Preliminary Evaluation of SHAVER: Sharing Vector Registers with an Accelerator

Tomoaki Tanaka
tomoaki-tanaka@st.go.tuat.ac.jp

Tokyo University of Agriculture and Technology
Michiya Kato
Tokyo University of Agriculture and Technology
Yasunori Osana
Kumamoto University
Takefumi Miyoshi
WasaLabo, LLC.
Jubee Tada
Yamagata University
Kiyofumi Tanaka
Japan Advanced Institute of Science and Technology
Hironori Nakajo
Tokyo University of Agriculture and Technology

Research Article

Keywords: RISC-V, Hardware Acceleration, FPGA, SoC

Posted Date: November 6th, 2024

DOI: https://doi.org/10.21203/rs.3.rs-5328960/v1

Additional Declarations: No competing interests reported.

Preliminary Evaluation of SHAVER: Sharing Vector Registers with an Accelerator

Tomoaki Tanaka[1*], Michiya Kato[1], Yasunori Osana[2], Takefumi Miyoshi[3], Jubee Tada[4], Kiyofumi Tanaka[5], Hironori Nakajo[6*]

[1*]Department of Electrical Engineering and Computer Science, Graduate School of Engineering, Tokyo University of Agriculture and Technology, 2–24–16 Nakacho, Koganei, 184–8588, Tokyo, Japan.
[2]Research and Education Institute for Semiconductors and Informatics, Kumamoto University, 2–39–1 Kurokami Chuo–ku, Kumamoto, 860–8555, Kumamoto, Japan.
[3]WasaLabo, LLC., K-2 Buransyu 6–5–20 Minaminaruse, Machida, 194–0045, Tokyo, Japan.
[4]Graduate School of Science and Engineering, Yamagata University, 1-4-12 Kojirakawa-machi, Yamagata, 990-8560, Yamagata, Japan.
[5]Japan Advanced Institute of Science and Technology, 1-1 Asahidai, Nomi, 923-1292, Ishikawa, Japan.
[6*]Division of Advanced Information Technology Computer Science, Institute of Engineering, Tokyo University of Agriculture and Technology, 2–24–16 Nakacho, Koganei-shi, 184–8588, Tokyo, Japan.

*Corresponding author(s). E-mail(s): tomoaki-tanaka@st.go.tuat.ac.jp; nakajo@cc.tuat.ac.jp;
Contributing authors: s231841x@st.go.tuat.ac.jp; osana@kumamoto-u.ac.jp; miyo@wasa-labo.com; jubee@yz.yamagata-u.ac.jp; kiyofumi@jaist.ac.jp;

Abstract

In recent years, demand for data-parallel processing has been growing, and this parallelism often appears in AI processes. One method to accelerate these processes is using DSA, domain-specific architecture. A common data transfer method on DSA is DMA, which is direct memory access. There are several studies on DMA-based accelerators. However, few studies focus on data transfer methods. In this paper, a vector register-sharing mechanism has been proposed as a new data transfer method. Our proposed mechanism is named ”SHAVER”. In this mechanism, a part of vector registers is directly shared with an accelerator. An open-source RISC-V vector co-processor is used to evaluate the mechanism’s potential. It has been implemented on an FPGA and a simulator for the evaluations. The results indicate the possibility of the proposal mechanism to achieve a maximum of 8.38% speedup over DMA transfer.

Keywords: RISC-V, Hardware Acceleration, FPGA, SoC

1 Introduction

AI tasks and image processing have been used in many recent applications. A recent AI application has spread in our work and life. Many smartphones use AI technology for face recognition to take a good picture and unlock the device [1] [2]. In addition, Microsoft released Microsoft 365 Copilot. It is an AI-powered assistant integrated into Office apps that helps users automate tasks, generate content, and boost productivity. It leverages large language models to provide personalized recommendations and streamline workflows [3]. For comfortable engineering, many AI assist applications are released around the world. Many companies have released an AI copilot service that supports us in writing and designing a program. GitHub, Inc. released GitHub Copilot, which assists the programmer and is published as free only for students [4]. AI-powered tools are becoming increasingly widespread across various applications, and the acceleration of AI processes has a significant impact. Optimizing AI performance not only enhances user experiences but also opens new possibilities in efficiency and scalability across industries. AI tasks and processes often contain instructions with data-level parallelism. The processes with data-level parallelism perform the same operation on multiple data, such as matrix multiplication.

One method of speeding up these processes is the implementation of domain specific architecture (DSA). DSA is an architecture specialized for a specific process and is usually used with a main core. The main core offloads specific processes to the DSA to accelerate the processes. An accelerator is often used as the main core’s hardware to send the heavy process. For example, Google developed the TPU (Tensor Processing Unit), which many AI developers use today. TPU also uses a technique that offloads some processing to an internal accelerator [5]. With the slowdown of Moore’s Law, this type of DSA-based acceleration, other than general-purpose processor acceleration, has gained a high degree of attention in both commercial and academic research.

In addition, recent tasks have tended to use large amounts of data for processing. The recent tasks that use images and videos appear in many applications. The resolution of images and videos, such as WQHD and 4K, is higher than ever. Therefore, it is important to be able to transfer large amounts of data to DSA as quickly as possible to increase the speed of recent applications. However, little research has been involved on data transfer to accelerators.

Normally, when two components (like CPU and an accelerator) share data via memory,
they need to make sure the data is the same (consistent) — this is cache coherence.
But with SHAVER:
There's no need to use memory or shared caches.
Data stays in the vector registers, not in the main memory.
Since the vector registers are directly shared with the accelerator, and not cached separately,
there's no need for cache coherence mechanisms.

The current data transfer rate depends on the width of the data bus. The SHAVER, which shares vector registers between the accelerator and processor, is proposed to avoid this dependence and achieve faster communication than now. Fig.1 shows the brief of this mechanism.

This mechanism enables high-speed data transfer to the accelerator because some vector registers are shared without a bus. Modern processors generally have a data cache to reduce overheads when accessing memory. SHAVER utilizes the data within the vector processor, allowing it to leverage the lowest-level cache similarly to the main processor. Therefore, SHAVER can communicate with the accelerator without worrying about data cache consistency.

In this paper, we implement SHAVER and a Direct Memory Access (DMA) Controller on FPGA using DDR4 memory. We also discuss aspects of the data transfer method through evaluation in simulation and implementation of FPGA.

