2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP) | 978-1-6654-8308-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ASAP54787.2022.00017

A "New Ara" for Vector Computing: An Open Source Highly Efficient RISC-V V 1.0 Vector Processor Design

Matteo Perotti
Integrated Systems Laboratory
ETH Zurich
Zurich, Switzerland
mperotti@iis.ee.ethz.ch

Matheus Cavalcante
Integrated Systems Laboratory
ETH Zurich
Zurich, Switzerland
matheusd@iis.ee.ethz.ch

Nils Wistoff
Integrated Systems Laboratory
ETH Zurich
Zurich, Switzerland
nwistoff@iis.ee.ethz.ch

Renzo Andri
Computing Systems Laboratory
Huawei Zurich Research Center
Zurich, Switzerland
renzo.andri@huawei.com

Lukas Cavigelli
Computing Systems Laboratory
Huawei Zurich Research Center
Zurich, Switzerland
lukas.cavigelli@huawei.com

Luca Benini
Integrated Systems Laboratory
ETH Zurich and University of Bologna
Zurich, Switzerland
lbenini@iis.ee.ethz.ch

***Abstract*—Vector architectures are gaining traction for highly efficient processing of data-parallel workloads, driven by all major ISAs (RISC-V, Arm, Intel), and boosted by landmark chips, like the Arm SVE-based Fujitsu A64FX, powering the TOP500 leader Fugaku. The RISC-V V extension has recently reached 1.0-Frozen status. Here, we present its first open-source implementation, discuss the new specification's impact on the micro-architecture of a lane-based design, and provide insights on performance-oriented design of coupled scalar-vector processors. Our system achieves comparable/better PPA than state-of-the-art vector engines that implement older RVV versions: 15% better area, 6% improved throughput, and FPU utilization >98.5% on crucial kernels.**

***Index Terms*—RISC-V, ISA, Vector, Efficiency**

I. Introduction

The top places of the current "Top500" list of the world's most powerful supercomputers can be divided into two groups: systems drawing their performance from accelerators and systems based on many-core processors with powerful vector units. The current leader is Fujitsu's Fugaku [1], [2] at the RIKEN research lab in Japan with 159 thousand nodes, each with a 48-core Fujitsu A64FX running at 2.2 GHz, supporting the Armv8.2-A Instruction Set Architecture (ISA) with Scalable Vector Extension (SVE) using 512 bit vectors. The Sunway TaihuLight [3] similarly builds on processors with vector extension—and while it is now ranked as number 4 (as of September 2021), it has been on the list since 2016. It has 26 thousand Sunway SW26010 CPUs, each chip running at 1.45 GHz with 4 clusters of 64 compute cores with a 256-bit vector unit each.

Vector processors find use not only in supercomputers. Arm's latest v9 ISA also includes the revised SVE2 vector extension [4] and is expected to become widely used in everyday devices such as smartphones and later on also make it into microcontrollers, real-time, and application processors.

The aforementioned ISAs are all proprietary, leaving their micro-architectural implications completely opaque and preventing open-source implementations. This is a significant obstacle to open innovation. Over the last few years, RISC-V has become a well-established, modern, and openly available alternative to proprietary ISAs. This has spurred a wave of publicly available implementations and enabled a public discourse on novel, custom ISA extensions as well as their micro-architectural implications and their potential standardization.

In this work, we focus on the RISC-V Vector Extension (RVV), initially proposed in 2015 [5]. Over the past six years, it has been intensively discussed, refined, and is now frozen and open for public review, version v1.0 [6]. The ratification of an extension is a key process for the RISC-V community, as both hardware and software will rely on a stable and standardized, but still open, ISA.

Vector instructions operate on variable-sized vectors, whose length and element size can be set at runtime. The architecture exploits the application parallelism through long, deeply pipelined datapaths and single instruction, multiple data (SIMD) computation in each functional unit. Moreover, a single vector instruction triggers the calculation of all the elements of a vector. This was initially introduced in Cray supercomputers and is sometimes called Cray-style vector processing. The benefits of this approach are a typically smaller code size and avoiding fetching and decoding the same instructions repeatedly, with a reduction in I-cache transfers and an increased system efficiency. Furthermore, it improves code portability. Today, Cray-like vector processors have gained new traction thanks to the race towards energy-efficient designs and the need

2160-052X/22/$31.00 ©2022 IEEE
DOI 10.1109/ASAP54787.2022.00017

Table I
OVERVIEW OF RISC-V VECTOR PROCESSORS

	Core Name	RVV version	Target	XLEN (bit)	float supp.	VLEN (bit)	Split VRF (lanes)	Open-Source
	This work	1.0	ASIC	64	✓	4096[a]	✓	✓
[8]	**SiFive X280**	1.0	ASIC	64	✓	512	?	✗
[9]	**SiFive P270**	1.0rc	ASIC	64	✓	256	?	✗
[10]	**Andes NX27V**	1.0	ASIC	64	✓	512	?	✗
[11]	**Atrevido 220**	1.0	ASIC	64	✓	128-4096	✓	✗
[12]	**Vicuna**	0.10	FPGA	32	✗	128-2048	✗	✓
[13]	**Arrow**	0.9	FPGA	32	✗	?	✓[c]	✗
[14]	**Johns et al.**	0.8	FPGA	32	✗	32	✗	✗
[15]	**Vitruvius**	0.7.1	ASIC	64	✓	16384	✓	✗
[16]	**XuanTie 910**	0.7.1	ASIC	64	✓	128[b]	✓	✗[d]
[17]	**RISC-V²**	?	ASIC	?	✗	256	✗	✓[e]
[7]	**Ara**	0.5	ASIC	64	✓	4096 [b]	✓	✓
[18]	**Hwacha**	Non-Std.	ASIC	64	✓	512[b]	✓	✓

[a] Parametric VLEN. In this work, we selected 4096. [b] VLEN per lane.
[c] VRF is split horizontally. [d] The vector unit is not open-source.
[e] The scalar core is not open-source.

for computing highly-parallel workloads. Moreover, thanks to their flexibility and straightforward programming model, they became a concrete alternative to Graphics Processing Units (GPUs) when operating on long vectors. Note that both the Cray-like and SIMD-style extensions are often referred to as vector extensions/instructions, e.g., Intel's Advanced Vector Extension (AVX) (introduced in 2011 and further developed into AVX2 in 2013 and into AVX-512 in 2016) and Arm's NEON extensions. However, these instructions with fixed-size operands are not vector extensions according to the definition above.

