Extending the NOEL-V Platform with a RISC-V Vector Processor for Space Applications

Mascio, Stefano Di; Menicucci, Alessandra; Gill, Eberhard; Monteleone, Claudio

DOI
10.2514/1.I011097

Publication date
2023

Document Version
Final published version

Published in
Journal of Aerospace Information Systems

Journal of Aerospace Information Systems
Vol. 20, No. 9, September 2023

Extending the NOEL-V Platform with a RISC-V Vector Processor for Space Applications

Stefano Di Mascio,* Alessandra Menicucci,† and Eberhard Gill‡
Delft University of Technology, 2629 HS Delft, The Netherlands
and
Claudio Monteleone§
European Space Agency, 2200 AG Noordwijk, The Netherlands

https://doi.org/10.2514/1.I011097

This paper describes the work carried out to extend the NOEL-V platform to include data-level parallelism (DLP) by implementing an integer subset of the RISC-V Vector Extension. The performance and resource utilization efficiency of the resulting vector processor for different levels of DLP (i.e., number of lanes) have been compared to the baseline scalar processor on a Xilinx Kintex Ultrascale field-programmable gate array, employing typical kernels for compute-intensive applications. The role of the memory subsystem has also been investigated, comparing the results obtained with a low-latency and a high-latency main memory. The results show that the speed-up due to the use of the vector pipeline increases with the number of lanes in the vector processor, achieving up to 23.0× the performance of the scalar processor with only 4.3× the resources of the baseline scalar processor. Using an implementation with 32 lanes increases performance even for problem sizes larger than the number of lanes, achieving up to more than 11.7× the performance of the scalar processor with just 1.9× its resource utilization for 128 × 128 matrix multiplications. This work proves that implementations of the selected subset are easily scalable and fit for small-processor implementations in highly constrained space embedded systems.

I. Introduction

STATE-OF-THE-ART processors for space embedded systems are based on simple microarchitectures, for instance, capable of executing a maximum of one instruction per clock cycle (CC), with simple or no speculation (e.g., static branch prediction), and without data-level parallelism (DLP) [1].

Recently, the HPP64 NOEL-V [2], a synthesizable Very High Speed Integrated Circuit Hardware Description Language (VHDL) model of a processor targeting space applications and based on the RISC-V Instruction Set Architecture (ISA), has been released open source to the public together with the GRLIB VHDL Intellectual Property (IP) Core Library.¶ As opposed to its predecessors, it has a dual-issue pipeline; i.e., it is capable of executing up to two instructions per CC [3]. Also, it has a two-level adaptive branch predictor with history buffer, a two-way branch target buffer, and a return-address-stack [3]. Furthermore, late arithmetic logic units (ALUs) and a late branch unit are implemented to allow ALU and branch operations normally done in the execution stage to be deferred to a later stage. This reduces the latency required to execute some instructions once they are fetched, as well as allowing some dependent instructions to be issued simultaneously [3].

These new features have pushed the CoreMark for a single core [4] from around 2 CoreMark/MHz of previous generations [5] to more than 4 CoreMark/MHz [2]. However, the limited amount of functional units (4 ALUs and one multiplier/divider) sets an upper bound on the achievable performance for workloads performing a large amount of operations, e.g., matrix multiplications.

Although this limitation was considered acceptable in previous generations of space processors, nowadays compute-intensive workloads are becoming of interest also in space applications. For instance, the use of deep neural networks (DNNs) inference and other machine learning algorithms is being proposed in space applications [6], to increase the capabilities [7] and dependability [8] of satellites. As shown in [9] and [10], a large part of these algorithms can be mapped into matrix multiplications, which are very compute-intensive workloads and can be sped up effectively by DLP. In [11] it was found that this is especially true for inference of convolutional networks, while DLP is less effective for other machine learning algorithms (e.g., recurrent neural networks for satellite telemetry forecasting or unsupervised learning). On the other hand, training of convolutional networks is composed of calculations similar to those of the inference [12]. However in this case a computation of the same complexity of the inference has to be performed many times until the training is complete. For this reason, an additional increase of performance to achieve training in reasonable time scales is required compared to hardware platforms used for inference [13].

One of the main reasons behind the gap between microarchitectures employed in space and terrestrial applications is that it is not possible to reuse in a straightforward way the hardware platforms employed in terrestrial applications, given the specific constraints of satellite data systems especially in terms of robustness to ionizing radiation [9].

Ionizing particles can change the value stored in sequential elements, causing single event upsets (SEUs) and single event functional interrupts (SEFIs) [14]. Even worse, they can cause permanent damage with single event latch-ups (SELs) [15]. In [9], it was shown that a graphics processing unit (GPU) is expected to fail almost three orders of magnitude more often than a state-of-the-art space processor because of its larger area. Given that the failure rate of a processor is proportional to its area, in this work the focus will be on area-efficient data parallel processors. However, as the failure rate due to ionizing radiation depends also on the radiation environment, the applicability of this type of processors in different orbits (e.g., low Earth orbit, geostationary orbit, or deep space) will require a tailored radiation tolerance approach, including radiation testing, redundancy, shielding, and operations (e.g., periodic power cycling and memory scrubbing) [16].

Received 3 January 2022; revision received 9 April 2023; accepted for publication 16 April 2023; published online Open Access 30 May 2023.

*Ph.D., Faculty of Aerospace Engineering, Space Systems Engineering; s.dimascio@tudelft.nl (Corresponding Author).

†Associate Professor, Faculty of Aerospace Engineering, Space Systems Engineering; a.menicucci@tudelft.nl.

