers data to memory (memory delay + n/1024 [clock])
4. Processor sends the address to DMAC (1 [clock])
5. DMAC loads data from memory (memory delay + n/1024 [clock])
6. DMAC transfers data to the FPGA (n/1024[clock])
7. FPGA runs process
8. FPGA transfers data to DMAC (n/1024[clock])
9. DMAC stores data to memory (memory delay + n/1024[clock])
10. Memory transfers data to processor (memory delay + n/1024[clock])
11. Processor runs post-processing
12. Processor transfers data to memory (memory delay + n/1024 [clock])

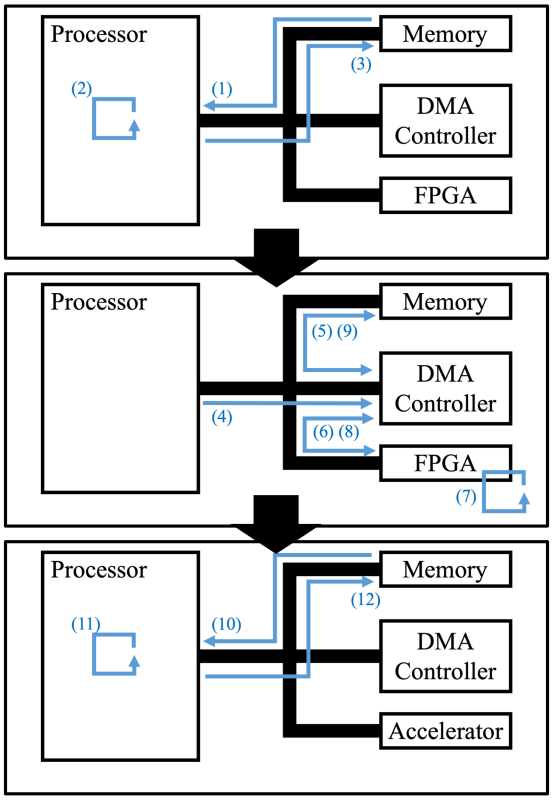

Figure 4. DMA transfer when processing before and after.

The proposed method can be executed in the order shown in Figure 5.

1. Memory transfers data to processor (memory delay + n/1024 [clock])
2. Processor runs preprocess
3. Processor transfers data to the FPGA (n/1024 [clock])
4. FPGA runs process
5. FPGA transfers data to processor (n/1024 [clock])
6. Processor runs post-process
7. Processor transfers data to memory (memory delay + n/1024 [clock])

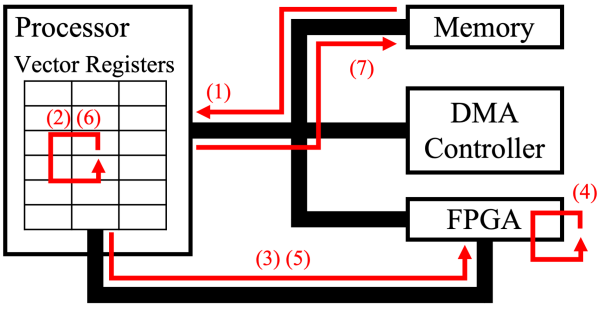

Figure 5. Transfer by vector register sharing mechanism when there is a process before or after the main process.

3.2 Current achievement

The processors inside existing SoC FPGAs cannot be identified. Therefore, to verify the vector register sharing mechanism, it is necessary to design and implement a processor with vector registers and realize functions in the following order.

1. Implement a processor with vector registers
2. Redesign the vector registers to be dual port
3. Connect accelerator to vector register
4. Enable vector load to supply data to accelerator
5. Start verification

In this paper, we designed and implemented a processor with vector registers and dual-port vector registers.

4 System Configuration

4.1 Extended ISA

The processor supports RISC-V (RV64IMV) partially. Vector extension v1.0-rc2 is adopted. The configuration parameters are implemented by assuming that the number of vector registers (LMUL) used in one instruction is less than or equal to one. The instructions are implemented with only some integer arithmetic instructions such as the RV64IM extension, and arithmetic instructions that change the width of one element (SEW) in a vector register only in the destination register (widening and reduction instructions) are not implemented.