This work presents the first open-source implementation, including hardware and software, of the RVV 1.0 ISA, and makes the following contributions:

1) It describes a design of the RVV 1.0 vector unit as a tightly coupled extension unit for the open-source CVA6 RV64GC processor. It introduces a new interface and hardware memory coherency/consistency features.
2) It presents a comparison with a RVV 0.5 vector unit [7] and an analysis of the impact that the new RVV ISA has on microarchitectures that use lanes with a split vector register file.
3) It provides a power, performance, and area (PPA) evaluation of the unit implemented with GLOBALFOUNDRIES 22FDX Fully Depleted Silicon-on-Insulator (FD-SOI) technology, proving that the implementation achieves competitive PPA compared to the RVV 0.5 vector unit.
4) It analyzes the impact that the scalar processor has on the final achieved vector throughput, especially in the case of medium/short vectors.

II. RELATED WORK

The new RVV 1.0 vector unit is inspired by Ara [7], a vector unit implementing RVV v0.5. Ara is a highly-efficient vector coprocessor developed in 2019 that works in tandem with the application-class processor CVA6 (formerly Ariane) [19] and is compliant with one of the first and loosely defined RVV proposals [20]. Ara achieved 97 % peak utilization with 256×256 matrix multiplication with 16 lanes, 33 DP-GFLOPS of throughput, and 41 DP-GFLOPS/W of energy efficiency at more than 1 GHz in typical conditions in a 22 nm node.

In addition to Ara, we provide an overview of RISC-V processors that currently implement a vector extension in Table I. Notably, most of them are limited to a very short VLEN, which avoids many of the challenges, particularly those introduced in the RVV v1.0 specifications and addressed in this work. In terms of performance, the SiFive P270 is claimed to achieve 5.75 CoreMark/MHz, 3.25 DMIPS/MHz, and a 4.6 result on SPECint 2006 [9]. The XuanTie 910 obtains 7.1 CoreMark/MHz and 6.11/GHz on SPECint 2006. Even though detailed comparisons with other cores than Ara is not possible because they either do not comply with RVV (Hwacha) or they are not (fully) released, we can compare the available achieved FPU utilization during computational kernels: SiFive's X280 and Vicuna claim >90%, and Hwacha obtained >95%. Our new architecture reaches >98% of utilization and >35 DP-GFLOPS/W.

III. THE EVOLUTION OF RISC-V V

The RISC-V V extension enables the processing of multiple data using a single instruction, following the computational paradigm of the original Cray vector processor. This extension introduces 32 registers organized in a Vector Register File (VRF), with each register storing a set of data elements of the same type (e.g., FP32). Typical vector operations work element-wise on two vectors, i.e., on elements with the same index. In addition, predicated computation is supported, preventing the processing of some of the elements based on a Boolean mask vector.

The initial RVV extension was proposed in 2015 [5], with several updates presented by Krste Asanović and Roger Espasa [20]–[22] until 2018. Then, the specification started to be more officially maintained until today. Following the notation used in [7], we will refer to the last informal specification (2018) as v0.5. The current version of the specification is v1.0, and it is the frozen specification for public review. Together with V, it also describes various other extensions, like the ones targeting embedded processors: Zve. Throughout this work, we will only refer to the main extension for application processors. Even if the core concept of the RVV extension remained the same through time, there have been notable modifications: 1) the organization of the vector register file, 2) the encoding of the instructions, and 3) the organization of the mask registers. We will discuss these major changes in the following subsections.

A. Vector Register File

1) VRF state: The VRF is the most critical part in the design of a vector processor. It contains the vector elements, and its layout highly impacts the design choices. When the supported vector length is wide enough, it is usually implemented with Static Random-Access Memories (SRAMs), virtually creating another level in the memory hierarchy.

v0.5: The state of the register file was kept both globally and locally. The user dynamically specified how many registers

were enabled, and the hardware calculated the maximum vector length by dividing the byte space of the register file among all the enabled registers. Then, each register could be individually programmed to store a different data type.

v1.0: The state of the register file is only global. The vector register file is composed of 32 VLEN-bit vector registers, where VLEN is an implementation-dependent parameter and indicates the number of bits of a single vector register. It is possible to tune the parameter LMUL to change the granularity of the register file, e.g., setting LMUL to 2 means that the register file will be composed of sixteen $2 \times$ VLEN-bit vector registers. Moreover, the register file is agnostic on the data type of the stored elements.

2) Striping distance: The original proposal did not constrain the vector register file byte layout in a clear way. Later, the striping distance (SLEN) parameter was added to further specify how implementations could organize their internal vector register file byte layout. This parameter became lanes-friendly, especially in version 0.9 of the specifications.

v0.9: SLEN $\leq$ VLEN: each vector register is divided into a total of VLEN/SLEN sections with SLEN bits. Consecutive vector elements are mapped into consecutive sections, wrapping back around to the first section until the vector register is full [23].

v1.0: SLEN $=$ VLEN: the VRF is seen as a contiguous entity, and consecutive element bytes are stored in consecutive VRF bytes.

B. Instruction Encoding

v0.5: Since the data type of the vector elements was specified in a control register for each vector register, the instructions could use a polymorphic encoding, e.g., the instruction `vadd` would be used to add two vector registers, regardless of their data type.

v1.0: The encoding is monomorphic, and there are different instructions for different data types, i.e., integer, fixed-point, floating-point. The ISA has, therefore, more instructions, being one the longest extensions in the whole RISC-V environment.

C. Mask Register Layout

Mask bits are used to support predication, the way in which vector processors run conditional code. There is one mask bit per element in a vector, and the core executes the instruction on element i only if the i-th mask bit has a specific value.

v0.5: Only one vector register (`v1`) could host the mask vector. Each element of this vector could host one mask bit in its Least Significant Bit (LSB).

v1.0: Every register of the VRF can be a mask register, and the mask bits are sequentially packed one bit after each other, starting from the LSB of the vector register file.

IV. RISC-V V and Lanes

In this section, we discuss the impact that the RVV extension has on the microarchitecture. We will consider Ara as an example of a design tuned to RVV 0.5, even if the discussion is not limited to it. In the following, we refer to it as Vector Unit 0.5 (VU0.5), and to our new architecture as Vector Unit 1.0 (VU1.0).