‡Professor, Faculty of Aerospace Engineering, Space Systems Engineering; e.k.a.gill@tudelft.nl.

§On-Board Computer Engineer, Microelectronics and Data Systems Division, European Space Technology Centre, Keplerlaan 1; claudio.monteleone@esa.int.

¶https://www.gaisler.com/index.php/downloads/leongrlib.

A. State-of-the-Art of Data Parallel Processors

Processors for terrestrial applications typically address compute-intensive applications with ISA extensions employing a type of DLP called packed single instruction multiple data (SIMD), both for personal computers (Intel's MMX extensions for integers [17] and the Streaming SIMD Extensions for floats [18]) and embedded applications (NEON extension [19]). Also PULP, one of the most popular sets of RISC-V cores, employs the RI5CY packed SIMD extension, defined outside of the RISC-V standard [20]. Packed SIMD extensions are typically chosen by hardware designers because they can be implemented without extensive modifications to the microarchitecture of the baseline processor [21]. However, with the end of Moore's law, computer architects are starting to consider more efficient ISA extensions for DLP. For instance, ARM recently released its ARMv8-A Scalable Vector Extension (SVE, 2017) [22]. Although previous Fujitsu's supercomputers were based on SIMD extensions of the SPARC ISA, the Fujitsu A64FX is the first processor based on the ARMv8-A SVE, targeting supercomputer applications [23]. Vector extensions can be seen as more flexible versions of packed SIMD extensions thanks to their time-multiplexed and vector-length-agnostic (VLA) approach. For VLA ISAs, the software is not required to know the hardware vector length of a specific implementation and the code can be written to run the same executable with the largest parallelism available on every platform without any modification [21,22,24]. In SIMD extensions the data width of the operations are encoded directly in the instruction opcodes instead. Therefore, the code must target a specific width. Furthermore, when the architects of SIMD ISAs want to increase performance by widening the vectors, they must add a new set of instructions to process these vectors [21]. Therefore, application code compiled for previous versions of the ISA cannot automatically leverage the widened vectors of new implementations and the code compiled for wider SIMD extensions fails to execute on older machines (as the new instructions are not known to older implementations) [21].

The RISC-V Vector Extension (RVVE) [25] is similar to the ARMv8-A SVE and was inspired by the Hwacha development [26]. It defines a configurable vector unit with 32 vector registers, where the number of elements and size of elements can be configured with configuration instructions [25]. Its configurability and VLA approach allow the same binary code to work efficiently across a variety of hardware implementations, varying in physical vector storage capacity and datapath parallelism [25].

B. Goal and Methodology

Although vector processors are often seen as processors fit for power-hungry high-performance applications [26–28], this paper aims to show that vector processors can be employed efficiently in highly constrained embedded systems as well (space embedded systems are an extreme example of this type of systems). To do this, an integer subset of the RVVE was implemented in synthesizable VHDL, comprising an immediate configuration instruction (*vsetvli*), a unit-stride load instruction (*vle32.v*), a unit-stride store instruction (*vse32.v*), a vector-vector addition instruction (*vadd.vv*), a whole vector register move instruction (*vmv1r.v*), a multiplication-accumulation on the addend (*vmacc.vx*), and a multiplication-addition overwriting the multiplicand (*vmadd.vx*). The use of only integer instructions is common for implementations targeting applications where the precision of the floating-point formats is not needed, and fixed-point formats can be employed, resulting in processors with small footprints for low-power applications [8,20].

The performance of the highly optimized dual-issue scalar pipeline of the HPP64 (configured to support the RV64IM subset and referred to as *scalar* processor) will be compared to the performance of an implementation where the scalar processor is extended with a simple and modular vector processing unit (VPU).[**] To do this, a scalar C function was executed on the scalar processor and then an analogous function, coded using intrinsic functions [29] to generate vector instructions, was executed on the vector processor. The executables were generated with a patched version of the RISC-V toolchain, supporting intrinsic functions for generation of RVVE instructions,[††] as (at the time of writing) the RVVE has not been frozen yet and vector instructions are not supported yet by the standard toolchain. The use of intrinsics allows considering only a subset of the RVVE, as the generated instructions depend on the intrinsic functions employed [30]. For instance, the intrinsic function

$$\text{size\_t vsetvl\_e32m1 (size\_t avl)}$$

generates a *vsetvli* instruction with a selected element width (SEW) of $32b$ (*e32*) and no register grouping (*m1*), having in the field *rs1* the register address of the register storing the variable *avl* indicating the application vector length (AVL), i.e., the number of elements in the vector defined at software level. The function returns a value in the register indicated in the field *rd*, which can be implicitly selected by the user by assigning the result of the intrinsic function to a variable. Examples of use of this intrinsic function are reported in Sec. III. Both for the scalar and the intrinsic functions the code obtained from the compiler was employed, without hand optimizations at assembly level.