In addition, unique instructions are implemented. To evaluate the vector register sharing mechanism, several instructions are implemented for measurement, such as an instruction to count the number of cycles and instructions of a specified process and an instruction to display the value of a specified register. In addition, to speed up matrix operations, we implement an original vector extension instruction (VACC instruction) that calculates the sum of the elements in a vector register.

4.2 Microarchitecture

The designed processor is shown in Figure 6. This processor is in-order execution and can operate in a pipeline. This processor operates in five stages. The decode, execution, and write-back stage run in separate modules for vector extension instructions and other instructions. The fetch and memory access stage modules use memory and are shared by all instructions. To avoid the READ-AFTER-WRITE hazard, stalls are placed after branch instructions, jump instructions, and load/store instructions.

4.3 Dual ported vector registers

To implement the vector register sharing mechanism, vector registers must be dual-ported to make them accessible from both the accelerator and processor. A ready/valid signal is used by the accelerator to transfer data to the processor. First, when write_valid is one, the arbiter checks the signal from the write back stage. If the signal from the write back stage indicates a write to the same vector register, the arbiter maintains its status and waits. Otherwise, the arbiter writes write_data to the specified address in the vector register and sets write_ready to one. The accelerator always outputs the value of the specified vector register to the processor without valid or ready.

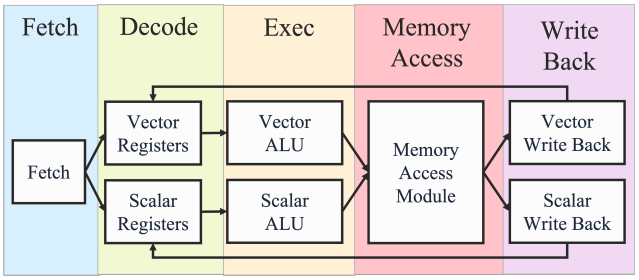

Figure 6. Organization of the processor.

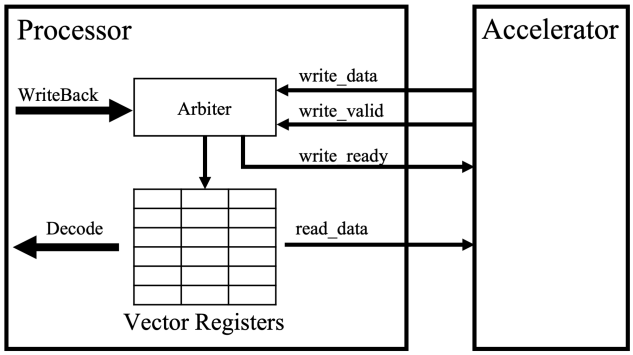

Figure 7. Dual-ported vector registers and their input/output.

5 Preliminary Evaluation

5.1 CPI

Two test benches are prepared to measure CPI. The number of cycles required for the execution stage is the same for both the RVV and scalar instructions, 1[clock]. The two test benches used to verify CPI use scalar instructions. First, a program is written in assembly language to determine whether the number n is prime. In this program, n is divided by n-1 to 2, and if there is a zero at the remainders, it is judged not to be a prime number. Next, a program is written in C to determine the nth tribonacci number. The nth tribonacci number is calculated using Equation (1).

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n \leq 2 \\ f(n-1) + f(n-2) + f(n-3) & \text{if } n > 2 \end{cases} \quad (1)$$

Table 1. Maximum operating frequency/Resource usage.

Processor Type	Result		Utilization in Alveo U250[%]	
	without RVV	with RVV	without RVV	with RVV
Maximum Operating Frequency[MHz]	59.72	9.84	-	-
LUT	32743	933660	1.89	54.03
FF	6106	51862	0.18	1.50
DSP	50	1262	0.41	10.27

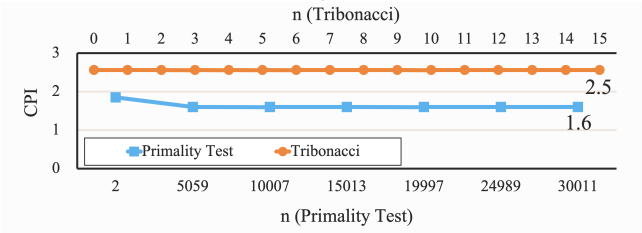

Figure 8. CPI measurement results.

The CPI have been measured as shown in Figure 8.