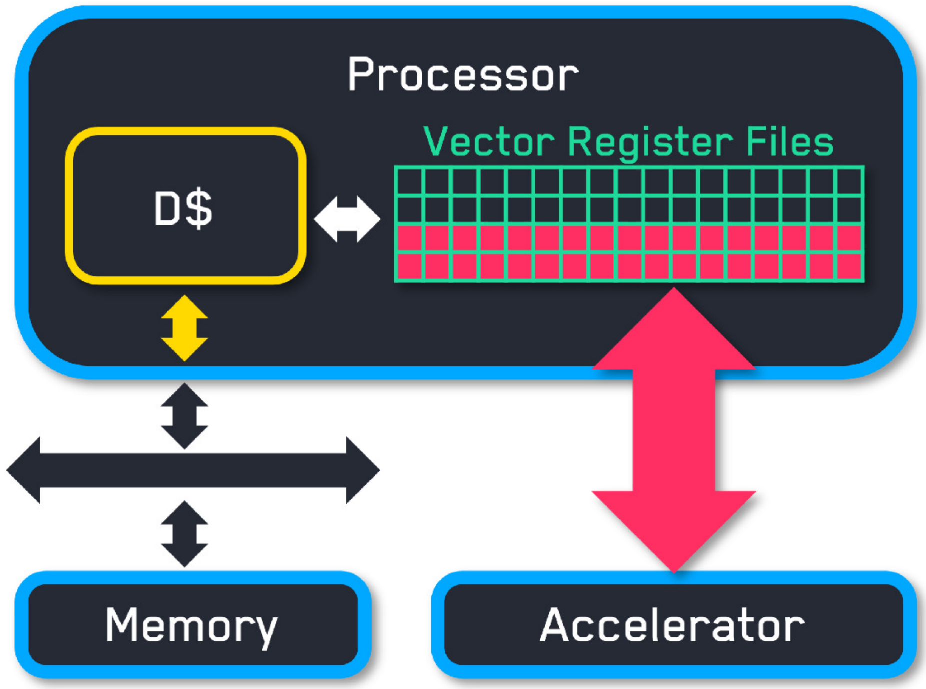

Fig. 1 Data transfer method in this research. (SHAVER: Sharing Vector Registers)

2 Background and Related Work

2.1 System on Chip

System on Chip (SoC) is a device that integrates multiple components of a computer or electronic system into a single integrated circuit. They typically include a CPU and memory modules such as RAM and ROM. SoC is small in size and has low power consumption due to the high integration of each component. For this reason, SoC has been used in most smartphones. In recent years, SoC has included a GPU and a hardware accelerator. By consolidating these elements onto a single chip, SoCs offer significant advantages in terms of power efficiency, reduced physical footprint, and enhanced performance for targeted applications.

For example, Raspberry Pi, a well-known device for embedded systems, is also equipped with a SoC. Raspberry Pi 5 is equipped with BCM2712 developed by Broadcom [6]. In recent years, the Raspberry Pi Foundation has also developed the RP2350. This SoC is attracting attention because of its mix of Arm and RISC-V cores [7]. Apple has developed some SoC devices. As a chip for a smartphone, A series SoCs have been developed. The recently released iPhone 16 is equipped with the A18 chip [8]. The ship has a 6-core CPU, unified memory units, and a 5-core GPU. In addition, Neural Engines are implemented on A18 to accelerate AI processing. Not only that, Apple has also developed M series chips for Mac series computers. M3 chips include an 8-core CPU, unified memory units, and a 10-core GPU [9]. Intel has also developed an Intel Core Ultra processors series SoC. This SoC targets notebook PCs and includes a neural engine processing unit (NPU) to accelerate AI applications [10].

Due to their low power consumption and compact size, SoCs are widely utilized across various devices. In particular, accelerators are installed in many SoCs to meet the growing demand for faster AI. A more efficient data transfer method is essential to take advantage of these accelerators. So SHAVER, the proposed data transfer method, focuses on a SoC device. In this method, the vector registers of a main processor are connected to an accelerator directory. The vector register can hold large amounts of data. Thus, SHAVER allows large amounts of data to be transferred at once. In addition, a main processor can send the data to an accelerator immediately after preprocessing it.

2.2 Accelerator

IoT has been spreading worldwide, so there have been demands for digital signal processing and AI processing in embedded systems in recent years. Accelerators are used to achieve faster processing in some embedded systems. Xiaowen Chen et al. implemented a variable-size Fast Fourier transform (FFT) accelerator in 2018 [11]. FFT is a very common process in the domain of digital signal processing. Commercial devices have been released with the implementation of an FFT accelerator. For example, Sipeed has released the M1 module that implemented FFT accelerator [12]. Many AI domain accelerators have been developed to perform faster AI tasks on edge devices. Kyubaik Choi et al. have developed a low-cost convolutional neural network (CNN) accelerator focused on edge AI using FPGA in 2022 [13]. Commercially, Katsushige

Matsubara of Renesas Electronics Corporation et al. have developed a processor that targets applications for autonomous driving systems. CNN accelerator is implemented on this processor [14].

Many accelerators can be classified into a tightly-coupled accelerator and a loosely coupled accelerator. A tightly-coupled accelerator and a loosely-coupled accelerator exist as a type of accelerator. A tightly-coupled accelerator is an accelerator that is connected to a main processor tightly. A loosely coupled accelerator is connected to a bus, not a main processor. Our proposed mechanism and a tightly coupled accelerator are especially alike. Alireza Amirshahi et al. have implemented tightly coupled small-scale systolic arrays(TiC-SATs) in the CPU. The accelerator is connected to general-purpose registers implemented by the CPU [15]. The study focused on recently quantized applications such as quantized neural networks (QNN). However, we have proposed a mechanism to get more performance out of an accelerator that can give and receive big data at once, not only for the acceleration of quantized applications.

This paper uses a simple accelerator for convolution to evaluate our proposed mechanism's performance.

2.3 Direct Memory Access

A commonly used method of transferring data to an accelerator is DMA. This transfer method is used on many devices. For example, ARM has developed DMA-230, a micro DMA controller for Cortex-M series cores, and achieved lower power consumption [16]. The ESP32 series, released by Espressif, is a popular series of microprocessors for IoT because they can use 2.4 GHz Wi-Fi and Bluetooth with no other hardware requirements. They also have a DMA controller [17]. In recent years, the use of GPUs has become increasingly widespread, and DMA plays a crucial role in many of these applications. For example, NVIDIA has introduced a data transfer technology known as NVIDIA GPU Direct. This mechanism utilizes DMA to facilitate high-speed and low-latency data transfers from peripheral devices, such as Network Interface Cards (NICs), directly to NVIDIA GPUs, thereby enabling efficient data processing [18]. Infinity Fabric, released by Advanced Micro Devices (AMD), is designed to provide a high-speed communication pathway between CPU cores, GPUs, memory, and other peripherals. By employing DMA, Infinity Fabric enables direct data transfers between these components without burdening the CPU, thereby reducing latency and improving overall system throughput. The use of DMA within AMD's Infinity Fabric optimizes performance by minimizing CPU intervention during large data transfers, resulting in better resource utilization and lower processing overhead [19].

Fig. 2 shows the brief flow of DMA transfer. In this data transfer method, the processor first notifies the DMA controller with the address to write and read and the data length to be transferred. Then, the main processor gives the DMA controller control of the bus connected to the memory and accelerator. The DMA controller is connected to the bus to allow direct communication between the devices to read from and write to. Communication is then performed between both devices. When the communication is completed, the DMA controller returns the bus mastership to the main processor. While the DMA controller has bus mastership, the main processor cannot access the memory or accelerators but can execute other instructions. After these flows, this system of peripherals and processors returns to normal operation.