II. Implementation of the Vector Processing Unit

Figure 1 shows the complete vector processor, with a modified HPP64 interfaced with the VPU. The VPU comprises three elements: a sequencer, the lanes, and a vector load and store unit (VLSU). The sequencer interfaces with the modified pipeline of the NOEL-V, receiving scalar parameters (e.g., base addresses of vectors) and vector instructions, while sending to the scalar pipeline the scalar results (when applicable). Each lane consists of a slice of the vector register file (VRF), containing a single 32-bit element of the vector for each of the 32 vector registers defined by the RVVE, and the relative combinational paths to execute the implemented RVVE instructions. The *vmadd* and *vmacc* instructions are executed in two CCs (pipelining) to avoid penalties on the maximum frequency compared to the scalar processor when using large amounts of DLP required for DNNs (as shown analyzing DNNs in [9]). For maximum flexibility of the prototype and quick design exploration, the VLSU uses a generic bus master from the GRLIB to read and write data on a 128-bit-wide AMBA high-performance bus (AHB). Both the sequencer and the VLSU are designed to handle a configurable number of lanes N_{Lanes} that can be set at compile time with a *generic* in VHDL. To decide which range of number of lanes to consider in this study, we carried out a study of DNNs in [9]. For the convolutional layers of the DNN analyzed in [9], only 18.2% (24 out of 132) of the dimensions of the multiplied matrices are smaller than 128. For this reason, we will focus on improving performance of kernels with 128-element parallelism, as this seems a good tiling size to speed up a large majority of matrix multiplications in DNNs. Therefore, we will only consider configurations with $N_{\text{Lanes}} \leq 128$, as configurations with a level of DLP larger than allowed by the algorithm do not provide any advantage (unless software solutions like batching are employed [10]).

Furthermore, the integration of the VPU required slight modifications to the scalar baseline HPP64 NOEL-V processor. The instruction decode stage (D) was modified to access the required scalar registers for vector instruction reading or writing scalar values:

1) Instruction *vsetvli* returns a scalar value to be stored in a scalar register.

2) Instructions *vle32.v* and *vse32.v* require a scalar value as a base address.

3) Instructions *vmadd.vx* and *vmacc.vx* require a scalar value as a coefficient.

In the register access stage (RA), where instructions are issued, only a vector instruction can be issued per time and no other instruction can be issued simultaneously (to avoid conflicts). Therefore, when vector instructions are issued, the dual-issue capability of the scalar pipeline is disabled. The execution stage (EXE) was modified

[**]This extended processor, comprising the baseline scalar processor and the vector extension, will be called *vector* processor.

[††]https://github.com/riscv/riscv-gnu-toolchain/tree/rvv-intrinsic.

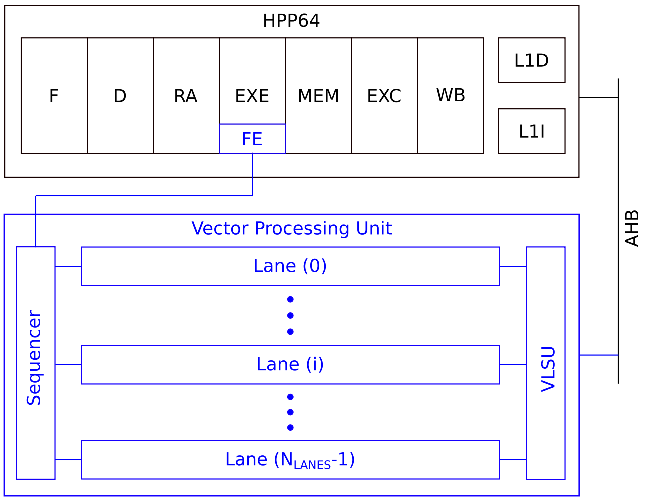


Fig. 1 The HPP64 NOEL-V interfaced with the VPU. In blue are the additions to the NOEL-V due to the vector extension described in this work.

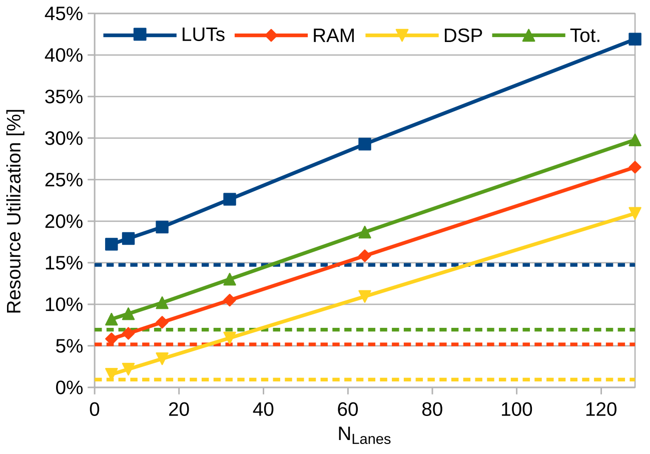


Fig. 2 Resource utilization on the XCKU040 for the scalar processor (dashed lanes) and vector processors with 4, 8, 16, 32, 64, and 128 lanes (solid lines with data points).

to include also a VPU front end (FE), as shown in Fig. 1. The other stages, i.e., fetch stage (F), memory stage (MEM), exception stage (EXC), and write back (WB) remained unchanged.

A. Resource Utilization

To have an estimation of the resources required (i.e., resource utilization) by configurations with different numbers of lanes, syntheses for the Xilinx Kintex Ultrascale (XCKU040) field-programmable gate array (FPGA) were run for different configurations. Resource utilization from implementation reports is shown in Fig. 2 for configurable logic blocks look-up tables (LUTs), block random-access memory (RAM) tiles, and digital signal processing (DSP) blocks (of the, respectively, 242,400, 600, and 1920 available in the selected FPGA). Furthermore, a total resource utilization (Tot.) has been calculated assuming that all the resources have the same importance (i.e., with an average of the three utilizations). The increase in terms of resource utilization is linear with the number of lanes, showing good scalability up to 128 lanes. Furthermore, all configurations are able to reach a frequency target of 100 MHz (the same as the scalar processor), which will be used as clock frequency of the processors in the remainder of the paper.