VU1.0 is a flexible architecture with parametric VLEN developed to obtain high performance and efficiency on a vast range of vector lengths. For example, with VLEN $= 4096$, the unit can process vectors up to 4 KiB, when LMUL $= 8$, with a 16 KiB VRF. Pushing for high vector lengths has many advantages: operating on vectors that do not fit the VRF requires strip-mining with its related code overhead, which translates into higher bandwidth on the instruction memory side and more dynamic energy spent on decoding and starting the processing of the additional vector instructions.

A. VRF and Lanes

In VU0.5, the VRF was implemented by splitting it into chunks, one per lane. VU1.0 maintains the same lane-based VRF organization. In this section, we discuss an alternative form of re-implementing the VRF with a monolithic architecture, which would increase the routing complexity to/from each bank and the interconnect area, adding an additional dependency on the number of lanes. In general, the area of the VRF crossbar on the ports of the banks (A_{xbar}) is proportional to both the number of masters and slaves, as it requires `Masters` de-multiplexers and `Slaves` arbiters.

More in detail, in the lane-based organization, each lane contains a VRF section with of 8 1RW SRAM banks. All the masters of a lane (M_{lane}) connect to a bank (B), so the total interconnect area is the interconnect area of one lane multiplied by the number of lanes (ℓ),

$$A_{\text{xbar}}^{\text{split}} \propto M_{\text{lane}} \times \text{B}_{\text{lane}} \times \ell = M_{\text{lane}} \times 8 \times \ell, \tag{1}$$

while a monolithic VRF would connect each bank to every master of each lane,

$$A_{\text{xbar}}^{\text{mono}} \propto (M_{\text{lane}} \times \ell) \times B_{\text{tot}} = M_{\text{lane}} \times 8 \times \ell^2. \tag{2}$$

The quadratic dependency on the number of lanes would easily limit the potential scaling of a vector processor with a monolithic VRF. Moreover, a split VRF allows for additional freedom on the macro placement during the floorplanning of the vector unit and improved routability since the interconnect is local to the lane.

B. Byte Layout

During vector memory operations, the vector processor maps bytes from the memory to bytes in its vector register file. Following RVV 1.0, the memory and VRF layout must be the same, i.e., the i-th byte of the vector in memory is stored in byte i-th of the VRF. This condition cannot hold in the case of a split vector register file, as subsequent elements should be mapped to consecutive lanes to better exploit Data Level Parallelism (DLP) and not to complicate mixed-width operations. Since the element width can be changed and the mapping between elements and lanes remains constant, the one between bytes and lanes does not. Depending on the element width, the same byte is mapped to different lanes.

As a consequence, the processor must keep track of the element width of each vector register to be able to restore its content, and each unit that accesses a whole vector register must be able to remap its elements.

C. Shuffle/Deshuffle

The remapping is realized with shuffling (bytes to VRF) and deshuffling (bytes from VRF) circuits, which translate into a level of byte multiplexers, one per output byte. If N lanes operate on a 8 B datapath, and the units that access the whole VRF gather data from each lane in parallel to sustain the throughput requirements, $N \times 8$ bytes are shuffled/deshuffled every cycle, using $N \times 8$ multiplexers. Since an RVV unit should support four different element widths (8 bit, 16 bit, 32 bit and 64 bit), each multiplexer has four input bytes.

D. RVV 1.0 Implementation Challenges

Some of the changes introduced by RVV 1.0 simplify the interface to software programmers but complicate the design of a vector machine partitioned into lanes.

1) Mask Unit: When a vector operation is masked, the lane should not update the masked element bytes in the destination vector register. To do so, the information about the masked indices should be available within each lane. Due to the new mask register layout, every vector register can be used and read as a mask register, and because of the new mask vector layout, a lane can need mask bits stored in a different lane. Since data is shuffled within each register of the VRF, we need a unit (Mask Unit) capable of fetching and deshuffling the data with the knowledge of the previous encoding used for that register and then expanding and forwarding the masks to the correct lanes. The introduction of another unit that access all the lanes leads to a greater routing complexity, especially when scaling up the number of lanes, as already noticed in [7].

2) Reshuffle: The choice of constraining the VRF byte layout imposing SLEN = VLEN in architectures with lanes leads to peculiar issues when executing specific instruction patterns. Following the specifications, the architecture should also support the tail-undisturbed policy, i.e., the elements past the last active one should not be modified. When an instruction writes a vector register `vd` that was encoded with $\text{EEW}_{vd}^{\text{old}}$, with $\text{EEW}_{vd}^{\text{new}} \neq \text{EEW}_{vd}^{\text{old}}$, and the old content of the register is not fully overwritten, the previous data get corrupted since the byte mapping of `vd` is no more unique.

Not to corrupt tail elements, VU1.0 must deshuffle the destination register using $\text{EEW}_{vd}^{\text{old}}$ and reshuffle it back using $\text{EEW}_{vd}^{\text{new}}$. This operation is done by the slide unit since it can access all the lanes and already has the necessary logic to perform this operation (which is a `vslide` with null stride, and different EEW for the source and destination register). We called this operation *reshuffle*.

The issue is exacerbated by the fact that the program can change the vector length and element width at runtime, so it is not possible to know how many bytes need to be reshuffled unless both the vector length and the element widths are dynamically tracked for each vector register. Without this knowledge, the architecture must always reshuffle the whole register. In our architecture, the vector unit injects a reshuffle operation as soon as the front-end decodes an instruction that writes a vector register changing its encoded EEW. The reshuffle is executed before the offending instruction, and it is not injected if the instruction writes the whole vector register. In general, reshuffling hurts the Instructions Per Cycle (IPC) if the latency of the shuffle cannot be hidden and if this operation causes structural hazards on following slide instructions. A special case of instructions that suffer from this problem are the narrowing instructions, especially those that have the same source and destination registers. Without a renaming stage in the architecture, it is not possible to decouple source and destination registers, and a simple reshuffle operation is not enough to preserve the tail elements: when reshuffling to a lower EEW, bytes that belong to the same element are placed in different lanes, and cannot, therefore, be fetched anymore within one lane by the narrowing operation.

Using tail-agnostic policies presents some issues as well. The tail bytes must be either left unchanged or overwritten with ones. The bytes that are left unchanged get corrupted since the information about their mapping also gets lost, and writing all the tail elements to 1 has significant overhead, as these additional writes deteriorate the IPC and possibly generate new bank conflicts on the VRF.