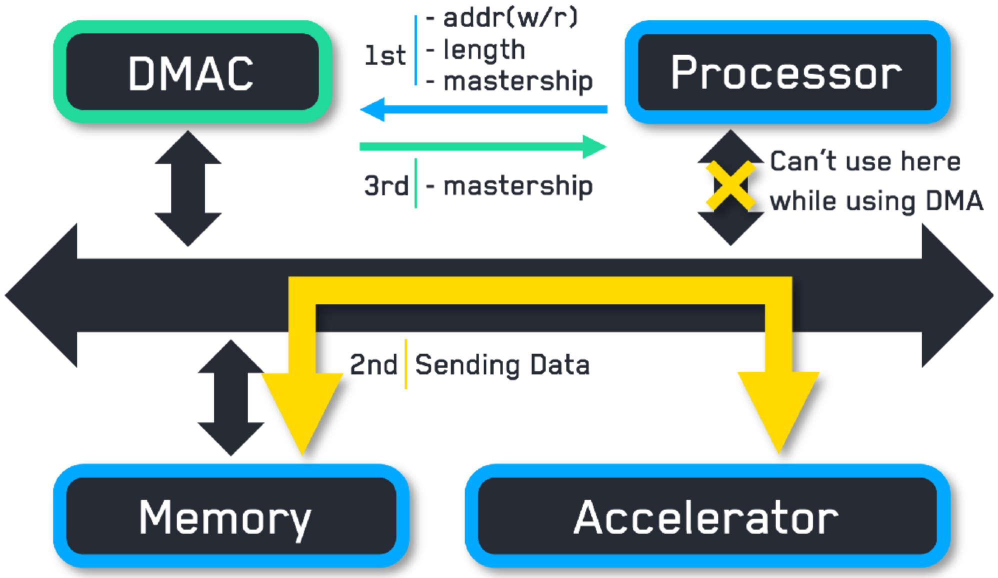

Fig. 2 Direct Memory Access.

Several studies have already been conducted on accelerators using DMA. However, Li Zhao et al. show us some problems with DMA transfer. DMA transfer frees the core from I/O operations. On the other hand, a significant latency occurs on DMA transfer by setup processes [20]. Almost all accelerators require a certain amount of data to maximize performance. In addition, it transfers the data with DMA transfer many times because the size of the data is bigger than the limit of the size of the DMA transfer. Therefore, the setup latency of DMA transfer prevents building a larger accelerator even if an accelerator can be implemented with a bigger size on a chip than now. In other words, DMA transfer cannot achieve the ideal performance of an accelerator. In general, working cache coherency and DMA together is a difficult problem. To keep cache coherence during DMA transfers as a automatically makes hardware complication. On the other hand, explicit cache content dispatching by the processor incurs a large overhead. Our proposed method, SHAVER, uses a load/store unit in the vector processor to maintain cache consistency. In addition, using SHAVER, the data preprocessed in the vector register can be immediately transferred to the accelerator. This is particularly effective in systems with cache hierarchies, where processing takes place before and after the accelerator. In this paper, simple transfer performance is compared with DMA, and further experiments are conducted when preprocessing is added.

2.4 Vector Processor

Vector processors are categorized as a SIMD architecture that can take advantage of data-level parallelism. In general, a vector processor consists of the following elements.

- Vector register file
  A vector register is a register for storing multiple data, like an array in memory. They have a larger data width than general-purpose registers and are generally designed to store a large number of elements.
- Vector functional unit
  The vector functional unit is a unit calculating each element in a vector register. The unit is pipelined so that every clock cycle can perform a new operation.
- Vector load and store unit
  This unit can control the communication between a processor and a memory.

The typical vector processor process flow starts with loading values from memory into vector registers. Then multiple data elements can usually be loaded in a single instruction, and vector load-store units support relatively efficient load operations compared to general purpose processors due to pipelining. After the values have been loaded into the vector processor, operations are performed by the vector functional units. These units use the values in the vector registers to process each element. Instead of calculating all the elements in one run, the vector functional unit performs the calculation one element at a time and writes it back to the vector register. This process is repeated sequentially, and the result is stored in memory. Vector processors are an architecture that performs calculations efficiently by pipelining rather than computing one at a time.

Vector processors' history began in the 1970s with the Cray-1 supercomputer from Cray Research, which was equipped with a vector processor. This computer was the fastest at the time, and from there, many supercomputers worldwide began to support vector processing [21]. Cray Research further developed various vector processor series from there, including the Cray-2, X-MP, Y-MP, C90, T90, and SV1. Vector processor development was also carried out in Japan, and Fujitsu developed the FACOM 230-75APU supercomputer using a vector processor, after which NEC, Hitachi, and various other companies began to create vector-processing computers [22].

2.5 RISC-V

RISC-V is an instruction set architecture (ISA) developed at the University of California, Berkeley (UCB) in 2011. The ISA specification is open and public and is provided in a format that anyone can freely access and use, unlike other architectures such as x86 and ARM [23].

RISC-V provides RV32I and RV64I as the basic instruction set. RV32I is a 32-bit basic instruction set, and RV64I is a 64-bit basic instruction set. RV64I is the instruction set that several 64-bit instructions are added to RV32I. These instruction sets contain only the basic instructions for integer operations. Some other instructions are provided as extension instruction sets, such as integer multiplication/division instructions (M extensions), floating-point arithmetic instructions (F extensions, D extensions), compressed instructions(C extensions), and atomic instructions (A extensions). In particular, the combination of the basic instruction set (I), M extension, F extension, D extension, and A extension is also called G extension [23].

Commercially, some companies have developed a RISC-V core. For example, Andes Core N22 has been developed by Andes Technology. This core includes a 32-bit RISC-V processor with a 2-stage pipeline [24]. Besides N22, Andes Core AX65, which is a 64-bit Out-of-Order RISC-V core, has been developed [25]. In addition, as mentioned earlier, unlike other architectures such as x86 and ARM, RISC-V has an open specification and various open source cores exist. For example, UCB, the developer of RISC-V, has developed a relatively large RISC-V core called BOOM (The Berkeley Out-of-Order Machine, BOOM) [26]. This core supports RV64GC and all source code is available on github [27]. In addition, Rocket core has been developed by UCB [28]. It can be generated and customized through a Rocket Chip Generator. There is also another open-source RISC-V core called PicoRV32, which is a very small core written in a single Verilog file [29]. As these, RISC-V cores, both large and small, exist as open-source cores. They are very inexpensive compared to x86 and ARM cores and can be easily modified for experimentation. It was decided to use RISC-V cores for the evaluations of this paper.

2.6 RISC-V Vector Extensions

RISC-V Vector Extension (RVV) is one of the extended instruction sets of RISC-V. The first frozen version of 1.0 of RVV was released in 2021, making this extension a relatively new instruction set [30].

In RVV, there are 32 vector registers, v0 through v31. The length of the vector register (VLEN) is implementation-dependent. The size of the elements to be stored in them (ELEN) can be selected by the user as 8-bit, 16-bit, 32-bit, or 64-bit. ELEN and the length of vector registers can be set by manipulating the designated control status register (CSR).

Various RVV-compatible processors have been developed even before 2021. Matthew Johns et al. have developed a microprocessor that supports RVV version 0.8 in 2020 [31]. Guillem Cabo et al. also developed an ASIC implementation of a vector architecture for RVV version 0.7.1 in 2022, called the DRAC Vector In-Order (DVINO) processor. Several other commercial cores also exist [32]. For example, SiFive released the X280, a core that supports RVV for AI and machine learning at the edge [33].