B. Memory Subsystem

As can be seen in Fig. 1, when using the scalar pipeline, the data are cached in a 16 KiB level 1 data cache (L1D). This is not the case for the data used by the vector pipeline. On the other hand, also operations on vectors are sped up by the level 1 instruction cache (L1I) in the HPP64. To investigate the role of the memory subsystem in the performance of scalar and vector processor, we investigated two different memory configurations:

1) *SRAM configuration:* Instructions and data are read from and written to a 4 MiB static random-access memory (SRAM) array on the AHB bus. This ensures a low latency of access from the processor. It is representative of memory hierarchies where tiling is taken care of (similarly to what done in [8]) or caching is implemented at level 2 (L2).

2) *SDRAM configuration:* Instructions and data are read from and written to an external 2 GB double data rate 4 (DDR4) synchronous dynamic random-access memory (SDRAM) module. This is representative of how the vector processor would operate at the current development stage in real-world applications, where the large amount of data involved typically requires an external main memory with relatively long latency of access.

To compare quantitatively the two memory configurations, we investigated the bandwidth of the SRAM configuration and of the SDRAM configuration when using the vector pipeline to read the first N_{Lanes} elements of a vector, as shown in Fig. 3 (top). To investigate further the behavior of the SDRAM module, we considered both an ideal model of the SDRAM (assuming that the SDRAM module is equivalent to the SRAM except for a longer latency), and real measurements from the SDRAM module on the KCU105 Evaluation Board by Xilinx [31]. We counted the CCs between the read request by the vector processor on the bus and the data available on the bus for the vector processor (CC_{lat}), with simulations for the SRAM configuration and with an on-chip logic analyzer provided with the GRLIB (LOGAN) for the SDRAM on the KCU105 board. This value was 2 CCs for the SRAM configuration and 24 CCs for the SDRAM configuration.

Assuming that the latency occurs only once (SRAM and ideal SDRAM), this can be seen by the core as a lower effective bandwidth (BW_{eff}), given by

$$BW_{\mathrm{eff}} = \frac{D_{\mathrm{read}}}{CC_{\mathrm{lat}} + CC_{\mathrm{read}}} \tag{1}$$

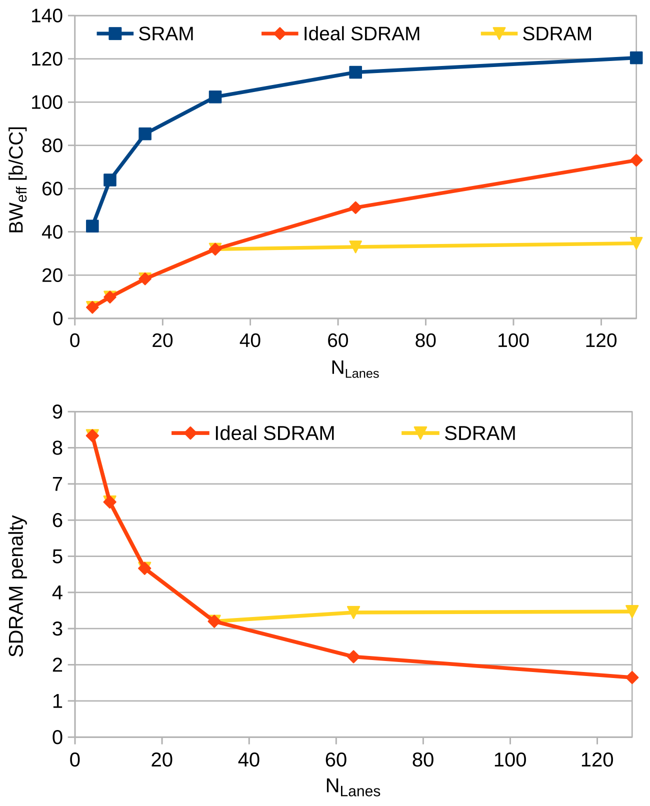


Fig. 3 Top: Effective bandwidth. Bottom: SDRAM penalty in terms of BW_{eff} for vector loads with different N_{Lanes}. Points represent measured data ($N_{\mathrm{Lanes}} = 4, 8, 16, 32, 64, 128$).

Table 1 Comparison of this work with other integer RVVE cores and other RISC-V cores targeting similar applications

Core	This work	Vicuna [39]	RI5CY [20,42]	Klessydra T13 [42]
DLP	VLA Lane-based	VLA Specialized units	SIMD (packed)	VLA Lane-based + MT
ISA	RVVE Int. subset	RVVE Int. subset	Int. NSE	Int. NSE (scratchpad)
Area overhead	24.1% (16 l vs 4 l) 98.6% (64 l vs bsp)	N/A	115.3% (vs bsp)	148.2% (8 l vs bsp)
Frequency penalty	~0% Up to 256 l, 8192b dp	20% 32b vs 1024b dp	22% (vs bsp)	5% (8 l vs bsp)
igemm ($N = 32$)	70.9 kCC (32b)	N/A	41.9 kCC (16b) 83.8 kCC (32b)*	N/A
igemm ($N = 64$)	314.3 kCC (32b)	N/A	1.4 MCC (32b)	414.4 kCC (32b)
igemm ($N = 256$)	81 ms	8.3 ms (8b) 33.3 ms (32b)*	N/A	N/A
iaxpy ($N = 65,536$)	678.8 μs (32b)	524.9 μs (8b) 2.1 ms (32b)*	N/A	N/A

Data with asterisks (*) are extrapolated for different vector element widths as described in Sec. IV.
In this table, "b" stands for bit, "dp" for datapath, "bsp" for baseline scalar processor, "l" for lanes, "Int." for integer, "MT" for multithreaded, and "NSE" for nonstandard extension.