Reshuffling is a costly operation, as the offending instruction always has a read-after-write (RAW) dependency on the reshuffle: the cost is even higher if the throughput of the slide unit is lower than the one of the computational unit, as chaining cannot proceed at full speed. The compiler can alleviate the problem by clustering the vector register file avoiding change in EEW in the same register as much as possible. On the hardware side, a renaming stage could also help in prioritizing the remapping of the destination registers to physical registers with the same EEW.

V. Architecture

Our unit supports the vast majority of RVV 1.0 with the following exceptions: no support for fixed-point arithmetic, floating-point reductions and very specific instructions (round towards odd, reciprocal, square root reciprocal), segment memory operations (non-mandatory since RVV 1.0), gather-compress, scalar moves (we emulate them through memory transfers), and some specific mask instructions (e.g., `vfirst`, `viota`). In Figure 1, we show the main block diagram of the system. The integration with the CVA6 processor is based on the RVV0.5 unit [7], with the following major enhancements.

a) Decoding: With the new RISC-V V specifications, the encoding of the vector instructions now fully specifies the data type of the vector elements on which the instruction operates. This allows moving most of the decoding logic and vector-specific Control and State Registers (CSRs) from CVA6 to the vector unit, making CVA6 more agnostic on the V extension. In the updated architecture, CVA6 keeps only the pre-decoding logic strictly needed to know 1) if the vector instruction is a vector instruction, to dispatch it to the vector unit when it

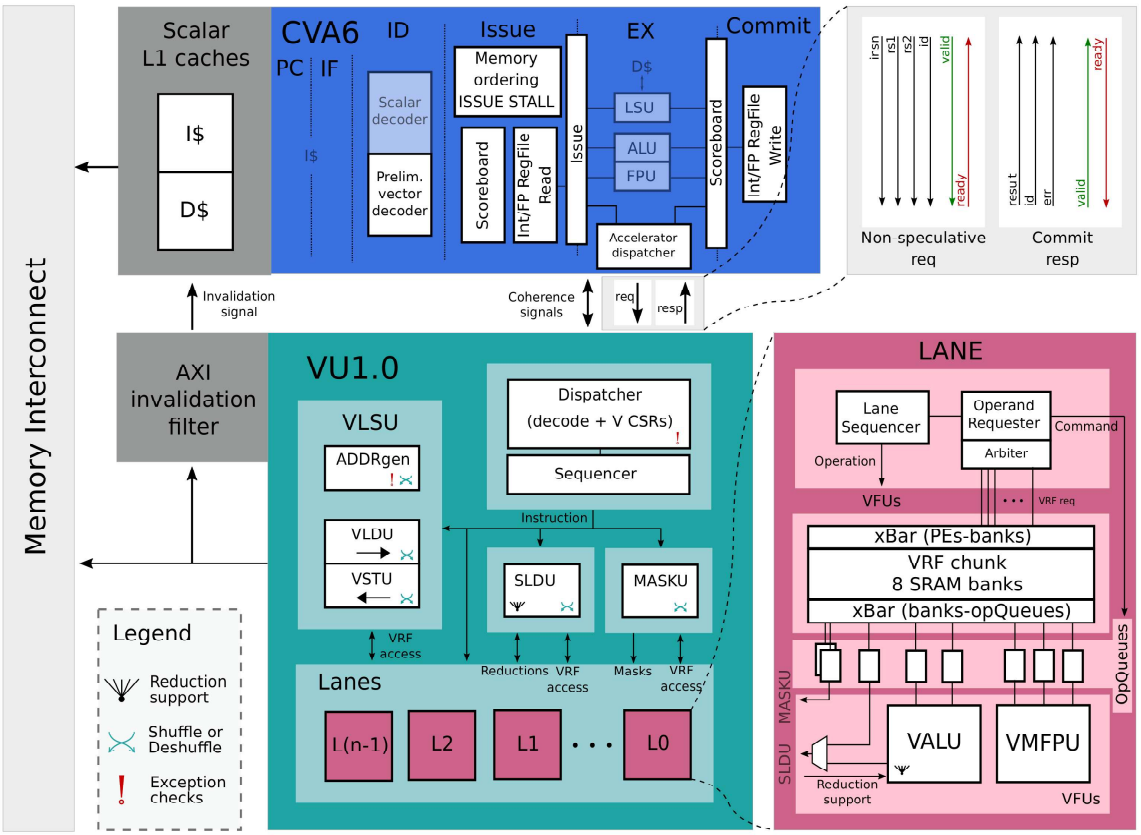

Figure 1. Top-Level block diagram of (the new) system with the vector co-processor marked in green, a more detailed diagram of the lane in magenta, and the host scalar core CVA6 in blue.

reaches the head of the scoreboard, 2) if the vector instruction is a memory operation (needed for cache coherency), and 3) if the instruction needs a scalar value from the integer or floating-point scalar register files.

b) CVA6-Vector Unit Interface: The interface between the host processor CVA6 and the vector unit is generalized: the unit is implemented as a modular accelerator with its own CSR file. While decoding, CVA6 identifies vector instructions, pushes them to a dispatcher queue, and dispatches them to the accelerator once they are no longer speculative.

c) Memory Coherency: CVA6 and the vector unit feature separate memory ports, and CVA6 has a private L1 data cache. At the same time, the RISC-V ISA mandates a strictly coherent memory view between the scalar and vector processors. VU0.5 [7] violates this requirement and needs explicit memory fences that write back and invalidate the CVA6 data cache between accesses on shared memory regions, adding a significant performance overhead and reducing code portability. In our VU1.0, we extend the system by a lightweight hardware mechanism to ensure coherency. We adapt the CVA6 L1 data cache to a write-through policy so that the main memory, which is accessed by the vector unit as well, is always up-to-date. When the vector unit performs a vector store, it invalidates the corresponding cache lines in the CVA6 data cache. Moreover, we issue 1) scalar loads only if no vector stores are in-flight, 2) scalar stores only if no vector loads or stores are in-flight, and 3) vector loads or stores only if no scalar stores are pending.

d) Mask Unit: After the update, mask bits are not always in the correct lane. Thus, we design a Mask Unit, which can access all the lanes at once to fetch, unpack and dispatch the correct mask bits to the corresponding lanes.

e) Reductions: Since our design has lanes, we implement integer reductions using a 3-steps algorithm: intra-lane, inter-lanes, and SIMD reduction steps. The intra-lane reduction fully exploits the data locality within each lane, maximizing the ALU utilization and efficiency, reducing all the elements already present in the lane. The inter-lanes reduction moves and reduces data among the lanes in $\log_2(\ell)+1$ steps, ℓ being the number of lanes, using the slide unit; since there is a dependency feedback between the slide and ALU units, the latency overhead of the communication is paid at every step.