To evaluate our proposed transfer method, a processor has to be able to be customized by ourselves. So, an open-source RVV core was used in this study.

2.7 RV64GC Processor: CVA6

CVA6 is an in-order processor implementing RV64GC developed by the Swiss Federal Institute of Technology in Zurich (ETH) and the University of Bologna [34]. This processor consists of 6-stages and a cache subsystem. The cache subsystem manages an instruction cache and a data cache. In addition, the source code of this processor is open source and available on github [35]. It is written in SystemVerilog and has an interface for communicating with Ara, a co-processor supporting RVV. Only vector extension instructions are identified and sent to that interface.

2.8 RVV Co-processor: Ara

Ara is a co-processor implementing RVV, which was developed by ETH and the University of Bologna. Ara works in tandem with CVA6 [36]. An implementation of Ara mounted on CVA6 and a simulation evaluation environment are available on Git Hub. The core was released in Dec. 2019 before the RVV specification was frozen. The first implementation was based on the version 0.5 draft of RVV [37]. After the frozen version 1.0 of RVV was released, Ara started to support its instructions. The brief block diagram of this system is shown in Fig.3.

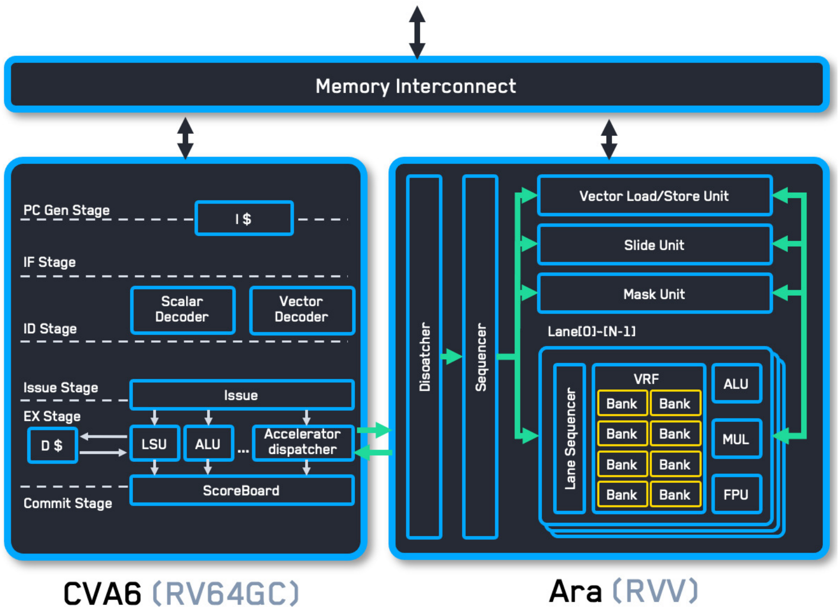

Fig. 3 Brief block diagram of system constructed CVA6 and Ara.

Vector extension instructions are sent from CVA6 to Ara. In Ara, the dispatcher receives instructions and sends the instructions to each lane after the sequencer arbitrates between these instructions, vector load/store unit, and slide unit. When the instruction is finished to execute, the sequencer sends an Ack signal or scalar result to CVA6. Vector processors execute instructions in parallel by distributing processing across lanes. In Ara, the number of lanes is variable and can be changed to 1, 2, 4, 8, or 16. In addition, VLEN can be selected from 128 bits, 256 bits, 512 bits, 1024 bits, 2048 bits, or 4096 bits. The vector register file (VRF) is divided into eight banks per lane. All elements of the vector registers are placed on the banks. The banks are designed to avoid access conflicts by arithmetic units and other units out of the lane. Therefore, all banks can be implemented with a one-port SRAM on a hardware device such as an FPGA because each bank has only one port to read and write.

However, Ara doesn't contain data caches. In this paper, Ara and CVA6 are used for evaluation. When measuring the performance of the SHAVER, the performance with a data cache is estimated from a result on a simulator.

3 Implementation

3.1 CVA6 and Ara on an FPGA

A way to execute a system of CVA6 and Ara on a simulator is published on the GitHub repository [36]. However, there are no publications about implementing this system using the FPGA. So, some modules of CVA6 and Ara have been modified to be implemented on an FPGA. Displaying characters is required to measure the performance of a system executing on an FPGA. So, a serial communication module is implemented as a memory-mapped device. This module allows characters to be displayed through a C function. Moreover, it allows the main memory to be rewritten via a user serial console. Alveo U250 is used for evaluation in this paper [38]. This board has four 16 GB DDR4 memories. In addition, this board includes one of UltraScale+ Devices, so the UltraRAM (URAM) is supported [39]. In our implementation, the main memory is replaced by DDR4 memory from a BRAM to measure the latency of DMA transfer. All banks of vector register files on Ara are implemented with URAM through XPM_MEMORY_SPRAM macro. The macro, available only in Vivado, can generate a single-ported RAM [40].

3.2 DMA Controller

The Ara and CVA6 implementations published on GitHub do not provide a DMA controller. Hence, the DMA controller is implemented independently. The published systems of Ara and CVA6 use the AMBA AXI bus. Therefore, a DMA controller is connected to the subordinate side of AXI so that it can communicate with Ara and CVA6 through its interface. A DMA controller can be accessed as one of the memory-mapped devices. The length of data and addresses are needed to be written and read. Before starting the DMA transfer, they are sent to the DMA controller from Ara and CVA6. After that, Ara and CVA6 signal that the DMA transfer starts to the DMA controller. After receiving the signal from the processor, the DMA controller switches buses on AXI Interconnect to connect between subordinate devices. After all data has been sent, the DMA controller restores buses between the AXI interconnect and devices. The data size that can be sent in one burst transfer on AXI depends on the width of the data signal on the AXI interface. The AXI interface is capable of 256 beats burst transfers. For example, if the data signal width is 128 bits, the data can transfer up to 256 times in one transaction. Thus, in this example, the maximum data size is 32,768 bits per transaction [41]. The brief block diagram of this system is shown in Fig. 4. An example of C programming for using DMA is shown by Listing 1. For startup DMA transfer, some parameters are written before starting the transfer.

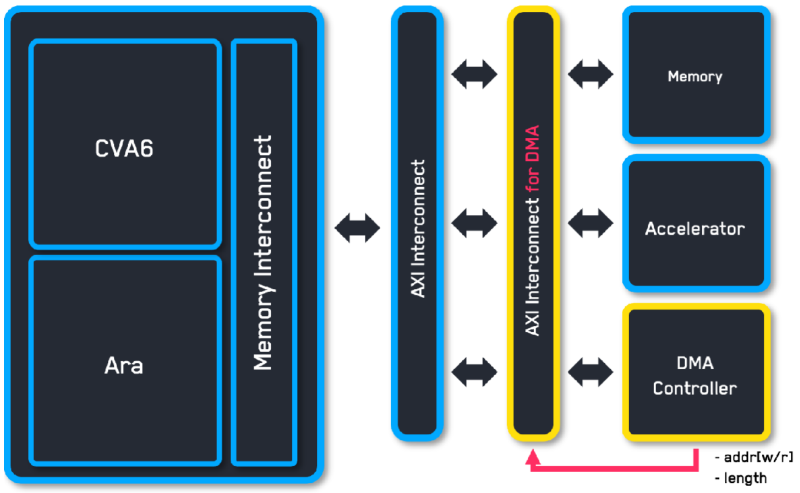

Fig. 4 Implemented DMA Controller.