In the measurement, both data and instruction memory are simulated without a memory delay. The CPI for prime number determination is 1.60, and the CPI for tribonacci number calculation is approximately 1.75.

The difference between the two test benches is in the frequency of stall signals. The source code for prime number evaluation is written in assembly language. This program does not have a load/store instructions inside the loop. By loading and storing memory before entering the loop, memory accesses are not performed in a process that is repeated n times. Conversely, the source code for the tribonacci number calculation is written in C language for recursive processing. Because no optimization options are used at compile time, many load and store instructions are used before and after the function. Therefore, many load and store instructions are used in this program. Additionally, many jump instructions are executed using function calls. Because these instructions require stalls, the CPI of the tribonacci number is slightly higher.

5.2 Speed of matrix multiplication with V instructions

Matrix multiplication (matmul) refers to the operation in equation 2. In this test bench, A, B, and C are each $n \times n$ square matrices.

$$C = AB. \tag{2}$$

To compare the speed of matmul, two programs were prepared for this equation.

Without the vector extensions, the program is a triple loop as shown below.

Listing 1. matmul without RVV and the original instruction.

```
1 for(i){
2   for(j){
3     for(k){
4       C[i][j] += A[i][k] * B[j][k];
5     }
6   }
7 }
```

With the vector extensions, the matrix multiplication code is written in a double loop, as in the following program.

Listing 2. matmul with RVV and the original instruction.

```
1 for(i){
2   for(j){
3     vmul.vv v1, A[i], B[j]
4     vacc C[i][j], v1
5   }
6 }
```

The number of clocks for matrix multiplication with and without vector extensions are compared using these test benches. The results are shown in Figure 9.

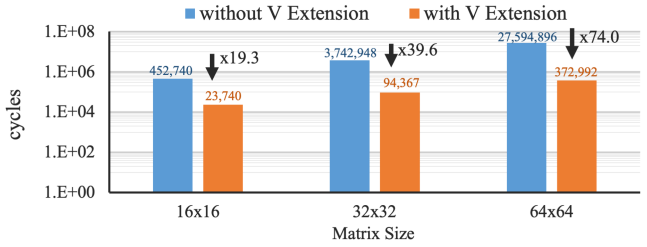

Figure 9. Comparison of the speed of matmul.

The results indicate that the number of clocks required to perform matrix multiplication decreases when vector extensions are used. The original instruction (VACC) reduces the number of statements, which may have resulted in an exponential decrease in the number of clocks with respect to matrix size n.

5.3 Operating frequency/Resource usage

The maximum operating frequency and resource usage are measured with and without RVV. The vector register length is 1024[bit]. Vivado 2020.3 is used for the verification. The target device for the synthesis is Xilinx Alveo U250. The operating frequency and resource usage are listed in Table 1.

These results show that processors with RVV have a significant increase in resource usage and a significant decrease in operating frequency have compared to processor without RVV. The processor with RVV is found to be the most feasible processors for FPGAs. The maximum CPI is approximately 1.6. The maximum transfer rate of the vector register sharing mechanism is 787.2 [MByte/s].

6 Conclusion

We have proposed a vector register sharing mechanism for fast data transfer between a processor and an accelerator. In the mechanism, some vector registers inside the processor are shared directly with acceleration circuits. After we compared the transfer speed of this mechanism with DMA, we have confirmed that the vector register sharing mechanism can achieve faster transfers for hardware acceleration.

From the preliminary evaluations, the implemented processor performs with the CPI as at most 1.6. Moreover, the processor executes matrix multiplication up to 73.9 times faster when using V-extensions with the maximum operating frequency as 9.84[MHz]. Therefore, when the vector register sharing mechanism is used, this processor can transfer data at a maximum rate of 787.2 [MByte/s].

Future work

One of the reasons for the low maximum operating frequency of the processor is that multiplication and division circuits are implemented without a pipeline, and the frequency is limited by the critical path. To overcome this problem, we will improve the frequency by pipelining for the multiplication and division circuits.

Longer vector registers for RISC-V are required to improve the transfer rate. Currently, 54 [%] of the LUTs are used for most vector registers and their arithmetic components.

In the future, we plan to implement vector registers on the BRAM so that they can be operated asynchronously. This is expected to reduce LUT utilization and allow longer vector registers to be implemented.