Finally, the SIMD reduction reduces the SIMD word, if needed; therefore, its latency logarithmically depends on the element width.

VI. Results

One of the main motivations under the design of a vector processor is to efficiently maximize the throughput exploiting the intrinsic application DLP. In the following sections, we will use the word *throughput* to indicate the number of useful computational results produced per clock cycle, e.g., when adding two vectors of length N, the throughput is N/#cycles, where #cycles is the number of cycles required to produce the N results. With our experiment, we explore how the changes introduced by the updated V specification and the novel features of the system affect the throughput. Moreover, we place and route our design and extract the related PPA metrics.

A. Performance

We manually optimize the benchmarks used in [7] (fmatmul, fconv2d with 7×7×3 kernel), adapting them to the new specifications and architecture, and compile them using LLVM 13.0.0. To make a comparison between the two systems, we measure the cycle count of the same benchmarks of [7] with a cycle-accurate simulation of our vector unit using Verilator v4.214. We tune the benchmarks in assembly, as the LLVM compiled code showed inferior performance w.r.t. the optimized one.

To provide further insights on the system, we run the same experiment measuring how much the scalar core limits the final performance because of its non-ideal issue rate of vector instructions to the vector unit, showing how the scalar memory accesses impact the final throughput.

Figure 2 shows Ara's roofline model for a variable number of lanes, and the performance results of the matrix multiplication benchmark between square matrices $n \times n$ for several matrix sizes n. The arithmetic intensity for this benchmark is proportional to n. The horizontal dashed lines mark the computational limit of the architecture for the corresponding number of lanes. Originally, Ara achieved almost peak performance on the compute-bound `fmatmul` and `fconv2d` kernels, showing high levels of Floating Point Units (FPUs) utilization; this was also shown by Hwacha, with utilization above 95% [24]. Despite having a VRF 1/4 of the size of [7], our new architecture achieves comparable or better performance for long vectors both for `fmatmul` and `fconv2d`, with a peak of utilization of >98.5% with 2 lanes on 128×128 `fmatmul`. Since the utilization is almost 100%, an increase in VRF size would barely increase performance for larger problems.

In Figure 2, we show both the performance of the real system and that extracted with a perfect dispatcher. The perfect dispatcher is simulated by replacing CVA6 with a pre-filled queue with the corresponding vector instructions. The system with the ideal dispatcher shows the real performance limitations of the vector architecture and marks a hard limit on the performance achievable only by optimizing the scalar part of the system itself. The dotted black diagonal line is the limit

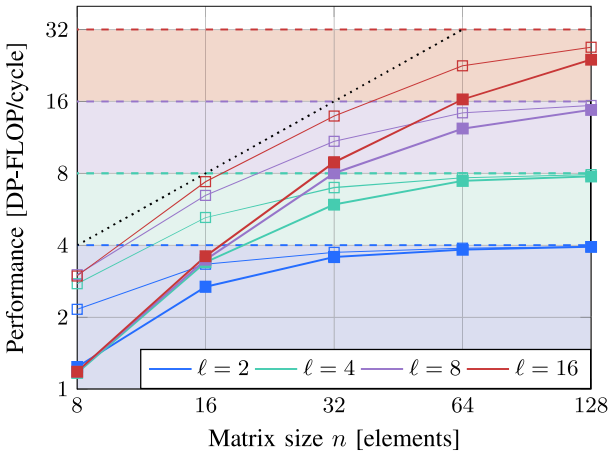

Figure 2. Runtime of matrix multiplication kernel of size $n \times n$ on our CVA6+Vector Unit system (■), compared with the ideal dispatcher (□), for several number of lanes ℓ.

given by the issue rate of computational vector instructions. In [7], the authors identify a hard limit to the performance of the matrix multiplication on the system, especially for short vectors. With RVV v0.5 and their algorithm, the issue rate of computational instructions for the main kernel is one instruction every five cycles. This was due to the presence of the `vins` instruction, used to move a scalar value from CVA6 to Ara. With the new specification, this is no longer needed since scalar operands can be passed with the vector multiply-accumulate instruction. This improves the computational instruction issue rate limit from 1/5 to 1/4, shifting the diagonal line to the left.

To study the impact that the scalar part of the system has on performance, we modify the Advanced eXtensible Interface (AXI) data width of the memory port of CVA6, as well as the D-cache line width, impacting the miss-penalty and miss-rate of the scalar data memory requests and, therefore, influencing the issue rate of the vector instructions of a kernel that, at every iteration, needs new scalar elements that are forwarded to the vector unit. In Figure 3, we summarize the throughput ideality of a 16 lane system executing `fmatmul` on a 16×16 matrix when varying the memory parameters of the scalar core. Increasing the cache line size decreases the miss rate, but if this comes without widening the AXI data width, the miss penalty is negatively influenced. The system throughput when both the cache line and AXI data width are 512-bit is 1.54× larger than when they are both set to 128 bits, showing the importance of the sizing of the scalar memory part of the system when improving the performance on medium/short vectors.

a) Short vectors: Even if the issue-rate limit is now improved for this kernel thanks to the new specification, short vectors' performance is lower than ideal. Our VRF does not implement the barber-pole VRF layout; so, the number of effective banks in each lane is reduced, and the system experiences more bank conflicts. For example, with 16 elements on 16 lanes, every element is stored in bank 0 in each lane, and every read/write operation will target the same bank. In general,

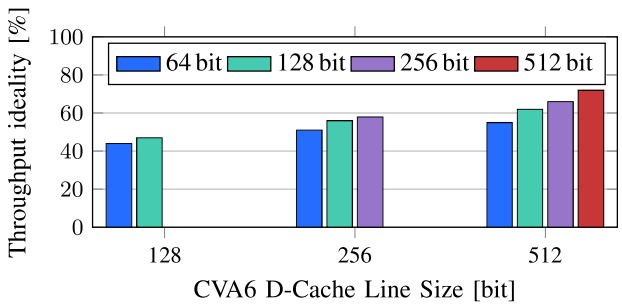

Figure 3. System throughput ideality relative to system with ideal dispatcher, as a function of CVA6's D-cache line size and AXI data width.