Listing 1 Example of DMA (memory to an accelerator).

```c
// Memory Map
//   DMA parameters
extern uint32_t dma_len_bits,dma_start;
extern uint32_t *dma_addr_read;
extern uint32_t *dma_addr_write;
//    Accelerator
extern uint32_t *acc;
int main(void){
  uint8_t source_array[512];
  ...
  // Sending length of data to DMAC
  dma_len_bits = 8*512;
  // Sending a first address of a read region
  dma_addr_read  = source_array;
  // Sending a first address of a written region
  dma_addr_write = acc;
  // Waiting until finished to set parameters
  _asm_ volatile("fence.i");
  // Start DMA transfer
  dma_start      = 1;
  // Waiting until finished to transfer
  _asm_ volatile("fence.i");
  ...
}
```

3.3 SHAVER: Sharing Vector Registers

In our proposed mechanism, an accelerator is connected to Ara directly, not through the memory interconnect module in the core. For the connection, some modules are designed to sort the values of the banks. In addition, the signals to control an accelerator are designed to use an accelerator. The brief block diagram of SHAVER is shown in Fig. 5.

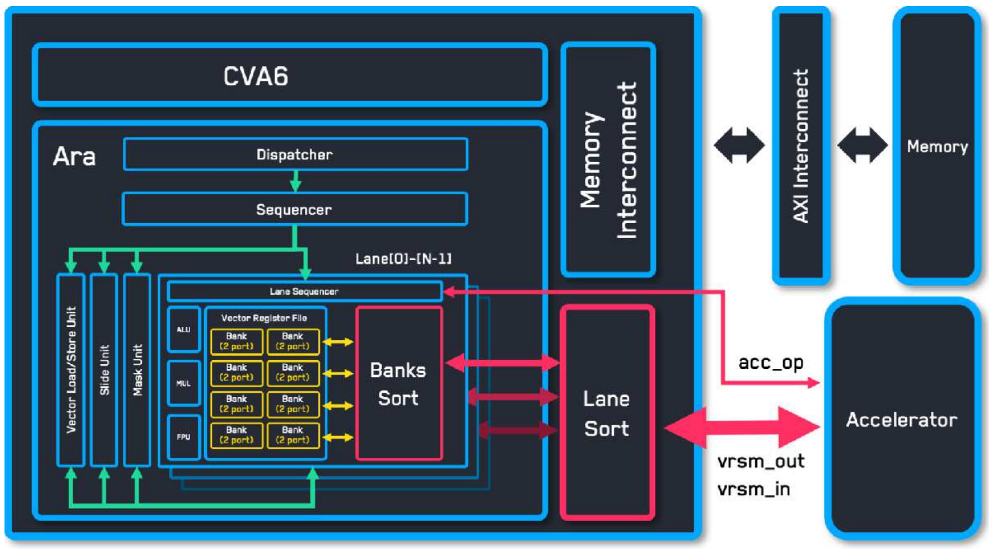

Fig. 5 The brief block diagram of the SHAVER implementation.

For this mechanism, signals are defined like TABLE 1. An accelerator usually needs some control signals to execute the task. These three signals are designed for communication between the shared vector registers and an accelerator.

Table 1 Signals for SHAVER.

name	description
acc_op	for controlling an accelerator
vrsm_out	for overwriting the vector registers to write through SHAVER
vrsm_in	for reading the vector registers to read through SHAVER

In the lanes of Ara, a dispatcher sends signals to each vector functional unit or bank according to an instruction. The 4-bit signal of *acc_op* is assigned by the dispatcher of Ara according to Table 2.

Table 2 acc_op.

binary value	description
0b0000	None
0b0001	Sending values to an accelerator
0b0010	Receiving values from an accelerator
(others)	Free (e.g. Starting to execute an accelerator)

The signals of *vrsm_out* and *vrsm_in* is connected with a part of banks on Ara. A bank is designed to be implemented with a single-ported SRAM in Ara. In SHAVER, a bank is accessed from an accelerator, not only Ara modules. Therefore, all banks should be designed as dual-port SRAM for processes on Ara and an accelerator. Dual-ported

SRAMs replace all banks through *XPM_MEMORY_TDPRAM* macro in Vivado [42]. The vector register file on Ara is shuffled to avoid conflicting register accesses. To use correct order values on an accelerator, the modules for sorting values of banks are implemented on a lane and a top module of Ara. Length of *vrsm_out* and *vrsm_in* is variable, but it depends on the VLEN of Ara and the number of vector registers shared with an accelerator. For example, each signal is 16384 bits if VLEN is 4096 bits and four-vector registers are shared.

An example of using SHAVER is shown by Listing 2.

Listing 2 Example of SHAVER (memory to an accelerator).

```c
int main(void){
  uint8_t source_array[512];
  uint8_t size_of_array = 512;
  ...
  // Sending RVV CSRs
  _asm_ volatile("vsetvli t0, %0, e8, m1, ta, ma" : : "r"(size_of_array));
  // Load from memory to vector register
  _asm_ volatile("vle8.v v8, (%0)" : : "r"(source_array));
  // Sending to an accelerator
  _asm_ volatile("vrsm.send v0, v0, v0");
  ...
}
```

First, the RVV parameters are set by *vsetvli* instruction. In Listing 2, RVV is set according to the value of *size_of_array*. After that, the values are loaded to a vector register from memory with *vle8* instruction. *vle8* is one of the load instructions of RVV. In this program, the loaded values are assigned at v8. For sending the value, *vrsm.send* instruction is defined as a custom instruction. Ara sets *acc_op* as *0b0001* when this instruction is called. *0b0001* shows an acceleration that the main core sends values to an acceleration.

4 Evaluation

4.1 Performance: only using an accelerator

In this evaluation, no processes are executed before or after the data transfer to know the SHAVER's data transfer performance. The convolution task is executed for evaluation. An accelerator is designed for a convolution process. The element length of the accelerator is 8 bits. The size of the window on convolution is 3x3. This accelerator can execute convolution that is constrained by 4096-bit rows. This evaluation measures the execution time when it changes the number of input rows from 512 rows to 8,192 rows.

VLEN is 4096 bits, and the number of lanes on Ara is 4. The width of data on the AXI interface is 128 bits. Ara does not contain data caches. So, when measuring SHAVER's performance, the performance with a data cache is estimated from the evaluation result on a simulator. Systems for measuring performance are implemented on Alveo U250 using Vivado 2021.2 [43]. Evaluation on a simulator executes on Verilator 4.214 [44].

The results of measurement are shown in Fig.6.