where D_{read} is the amount of data to be read and CC_{read} is the number of CCs to read the data on the bus once it is available. For instance, considering a load involving all the lanes of a configuration with $N_{\text{Lanes}} = 4$ ($D_{\text{read}} = 128b$), where $CC_{\text{read}} = 1$ for both SRAM and SDRAM, $BW_{\text{eff,SRAM}} = 42.67b/CC$ and $BW_{\text{eff,SDRAM}} = 5.12b/CC$ (8.33× lower). Figure 3 (bottom) shows that the penalty in terms of BW_{eff} is reduced when the number of lanes is incremented, as vector processors use a single, longer access to mitigate the latency of memories. This effect smoothly saturates for the ideal SDRAM for larger N_{Lanes} and the SDRAM penalty tends asymptotically to 1, although $N_{\text{Lanes}} = 128$ is already enough to achieve a penalty of just 1.65. The increase of performance with the increase of N_{Lanes} saturates abruptly instead after the maximum burst size of the SDRAM is reached in real measurements (in this case the maximum burst is $128B$, i.e., the bandwidth saturates for $N_{\text{Lanes}} > 32$) and more than one burst read access is required to read the first N_{Lanes} elements of the vector. In this case CC_{lat} is the sum of the latency of each read burst access (two burst reads for 64 lanes and four for 128 lanes). The slight increase of BW_{eff} after $N_{\text{Lanes}} = 32$ is because the following read burst accesses are in the same SDRAM page of the first one and for this reason CC_{lat} is reduced to 21. In the following sections more complex SDRAM penalty plots will be proposed to investigate the effects of the two memory configurations for several benchmarks and hardware configurations.

III. Benchmarking

The roofline model [32] shows that increasing computational capabilities with DLP speeds up very effectively kernels with high operational intensity (*OI*), i.e., ratio between number of operations ($\#OP$) and bytes (B) read from and written to main memory. On the other hand, performance of kernels with low *OI* is limited by memory bandwidth. Therefore, improving computational capability in this case provides little or no speed-up.

In high-performance computing (HPC) optimized Basic Linear Algebra Subprograms (BLAS) routines [33,34] are typically available for each HPC platform, so that it is possible to map software to highly optimized routines. One of the most popular BLAS routines is general matrix multiply (*gemm*), which (in its nontransposed form) implements the following algorithm:

$$C \leftarrow \alpha A \cdot B + \beta C \tag{2}$$

where A, B, and C are matrices of, respectively, size $n1 \times n2$, $n2 \times n3$, and $n1 \times n3$, and α and β are scalars. This routine typically achieves the peak performance of a given platform for large-enough matrices. The reason is that its *OI* increases with the size of the matrices involved. For instance, assuming matrices composed of 32-bit integers (*igemm*[‡‡]), $\alpha = 1$, $\beta = 1$, and $n1 = n2 = n3 = N$, the operational intensity is $OI = (2N + 1)/16OP/B$ (we will refer to N as "problem size"). The C functions used for the scalar and vector processor are reported, respectively, in Figs. 4 and 5.[§§]

On the other hand, other BLAS kernels have low *OI* and even increasing the problem size will not make the workload compute-bound. A typical example is *axpy* ("*ax* plus *y*"), which implements the algorithm below:

$$y \leftarrow ax + y \tag{3}$$

where a is a scalar, and x and y are vectors of size N. In this case, OI for 32-bit integers (*iaxpy*) is stuck at $1/6OP/B$. The C functions used for the scalar and vector processor are reported, respectively, in Figs. 6 and 7.

A. Results

The benchmarking was conducted exploring the scaling of performance both changing the number of lanes (while keeping the size of the problem constant) and changing the size of the problem (while keeping the number of lanes constant). The results are reported, respectively, in Secs. III.A.1 and III.A.2.

1. Scalability in Terms of Number of Lanes

Figure 8 shows the performance of the scalar processor and vector processor with several $N_{\text{Lanes}} \leq N$ (in the case of $N = 128$) for both memory configurations. Thanks to the VLA nature of the RVVE, the same executable of the vector function with $N = 128$ was used for all the hardware configurations ranging from 4 to 128 lanes, exploiting the maximum level of DLP available on each configuration.

Figure 9 shows that increasing the number of lanes in the vector processor increases the speed-up compared to the scalar processor with the same memory configuration until $N_{\text{Lanes}} = N$ is reached, for both *igemm* and *iaxpy*. The maximum speed-up achieved when employing the SRAM configuration is, respectively, around 15.6× and 4.8× the performance of the scalar processor. For the SDRAM configuration the maximum speed-up is higher for *igemm* and lower for *iaxpy*, respectively, around 23.0× and 4.1×. It should be noted that

[‡‡]Although BLAS routines are typically employed for floating operations and a letter is added to the name of the routine according to the precision (e.g., *sgemm* for single precision and *dgemm* for double precision), here we consider 32-bit integers and use the term *igemm*, as done in [35].