if there is not at least one element per bank (128 elements for 16 lanes, with VLEN $= 4096$), there will be more bank conflicts and related stalls. Barber-pole can mitigate this issue, as also with 16 elements in 16 lanes, elements of different registers occupy different banks. This is not a critical issue, as our vector architecture primarily targets long vectors; moreover, low performance on short vectors was already observed in Hwacha and Ara, even when implementing Barber-Pole layout in their VRF [7]. Designers can choose a lower vector length or opt for more efficient non-vector SIMD architectures.

b) Reductions: In Table II we report the performance and efficiency results from running the dot product kernel, varying the number of lanes, the vector byte length, and the element width. The measured cycle count only refers to the actual computational dot product, i.e., the vector-vector element-wise multiplication and the subsequent reduction, without the memory operations. Since our unit's multiplier and adder belong to different functional units, the product and reduction can be successfully chained so that the final cycle count scales only with the number of elements in the vector and not with the number of instructions (in this case, two). The efficiency is calculated on the ideal performance counted as $\mathrm{VL_B}/8\ell + 1 + \log_2(\ell)$, where $\mathrm{VL_B}$ is the vector length in bytes, and the added 1 keeps into account the chaining of the multiplication. 1) Longer vectors linearly increase the execution time of the intra-lane reduction. The longer the vector, the more this step hides the overhead of the other two, contributing to higher efficiencies. 2) The number of lanes linearly speeds up the intra-lane reduction phase and logarithmically negatively impacts the time spent during the inter-lanes reduction. When this phase is not hidden enough (e.g., short vectors), the overhead can be important w.r.t. the theoretical maximum achievable. The more the lanes, the *shorter* is the vector, in relative terms. To reach high levels of efficiency, a design with more lanes requires longer vectors. 3) Lower element widths positively influence the throughput, adding only a logarithmic overhead during the SIMD-reduction phase. The advantage over the scalar core is critical and can lead up to $380\times$ of performance improvement, especially for reduced element widths and long vectors, where the scalar cycle count skyrockets (>24k cycles peak): our vector unit exploits SIMD-like computation to keep a constant execution time while the element width is lowered, with a negligible overhead during the SIMD reduction phase.

For short vectors, the startup time of the vector operations is not amortized, and the overall efficiency drops. For example, our vector unit needs about ten cycles to practically produce results from the reduction after the vector multiplication is issued to the back-end.

Table II
CYCLE COUNT FOR REDUCTION OPERATION, WITH 2/16 LANES, VECTOR BYTE-LENGTHS, AND VECTOR ELEMENT SIZES. SLASHES DIVIDE RESULTS FOR DIFFERENT ELEMENT SIZES: 8-BIT ELEMENTS (LEFT), AND 64-BIT ELEMENTS (RIGHT).

	Cycle Count (#)			Efficiency (% on ideal)		
	64 B	**512 B**	**4096 B**	**64 B**	**512 B**	**4096 B**
2 Lanes	25 / 23	55 / 51	279 / 275	24% / 26%	62% / 67%	92% / 94%
16 Lanes	33 / 32	36 / 32	64 / 60	17% / 17%	25% / 28%	58% / 62%

Table III
PHYSICAL IMPLEMENTATION COMPARISON BETWEEN VU0.5 AND VU1.0

	VU0.5 System	**VU1.0 System**	**Update Merit**
VRF Size [KiB]	64	16	-75%
Die Area [mm^2]	0.98	0.81	-15%
Cell Area [mm^2]	0.43	0.49	+14%
Memory macro Area [mm^2]	-	0.15	-
Worst case frequency [MHz]	925	920	-0.5%
TT frequency [GHz]	1.25	1.34	+7.2%
Performance [DP-GFLOPS]	9.8	10.4	+6.1%
Power @TT frequency [mW]	259	280	-
Efficiency [DP-GFLOPS/W]	37.8	37.1	-1.9%

B. Physical Implementation

To assess the impact of our architectural modifications on the PPA metrics of the system, we synthesize and place-and-route our 4-lane enhanced design (CVA6, its caches, and the new vector unit) with VLEN $= 4096$ (16 KiB of VRF), targeting GLOBALFOUNDRIES 22FDX FD-SOI technology. The scalar I-cache and the D-cache have a line-width of 128 and 256 bits, respectively. We use Synopsys DC Compiler 2021.06 for synthesis with topographical information and Synopsys IC Compiler II 2021.06 for the physical implementation. We extract post-layout area and frequency, and, lastly, the related power results by means of activity-based power simulations with SDF back-annotation on typical conditions, elaborated using Mentor QuestaSim 2021.2 and Synopsys PrimeTime 2020.09, while executing a 128×128 `fmatmul`.

The design is placed and routed as a 0.81 mm×1.00 mm macro. To improve the process, we develop an implementation flow that leverages the modularity of the design. Our vector unit is composed of a parametric number of lanes that contain most of the processing logic of the system. All the lanes are identical, and their synthesis requires automatic retiming of the pipeline stages of the FPU. We use a hierarchical synthesis and back-end flow, in which the lanes are designed as custom macros and synthesized once to reduce turn-around time significantly.

In Figure 4 and Figure 5, we present the physical layout of the whole system and a single lane, respectively. At the top

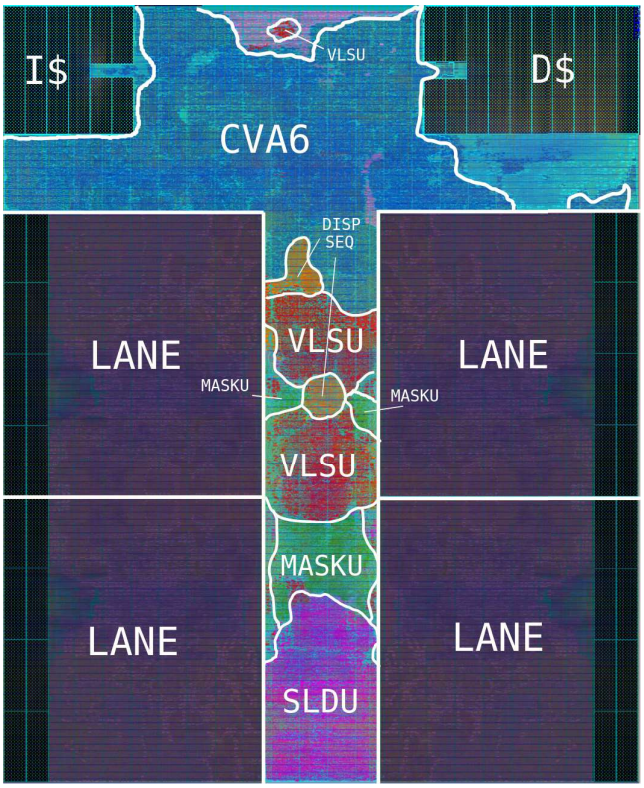

Figure 4. Physical implementation of the full system. The lane is implemented and enclosed in a macro and then placed on the die. The system input and output are at the top of the die (AXI interface).