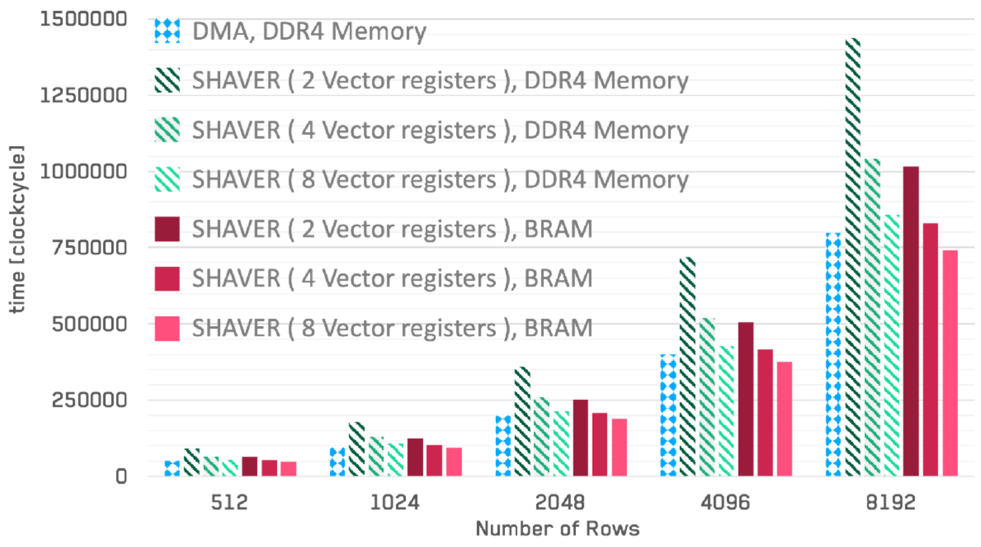

Fig. 6 Result of the data transfer performance evaluation.

Comparing the simulation results of SHAVER, when increasing the number of shared registers, the clock cycles for processing are reduced. Similarly, when SHAVER is run on an FPGA, the more registers are shared, the shorter the processing time. When comparing the outcomes of sharing two vector registers with those of sharing eight vector registers, a speedup of approximately 40% is observed. The reason for the 40% speedup is that the number of memory loads remains unchanged despite the number of shares being quadrupled. RVV instructions can load data to two, four, or eight vector registers simultaneously, reducing the memory overhead. These instructions are used differently in the programs of this evaluation, depending on the number of registers to be shared. These instructions allow the vector processor to execute load instructions with less overhead. As a result, up to 40% speedup can be achieved by increasing the number of instructions shared.

For all transfer methods of the evaluation, the number of clock cycles increased when running on the FPGA than on the simulation. The maximum increase is about 16%. Since DDR4 memory is used as the main memory on FPGA, operation on FPGA is slower than simulation. This is a latency-free memory access operation similar to a cache with no misses.

To compare SHAVER's use of cache with conventional methods such as DMA, we compare the simulation results of SHAVER with those of DMA running on FPGA using DDR4 memory. This comparison shows that our proposed SHAVER can perform the same process in up to 7.13% fewer clock cycles than DMA.

4.2 Performance: with a preprocessing

Fig. 7 shows us the data flow of each data transfer method.

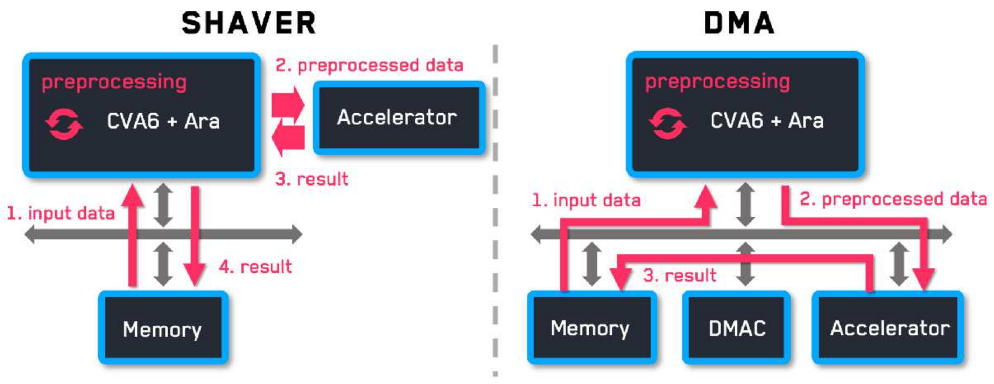

Fig. 7 The data flow of each data transfer method.

When using DMA, the bus is used for sending preprocessed data. A common DMA allows to execute in a main processor while transferring data.

On the other hand, when using SHAVER, the bus is used for getting data and sending the result. The accelerator communicates with CVA6 and Ara in SHAVER. It can transfer data between a main processor and an accelerator at once. The data preprocessed by the vector processor can be immediately transferred to the accelerator.

To know which data transfer method is faster, the padding process was applied to the data before sending it to the accelerator, and the total number of clock cycles required for the entire process was measured. The padding process is one of the common processes to keep data size after convolution. Fig.8 shows about the preprocessing.

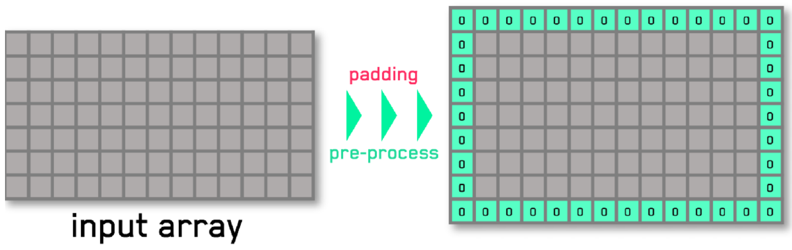

Fig. 8 The padding process is used as a preprocessing.

VLEN is 4096 bits, and the number of lanes on Ara is 4. The width of data on the AXI interface is 128 bits. The results of measurement are shown in Fig.9.

Compared with the result of DMA, SHAVER achieves 8.38% faster processing when using BRAM. In addition, when using DDR4, SHAVER achieves a 3.87% faster

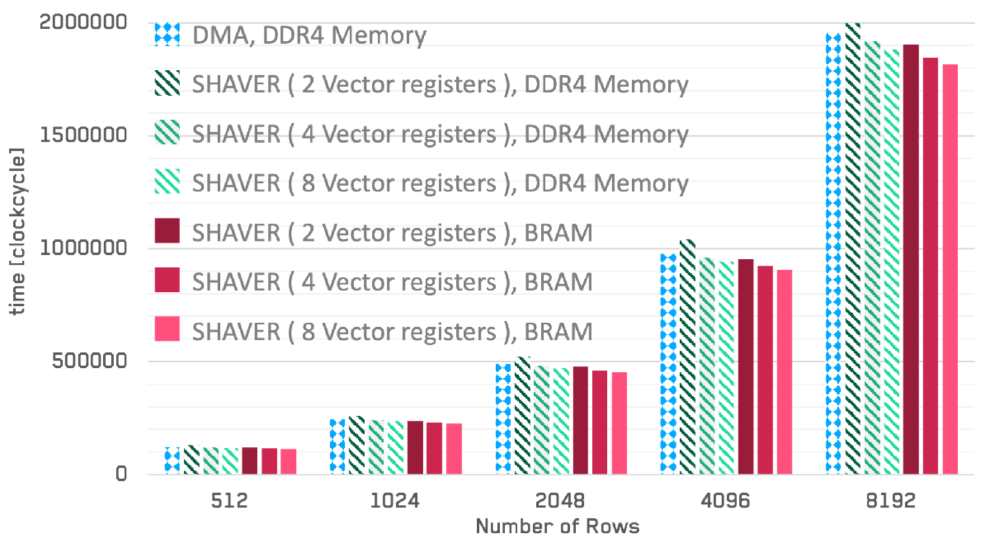

Fig. 9 Result of the performance evaluation when executing the preprocess.

processing rate. The result indicates that when all cache accesses hit, a maximum speedup of 8.38% can be achieved, and even when not all cache accesses hit, a speedup of approximately 3.87% can be expected.