[§§]Based on the *sgemm* function in https://github.com/riscv-non-isa/rvv-intrinsic-doc/blob/master/examples/rvv_sgemm.c.

```
int a_array[N*N];
int b_array[N*N];
int c_array[N*N];

void igemm() {
  for (size_t i = 0; i < N; ++i)
    for (size_t j = 0; j < N; ++j)
      for (size_t k = 0; k < N; ++k)
        c[i*N+j] += a[i*N+k] * b[j+k*N];}
```

Fig. 4 C function employed for *igemm* on the scalar processor.

```
void igemm_rvv(size_t  N, const int *a, size_t lda,
               const int *b, size_t ldb, int *c, size_t ldc) {
  size_t vl;
  for (int m = 0; m < N; ++m) {
    const int *b_n_ptr = b;
    int *c_n_ptr = c;
    for (int n = N; (vl = vsetvl_e32m1(n)); n -= vl) {
      const int *a_k_ptr = a;
      const int *b_k_ptr = b_n_ptr;
      vint32m1_t acc = vle32_v_i32m1(c_n_ptr);
      for (size_t k = 0; k < N; ++k) {
        vint32m1_t b_n_data = vle32_v_i32m1(b_k_ptr);
        acc = vmacc_vx_i32m1(acc, *a_k_ptr, b_n_data);
        b_k_ptr += ldb;
        a_k_ptr++;}
      vse32_v_i32m1(c_n_ptr, acc);
      c_n_ptr += vl;
      b_n_ptr += vl;}
    a += lda;
    c += ldc;}}
```

Fig. 5 C function employed for *igemm* on the vector processor.

```
void iaxpy (int32_t x[], int32_t y[], int32_t a, size_t N) {
  int i;
  for (i=0; i<N; i++) {
    y[i]=(a*x[i])+y[i];}}
```

Fig. 6 C function employed for *iaxpy* on the scalar processor.

```
void iaxpy_rvv(int32_t *x, int32_t *y, int32_t a, size_t N) {
  size_t vl;
  vint32m1_t vx, vy;
  for (int n = N; vl = vsetvl_e32m1(n); n -= vl) {
    vx = vle32_v_i32m1 (x);
    vy = vle32_v_i32m1 (y);
    vy = vmacc_vx_i32m1 (vy, a, vx);
    vse32_v_i32m1 (y, vy);
    x += vl;
    y += vl;}}
```

Fig. 7 C function employed for *iaxpy* on the vector processor.

there are some configurations for which the vector processor is slower than the scalar one. For the SRAM configuration this is the case when $N_{\text{Lanes}} = 4$ and $N_{\text{Lanes}} = 8$ in case of *iaxpy* (respectively, 0.36× and 0.68×), while this is the case only for $N_{\text{Lanes}} = 4$ for *igemm* (0.75×). For the SDRAM configuration this happens only for *iaxpy* with $N_{\text{Lanes}} = 4$ (0.51×) and $N_{\text{Lanes}} = 8$ (0.95×). It should also be noted that while the speed-up when running *igemm* on the vector processor with the SDRAM configuration is larger than the speed-up of the vector processor for the SRAM configuration, this advantage is reduced for $N_{\text{Lanes}} > 32$. In the case of *iaxpy*, there is a similar trend and the relation is even inverted for $N_{\text{Lanes}} > 64$. Figure 10 investigates further this aspect, showing that the penalty of using an SDRAM configuration, measured as the ratio between the execution time with SDRAM and SRAM configuration, is lower for the uncached vector processor than for the cached scalar processor for *igemm* (even though it is clear that the need of several burst read accesses to the SDRAM module to read N_{Lanes} elements of a vector increases the SDRAM penalty of the vector processor for $N_{\text{Lanes}} > 32$), while for *iaxpy* it becomes higher after $N_{\text{Lanes}} = 64$.

Given that the increase of performance with the number of lanes is typically less than linear (Fig. 8) and that the resource utilization increases linearly instead (Fig. 2), there is an optimal number of lanes that maximizes the resource utilization efficiency in terms of performance/resources, where the required resources are expressed as a fraction of those available on the FPGA. Figure 11 (top) shows that for the SRAM configuration this happens for $N_{\text{Lanes}} = 32$ in case of *igemm* for the DSP blocks, while the total resource utilization efficiency increases up to 3.63× the one of the scalar processor for

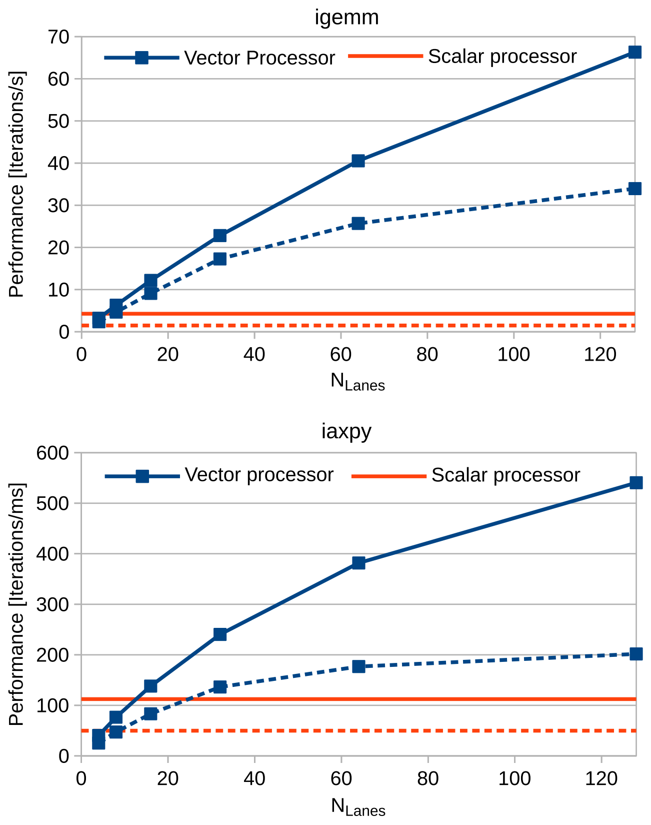


Fig. 8 **Performance (iterations/s) of the vector processor varying the number of lanes and of the scalar processor for *igemm* (top) and *iaxpy* (bottom) with $N = 128$ when using the AHB RAM (solid lines) and the external SDRAM (dashed lines). Points represent measured data ($N_{\text{Lanes}} = 4, 8, 16, 32, 64, 128$).**