In addition, it is necessary to connect the vector register sharing mechanism to the accelerator and compare the speed with AXI DMA. This comparison should confirm not only the transfer speed but also the speed improvement in the actual application. Therefore, it is necessary to compare the time taken and the number of clocks by running the processes that are often seen in edge-side computers. For the purpose, it is necessary to determine the application, implement logic to accelerate the application on the FPGA, and implement the AXI bus and a proprietary bus to support DMA transfers and vector register sharing mechanisms[1].

Acknowledgments

This research is partly supported by a project, JPNP16007, commissioned by the New Energy and Industrial Technology Development Organization (NEDO) and JSPS KAKENHI, Grant Numbers 21K11804 and 19K11879.

References

[1] Chen Chen, Xiaoyan Xiang, Chang Liu, Yunhai Shang, Ren Guo, Dongqi Liu, Yimin Lu, Ziyi Hao, Jiahui Luo, Zhijian Chen, Chunqiang Li, Yu Pu, Jianyi Meng, Xiaolang Yan, Yuan Xie, and Xiaoning Qi. 2020. Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance RISC-V processor with vector extension : Industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (Valencia, Spain). IEEE. https://doi.org/10.1109/ISCA45697.2020.00016

[2] Maher Jridi, Thibault Chapel, Victor Dorez, Guénolé Le Bougeant, and Antoine Le Botlan. 2018. SoC-based edge computing gateway in the context of the internet of multimedia things: experimental platform. *Journal of Low Power Electronics and Applications* 8, 1 (2018), 1. https://doi.org/10.3390/jlpea8010001

[3] Hossein Kavianipour, Steffen Muschter, and Christian Bohm. 2012. High performance FPGA-based DMA interface for PCIe. (June 2012). https://doi.org/10.1109/RTC.2012.6418352

[4] Rodrigo A Melo, Bruno Valinoti, Marie Baly Amador, Luis G Garcia, Andres Cicuttin, and Maria Liz Crespo. 2019. Study of the data exchange between programmable logic and processor system of zynq-7000 devices. In *2019 X Southern Conference on Programmable Logic (SPL)* (Buenos Aires, Argentina). 3–8. https://doi.org/10.1109/SPL.2019.8714328

[5] Ryo Nakamura, Yohei Kuga, and Kunio Akashi. 2020. How beneficial is peer-to-peer DMA?. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*. 25–32. https://doi.org/10.1145/3409963.3410491

[6] NVIDIA GPUDirect 2020. *NVIDIA GPUDirect.* Retrieved March 01, 2022 from https://developer.nvidia.com/gpudirect

[7] L Qingqing, F Yuhong, J Peña Queralta, TN Gia, H Tenhunen, Z Zou, and T Westerlund. 2019. Edge Computing for Mobile Robots: Multi-Robot Feature-Based Lidar Odometry with FPGAs. In *2019 Twelfth International Conference on Mobile Computing and Ubiquitous Network (ICMU)*. Kathmandu, Nepal, 1–2. https://doi.org/10.23919/ICMU48249.2019.9006646

[8] Artjom Rjabov, Alexander Sudnitson, Valery Sklyarov, and Iouliia Skliarova. 2016. Interactions of Zynq-7000 devices with general purpose computers through PCI-express: A case study. In *2016 18th Mediterranean Electrotechnical Conference (MELECON)* (Lemesos, Cyprus). 1–4. https://doi.org/10.1109/MELCON.2016.7495400

[9] Pramod Kumar Tanwar, Om Prakash Thakur, Kritik Bhimani, Gaurav Purohit, Vipin Kumar, Sanjay Singh, Kota Solomon Raju, Idaku Ishii, and Sushil Raut. 2017. Zynq SoC based high speed data transfer using PCIe: A device driver based approach. In *2017 14th IEEE India Council International Conference (INDICON)* (Roorkee). 1–6. https://doi.org/10.1109/INDICON.2017.8487747

[10] Zongwei Zhu, Junneng Zhang, Jinjin Zhao, Jing Cao, Duan Zhao, Gangyong Jia, and Qingyong Meng. 2019. A hardware and software task-scheduling framework based on CPU+FPGA heterogeneous architecture in edge computing. *IEEE Access* 7 (2019), 148975–148988. https://doi.org/10.1109/ACCESS.2019.2943179