4.3 Resource Usage and Maximum Operating Frequency

Resource usage and maximum operating frequency of each system are measured by Vivado 2021.2. The results are shown in Table 3.

Table 3 Resource Usage and Maximum Operating Frequency on Alveo U250.

Target System	LUT	FF	SRAM	DSP	Maximum Operating Frequency [MHz]
DMA	526,036	150,768	258	228	80.972
SHAVER: 1 VR	445,183	160,125	238	228	80.574
SHAVER: 2 VRs	447,440	172,425	366	228	81.182
SHAVER: 4 VRs	457,585	197,021	622	228	80.821
SHAVER: 8 VRs	462,695	246,039	1134	228	80.919

These results indicate that increasing the shared vector registers in SHAVER does not affect the maximum operating frequency. In addition, we found that the resource usage and maximum operating frequency of SHAVER do not change significantly compared to DMA transfers, except for SRAM. SRAM usage increases when the number of shared vector registers is increased. This increase in SRAM usage can be attributed to the increase in SRAM used for operations in the accelerator.

5 Conclusion

In this paper, SHAVER was proposed as a new data transfer method. This mechanism was implemented using Ara and CVA6. This implementation was run on FPGA and a simulator for evaluation. A DMA for comparison was also implemented and evaluated on Ara and CVA6.

In the evaluation to know the SHAVER of data transfer, we used an accelerator for convolutional operations and measured the number of clock cycles required. In addition, we compared SHAVER and DMA when adding the padding processing before using the convolution accelerator. The results indicate the potential of SHAVER to achieve a maximum of 8.38% speedup over DMA transfer.

The evaluation of resource usage and maximum operating frequency showed that SRAM utilization increases with the number of shared vector registers, depending on the accelerator's structure. The maximum operating frequency of the system using the SHAVER exhibited a minimal difference in comparison to the system using DMA transfer. Therefore, SHAVER does not significantly affect the maximum operating frequency.

6 Future Work

In the evaluation, we assessed SHAVER on a simulator, assuming no data cache misses. In future work, we plan to enhance the evaluation by incorporating caching into Ara, utilizing DDR4 memory as the main memory, and evaluating the performance of SHAVER. If a data cache is introduced in Ara, unlike the behavior observed in this simulation, penalties due to data cache misses are expected in SHAVER. However, considering temporal and spatial locality, the incurred latency is anticipated to be sufficiently low after excluding the initial miss penalty. Therefore, the performance evaluation results of SHAVER with a data cache in Ara are likely to be inferior compared to the behavior in the simulation, but the magnitude of the difference is expected to be minimal. On the other hand, attaching a cache to the DMA transfer system will invariably involve communication to maintain coherence with the data cache. Consequently, it can be anticipated that the performance evaluation results of the DMA system will also be inferior compared to the results of the current evaluation.

Furthermore, the accelerator used for the evaluation in this paper was exclusively designed for convolutional operations. In future work, we aim to diversify the types of accelerators and assess the performance of SHAVER across various tasks. Specifically, we intend to conduct a comparative evaluation between SHAVER and DMA using real-world applications. This will involve expanding the range of accelerators and exploring the effectiveness of SHAVER in diverse tasks.

Through the evaluation of resource utilization in this paper, it is speculated that the SRAM influenced the increase in resource usage within the accelerator. In other words, the resource utilization of our proposed mechanism depends on the length of the shared vector registers. In the future, it is necessary to discuss the structure of accelerators to remove the dependence.

In addition, we implement SHAVER for ASIC rather than FPGA. Some PDK tools are published as open-source software. In addition, some companies have started inexpensive chip fabrication services. According to them, chip fabrication services have expanded rapidly worldwide, not only in companies. Therefore, we aim to conduct measurements for not only resource utilization and maximum operating frequency on FPGA but also the impact of SHAVER on ASIC designed by an open-source EDA tool in the future. In particular, we anticipate significant differences between FPGA and ASIC in the dual-port SRAM used as the vector register banks. In future investigations, we aim to measure how these differences affect the maximum operating frequency and resource utilization.

Acknowledgements. This work is partially supported by JSPS Grant-in-Aid for Scientific Research (C) 24K14874.

References

[1] Team, A.C.V.M.L.: An On-device Deep Neural Network for Face Detection. https://machinelearning.apple.com/research/face-detection (Accessed: Dec 22, 2023)

[2] PAPER, X.M.P.W.: 4.14 AI Technology. https://trust.mi.com/docs/miui-privacy-white-paper-global/4/14 (Accessed: Dec 22, 2023)

[3] Microsoft: Copilot & Other AI-Powered Features. https://www.microsoft.com/en-us/windows/copilot-ai-features (Accessed: Sep 11, 2024)

[4] Github, I.: GitHub Copilot · Your AI pair programmer. https://github.com/features/copilot (Accessed: Dec 22, 2023)

[5] Jouppi, N., Kurian, G., Li, S., Ma, P., Nagarajan, R., Nai, L., Patil, N., Subramanian, S., Swing, A., Towles, B., Young, C., Zhou, X., Zhou, Z., Patterson, D.A.: Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In: Proceedings of the 50th Annual International Symposium on Computer Architecture. ISCA '23. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3579371.3589350

[6] Foundation, R.P.: Raspberry Pi Documentation — Processors — BCM2712. https://www.raspberrypi.com/documentation/computers/processors.html#bcm2712 (Accessed: Sep 14, 2024)

[7] Foundation, R.P.: RP2350. https://www.raspberrypi.com/products/rp2350/ (Accessed: Sep 14, 2024)

[8] Apple: iPhone 16. https://www.apple.com/iphone-16/ (Accessed: Sep 14, 2024)

[9] Apple: MacBook Pro. https://www.apple.com/macbook-pro/ (Accessed: Sep 14, 2024)

[10] technologies, I.: Intel® Core™ Ultra Processors Family. https://www.intel.com/content/www/us/en/products/details/processors/core-ultra.html (Accessed: Sep 14, 2024)

[11] Chen, X., Lei, Y., Lu, Z., Chen, S.: A variable-size fft hardware accelerator based on matrix transposition. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **26**(10), 1953–1966 (2018) https://doi.org/10.1109/TVLSI.2018.2846688

[12] Sipeed: Sipeed M1/M1W (Lichee Dan). https://wiki.sipeed.com/soft/maixpy/en/develop_kit_board/core_module.html (Accessed: Dec 22, 2023)