of the die, we have the AXI interface, around which there are the AXI crossbar, CVA6, and part of the Vector Load/Store Unit (VLSU). The lanes are on the two sides to ease the routing of all the cross-lane units that access them, like the Mask Unit (MASKU), the Slide Unit (SLDU), the VLSU, and the main sequencer. The lane area is dominated by the Vector Multiplier/Floating Point Unit (VMFPU), the module that contains the FPU and the SIMD multipliers.

Table III shows the parameters and the quality figures of our implementation, with respect to VU0.5. Since, as shown in Section VI-A, VU1.0 achieves a competitive throughput despite using a VRF $4\times$ smaller than VU0.5's, we obtain an overall area reduction larger than 15 % on the die size without trading off performance. VU1.0 achieves a frequency of 920 MHz in worst-case conditions (SS, 0.72 V, 125 °C), virtually the same as in VU0.5. VU1.0 achieves 1.34 GHz in typical conditions (TT, 0.80 V, 25 °C), 7.2 % faster than what is reported in [7] for VU0.5, thanks to an advanced hierarchical implementation strategy. This translates into a `fmatmul` peak throughput 6.1 % higher than VU0.5 (10.4 DP-GFLOPS).

VU1.0 consumes 280 mW while running a 128×128 `fmatmul`, which leads to an energy efficiency of 37.1 DP-GFLOPS/W. This is only 1.9 % lower than VU0.5, while supporting a much more complete ISA, including reductions and memory coherence support. In [18], the measured efficiency

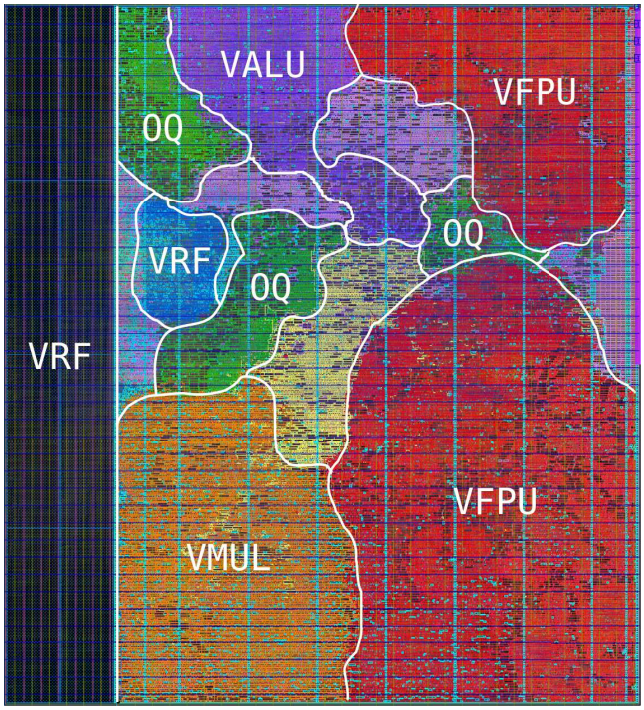

Figure 5. Physical implementation of a Lane. Modules without a label: lane sequencer, operand requesters (close to the VRF), VDIV and control logic for VMUL and VFPU (in the middle).

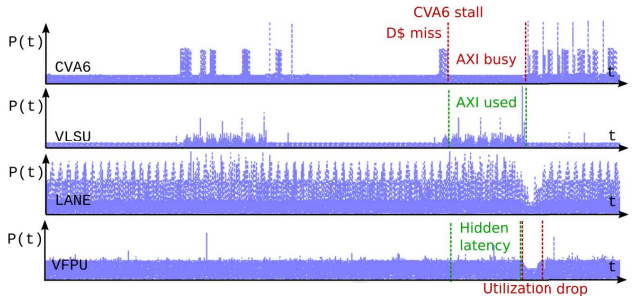

Figure 6. Time-based power simulation - two iterations of `fmatmul`. VU1.0 performs two vector loads, whose latency is hidden by the computation in its VFPUs (with utilization around 97%). Towards the end of the time span, CVA6 is temporarily stalled because of an L1D-cache miss that cannot be served by the upper memory system, which is already used by VU1.0's VLSU. To maximize visibility, every sub-plot is scaled to its maximum power consumption.

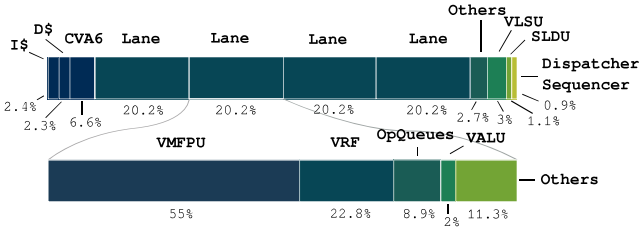

Figure 7. Power breakdown of the system. The overall average power consumption is 280 mW at 1.34 GHz when executing `fmatmul`.

for comparable voltage (0.8 V) or frequency (1.34 GHz) is lower than 33 DP-GFLOPS/W for a real system running Hwacha 4. VU1.0's efficiency is also much higher than the one reported for Hwacha 4.5 [25], although a direct comparison is not possible, as they only report the power measurements of the full manufactured system.

Figure 6 shows the power traces of two iterations of a `fmatmul` run, highlighting how the vector architecture can tolerate stalls on the scalar core. The vector unit and CVA6 compete for the AXI port to the L2 memory, and, during the second iteration, CVA6 stalls because of an L1 D-cache miss that cannot be served since a vector load is ongoing. During the interval in which CVA6 cannot forward new vector instructions, the vector unit does not starve until it has processed all the elements of the vector in its VRF, and the overall utilization remains at its peak for most of the time.