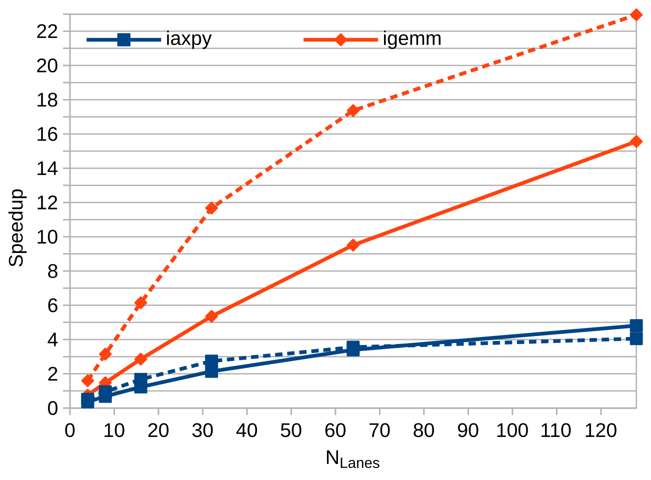


Fig. 9 **Speed-up of the vector processor over the scalar processor varying the number of lanes for *iaxpy* and *igemm* with $N = 128$ when using the AHB RAM (solid lines) and the external SDRAM (dashed lines). Points represent measured data ($N_{\text{Lanes}} = 4, 8, 16, 32, 64, 128$).**

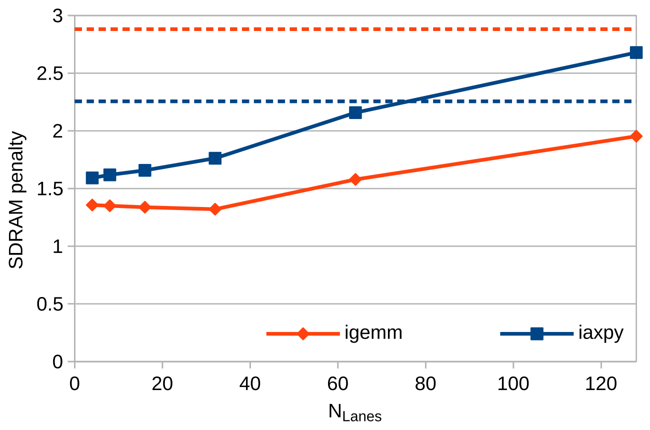


Fig. 10 **SDRAM penalty in terms of performance for the vector processor (solid lines) and for the scalar processor (dashed lines), varying the number of lanes for *iaxpy* and *igemm* with $N = 128$. Points represent measured data ($N_{\text{Lanes}} = 4, 8, 16, 32, 64, 128$).**

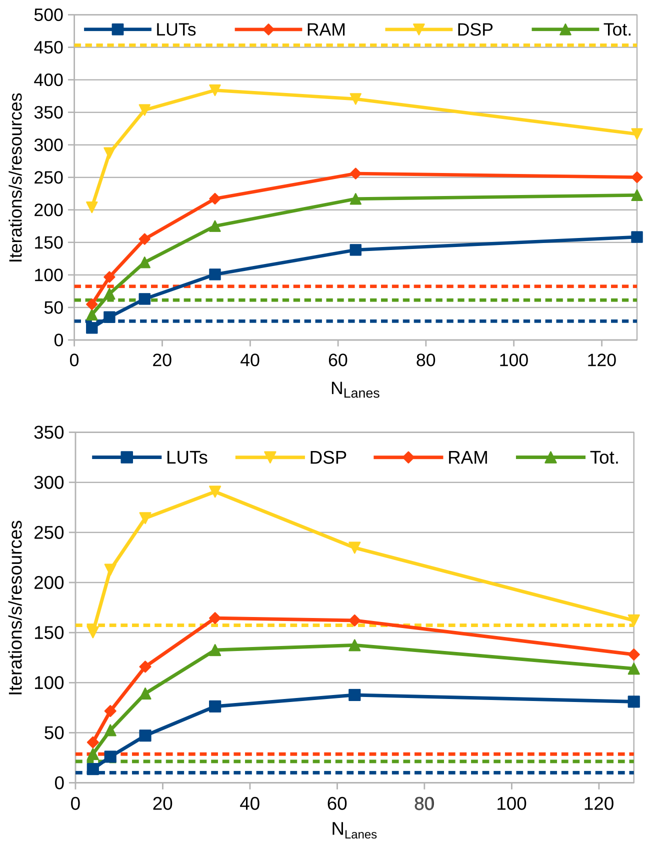
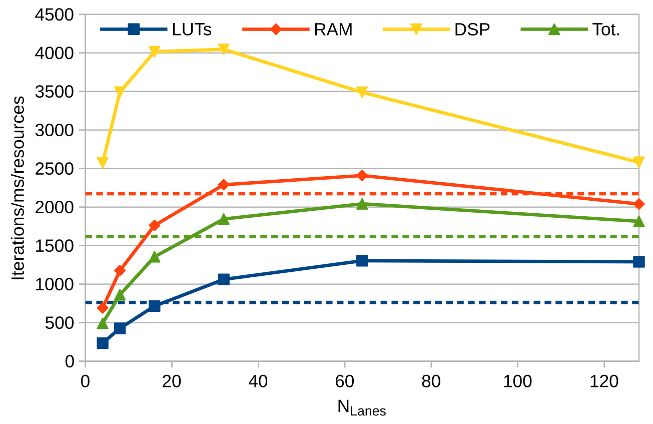