[13] Choi, K., Sobelman, G.E.: An efficient cnn accelerator for low-cost edge systems. ACM Trans. Embed. Comput. Syst. **21**(4) (2022) https://doi.org/10.1145/3539224

[14] Matsubara, K., Lieske, H., Kimura, M., Nakamura, A., Koike, M., Morikawa, S., Hotta, Y., Irita, T., Mochizuki, S., Hamasaki, H., Kamei, T.: A 12-nm autonomous driving processor with 60.4 tops, 13.8 tops/w cnn executed by task-separated asil d control. IEEE Journal of Solid-State Circuits **57**(1), 115–126 (2022) https://doi.org/10.1109/JSSC.2021.3120191

[15] Amirshahi, A., Klein, J.A.H., Ansaloni, G., Atienza, D.: Tic-sat: Tightly-coupled systolic accelerator for transformers. In: Proceedings of the 28th Asia and South Pacific Design Automation Conference. ASPDAC '23, pp. 657–663. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3566097.3567867

[16] arm: Primecell Micro DMA-230. https://www.arm.com/ja/products/silicon-ip-system/embedded-system-design/dma-230 (Accessed: Dec 22, 2023)

[17] Systems, E.: ESP32 Series Datasheet — v4.4. https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf (Accessed: Dec 22, 2023)

[18] NVIDIA Corporation: NVIDIA GPUDirect Storage Design Guide. https://docs.nvidia.com/gpudirect-storage/design-guide/index.html (Accessed: September 2, 2024)

[19] Advanced Micro Devices, I.: Teamwork builds achievement — AMD Infinity Fabric™ Link for workload sharing. https://www.amd.com/system/files/documents/ifl-solution-brief-radeon-pro-vii.pdf (Accessed: Sep 11, 2024)

[20] Zhao, L., Iyer, R., Makineni, S., Bhuyan, L., Newell, D.: Hardware support for bulk data movement in server platforms. In: 2005 International Conference on Computer Design, pp. 53–60 (2005). https://doi.org/10.1109/ICCD.2005.64

[21] Russell, R.M.: The CRAY-1 computer system. Commun. ACM **21**(1), 63–72

(1978) https://doi.org/10.1145/359327.359336

[22] Fujitsu: FACOM230-75 APU (1977). https://www.fujitsu.com/jp/about/plus/museum/products/computer/supercomputer/facom230-75apu.html (Accessed: January 22, 2024) (2023)

[23] Andrew Waterman, J.H. Krste Asanoviċ: The RISC-V Instruction Set Manual Volume II: Privileged Architecture Document Version 20211203. https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf (Published: Dec 4, 2021)

[24] Andes Technology Corporation.: AndesCore™ N22. https://www.andestech.com/en/products-solutions/andescore-processors/riscv-n22/ (Accessed: September 2, 2024)

[25] Andes Technology Corporation.: AndesCore™ AX65. https://www.andestech.com/en/products-solutions/andescore-processors/riscv-ax65/ (Accessed: September 2, 2024)

[26] Zhao, J., Korpan, B., Gonzalez, A., Asanovic, K.: SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine (2020)

[27] github — BOOM: The Berkeley Out-of-Order RISC-V Processor: riscv-boom. https://github.com/riscv-boom/riscv-boom (Accessed: December 22, 2023)

[28] github —The Rocket Chip Contributors: Rocket Chip: RISC-V Processor Core Generator. https://github.com/chipsalliance/rocket-chip (Accessed: September 2, 2024) (Accessed: 2024-09-13)

[29] github — Yosys Headquarters: picorv32. https://github.com/YosysHQ/picorv32 (Accessed: December 22, 2023)

[30] Asanoviċ, K.: Vector Extension 1.0, frozen for public review. https://github.com/riscv/riscv-v-spec/releases/tag/v1.0 (Accessed: Dec 22, 2023) (Published: Dec 20, 2021)

[31] Johns, M., Kazmierski, T.J.: A minimal risc-v vector processor for embedded systems. In: 2020 Forum for Specification and Design Languages (FDL), pp. 1–4 (2020). https://doi.org/10.1109/FDL50818.2020.9232940

[32] Abella, J., Bulla, C., Cabo, G., Cazorla, F.J., Cristal, A., Doblas, M., Figueras, R., González, A., Hernández, C., Hernández, C., Jiménez, V., Kosmidis, L., Kostalabros, V., Langarita, R., Leyva, N., López-Paradís, G., Marimon, J., Martínez, R., Mendoza, J., Moll, F., Moretó, M., Pavón, J., Ramírez, C., Ramírez, M.A., Rojas, C., Rubio, A., Ruiz, A., Sonmez, N., Soria, V., Terés, L., Unsal, O., Valero, M., Vargas, I., Villa, L., Ramíıez, C.: An academic risc-v silicon implementation

based on open-source components. In: 2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS), pp. 1–6 (2020). https://doi.org/10.1109/DCIS51330.2020.9268664

[33] SiFive, I.: SiFive Intelligence X280. https://www.sifive.com/\cores/intelligence-x280 (Accessed: Dec 22, 2023)

[34] Zaruba, F., Benini, L.: The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **27**(11), 2629–2640 (2019) https://doi.org/10.1109/TVLSI.2019.2926114

[35] github — OpenHW Group: cva6. https://github.com/openhwgroup/cva6 (Accessed: Dec 22, 2023)

[36] Perotti, M., Cavalcante, M., Wistoff, N., Andri, R., Cavigelli, L., Benini, L.: A “new ara” for vector computing: An open source highly efficient risc-v v 1.0 vector processor design. In: 2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 43–51 (2022). https://doi.org/10.1109/ASAP54787.2022.00017

[37] Cavalcante, M., Schuiki, F., Zaruba, F., Schaffner, M., Benini, L.: Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **28**(2), 530–543 (2020) https://doi.org/10.1109/TVLSI.2019.2950087

[38] Xilinx: Alveo U250 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html (Accessed: Dec 22, 2023)

[39] Xilinx: UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices (WP477). https://docs.xilinx.com/v/u/en-US/wp477-ultraram (Accessed: Dec 22, 2023)

[40] Xilinx: UltraScale Architecture Libraries Guide (UG974) — XPM_MEMORY_SPRAM. https://docs.xilinx.com/r/2021.2-English/ug974-vivado-ultrascale-libraries/XPM_MEMORY_SPRAM (Accessed: Dec 22, 2023)

[41] developer: AMBA AXI Protocol Specification. https://developer.arm.com/documentation/ihi0022/latest (Accessed: Dec 22, 2023)

[42] Xilinx: UltraScale Architecture Libraries Guide (UG974) — XPM_MEMORY_TDPRAM. https://docs.xilinx.com/r/2021.2-English/ug974-vivado-ultrascale-libraries/XPM_MEMORY_TDPRAM (Accessed: Dec 22, 2023)

[43] XILINX, A.: Adaptive Computing Support / Downloads / Vivado Archive / Vivado ML Edition - 2021.2 Full Product Installation. https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/archive.html (Accessed: Dec 22, 2023)

[44] github — verilator: verilator(tag:v4.214). https://github.com/verilator/verilator/tree/v4.214 (Accessed: Dec 22, 2023)