The average power breakdown of the system running `fmatmul` is shown in Figure 7. More than 80% of the power is consumed by the lanes, where the computation takes place. As expected, the VMFPU uses most of the energy of a lane and, together with the VRF and the operand queues, accounts for almost 90% of the total budget of a lane. This highlights how VU1.0 exploits the regular execution pattern of vector machines, leading to smaller and simpler controller logic. In comparison, CVA6 and its scalar caches consume less than 12% of the total power budget in spite of the significant area footprint of the scoreboard and instruction dispatch logic [26].

VII. Conclusions

In this paper, we present the first open-source vector processor compliant with the core of RVV 1.0. We compare our design with a RVV 0.5 unit and discuss the impact that the specification update has on architectures with a VRF split among lanes, the newly added features. We show competitive throughput and PPA results using the advanced GlobalFoundries 22FDX technology. The system runs at up to 1.34 GHz, with peak FPU utilization >98% on crucial matmul kernels. We provide insights on the performance of the mixed scalar-vector system for short vectors and an analysis of the new reduction engine that leads to speedups up to 380× with respect to a scalar core. The source code is released on https://github.com/pulp-platform/ara.

Acknowledgment

This work was supported by the ETH Future Computing Laboratory (EFCL), financed by a gift from Huawei Technologies.

References

[1] M. Sato, Y. Ishikawa *et al.*, "Co-design for A64FX manycore processor and fugaku," in *Proc. IEEE SC'20*, 2020.

[2] J. Dongarra, "Report on the Fujitsu Fugaku system," *University of Tennessee-Knoxville Innovative Computing Laboratory, Tech. Rep. ICLUT-20-06*, 2020.

[3] J. Gao *et al.*, "Sunway supercomputer architecture towards exascale computing: analysis and practice," *Science China Information Sciences*, vol. 64, no. 4, 2021.

[4] Arm, *Arm A64 Instruction Set Architecture - Armv9, for Armv9-A architecture profile*, 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0602/2021-12

[5] C. Schmidt *et al.*, *RISC-V Vector Extension Proposal*, 2015. [Online]. Available: https://riscv.org/wp-content/uploads/2015/06/riscv-vector-workshop-june2015.pdf

[6] K. Asanovic *et al.*, *Vector Extension 1.0*, 2021. [Online]. Available: https://github.com/riscv/riscv-v-spec/releases/tag/v1.0

[7] M. Cavalcante *et al.*, "Ara: A 1-GHz+ scalable and energy-efficient RISC-V vector processor with multiprecision floating-point support in 22-nm FD-SOI," *IEEE TVLSI*, vol. 28, no. 2, pp. 530–543, 2020.

[8] SiFive, "SiFive intelligence x280," accessed: 2021-11-20. [Online]. Available: https://www.sifive.com/cores/intelligence-x280

[9] ——, "SiFive performance P270," accessed: 2021-11-20. [Online]. Available: https://www.sifive.com/cores/performance-p270

[10] M. Demler, "Andes plots RISC-V vector heading, NX27V CPU supports up to 512-bit operations," May 2020, accessed: 2021-11-20. [Online]. Available: andestech.com/wp-content/uploads/Andes-Plots-RISC-V-Vector-Heading.pdf

[11] R. Espasa, *Introducing SemiDynamics High Bandwidth RISC-V IP Cores*, 2021. [Online]. Available: https://www.european-processor-initiative.eu/wp-content/uploads/2021/03/202012.RISCV-SUMMIT.pdf

[12] M. Platzer and P. Puschner, "Vicuna: A timing-predictable RISC-V vector coprocessor for scalable parallel computation," in *Proc. ECRTS'21)*, 2021.

[13] I. A. Assir *et al.*, "Arrow: A RISC-V vector accelerator for machine learning inference," *arXiv preprint arXiv:2107.07169*, 2021.

[14] M. Johns and T. J. Kazmierski, "A minimal RISC-V vector processor for embedded systems," in *Proc. IEEE FDL*, 2020.

[15] F. Minervini and O. P. Perez, *Vitruvius: And Area-Efficient RISC-V Decoupled Vector Accelerator for High Performance Computing*, 2021.

[16] C. Chen, X. Xiang *et al.*, "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 *Proc. ACM/IEEE ISCA*, 2020, pp. 52–64.

[17] K. Patsidis, C. Nicopoulos, G. C. Sirakoulis, and G. Dimitrakopoulos, "RISC-V^2: A scalable RISC-V vector processor," in *Proc. IEEE ISCAS*, 2020.

[18] C. Schmidt, J. Wright, Z. Wang, E. Chang, A. Ou, W. Bae, S. Huang, V. Milovanović, A. Flynn, B. Richards *et al.*, "An eight-core 1.44-GHz RISC-V vector processor in 16-nm FinFET," *IEEE Journal of Solid-State Circuits*, vol. 57, no. 1, pp. 140–152, 2021.

[19] F. Zaruba *et al.*, "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 TVLSI*, vol. 27, no. 11, 2019.

[20] K. Asanovic *et al.*, *RISC-V Vector Extension Proposal*, 2018. [Online]. Available: https://riscv.org/wp-content/uploads/2018/05/15.20-15.55-18.05.06.VEXT-bcn-v1.pdf

[21] Krste Asanovic, *Vector Extension Proposal v0.2*, 2016. [Online]. Available: https://riscv.org/wp-content/uploads/2016/12/Wed0930-RISC-V-Vectors-Asanovic-UC-Berkeley-SiFive.pdf

[22] K. Asanovic *et al.*, *The RISC-V Vector ISA*, 2017. [Online]. Available: https://riscv.org/wp-content/uploads/2017/12/Wed-1330-RISCVRogerEspasaVEXT-v4.pdf

[23] ——, *RISC-V Vector Extension Proposal*, 2020. [Online]. Available: https://github.com/riscv/riscv-v-spec/releases/tag/0.9

[24] C. Schmidt, A. Ou, and K. Asanovic, *Hwacha V4: Decoupled Data Parallel Custom Extension*, 2018.

[25] A. Gonzalez *et al.*, "A 16mm2 106.1 GOPS/W heterogeneous RISC-V multi-core multi-accelerator SoC in low-power 22nm FinFET," in *ESSCIRC 2021*, 2021.

[26] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Snitch: A 10 kGE pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads," Feb. 2020, arXiv:2002.10143v1 [cs.AR].