Fig. 11 **Utilization efficiency varying N_{Lanes} for *igemm* with $N = 128$ when using SRAM (top) and SDRAM (bottom). Solid lines represent data for the vector processor, and dashed lines for the scalar processor. Points represent measured data ($N_{\text{Lanes}} = 4, 8, 16, 32, 64, 128$).**

$N_{\text{Lanes}} = N = 128$. Finally, the *igemm* resource utilization efficiency for the vector processor is higher than the one for the scalar processor only if at least 8 lanes can be implemented, which means that adding the VPU increases resource efficiency only if a resource utilization of 1.28× the scalar processor can be tolerated (see Fig. 2). When considering the SDRAM configuration (Fig. 11, bottom), the efficiencies are smaller in terms of absolute value compared to the SRAM configuration and the total efficiency peaks for $N_{\text{Lanes}} = 64$. However, the peak reached has a higher relative value compared to the total utilization efficiency of the respective baseline (6.46×). Furthermore, for the SDRAM configuration all types of resource efficiency of the vector processors (except for DSP blocks and $N_{\text{Lanes}} = 4$) are higher compared to the scalar processor (providing higher total efficiency already for 1.18× the resources of the scalar processor). The scalability of the total resource efficiency of the vector processor with the increase of DLP is, as expected by the roofline model, worse for *iaxpy*. In this case, considering the SRAM configuration, the vector processor has a total resource utilization efficiency better than the scalar processor for $N_{\text{Lanes}} > 16$, peaking at 1.26× the one of the scalar processor for $N_{\text{Lanes}} = 64$ (Fig. 12). Furthermore, as in *iaxpy* the functional units are much less exploited by the software, even for the configuration with the peak DSP utilization efficiency ($N_{\text{Lanes}} = 32$), its value is 0.34×



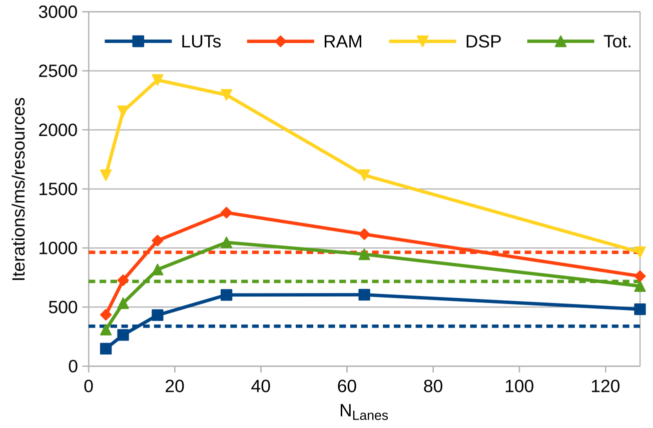


Fig. 12 Utilization efficiency varying N_{Lanes} for *iaxpy* with $N = 128$ when using SRAM (top) and SDRAM (bottom). Solid lines represent data for the vector processor, and dashed lines for the scalar processor. Points represent measured data ($N_{\text{Lanes}} = 4$, 8, 16, 32, 64, 128). The DSP utilization efficiency for the scalar version is not reported, because it is outside the represented range (11,953 iterations/ms/resources for SRAM, 5299 iterations/ms/resources for SDRAM).

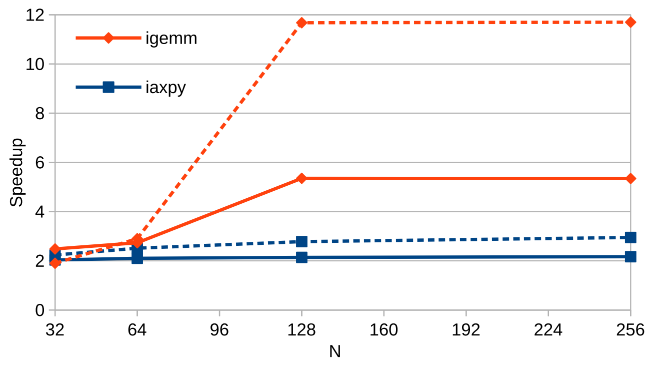


Fig. 13 Speed-up varying the size of the problem N with an SRAM (solid lines) and SDRAM (dashed lines) configuration for both *iaxpy* and *igemm* with $N_{\text{Lanes}} = 32$. Points represent measured data ($N = 32$, 64, 128, 256).

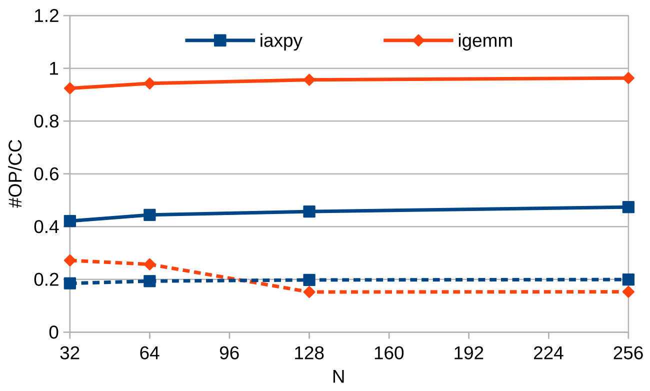


Fig. 14 Number of operations per CC varying the size of the problem N for both *iaxpy* and *igemm* with $N_{\text{Lanes}} = 32$ when using the SRAM configuration. Solid lines represent data for the vector processor, and dashed lines for the scalar processor. Points represent measured data ($N = 32$, 64, 128, 256).

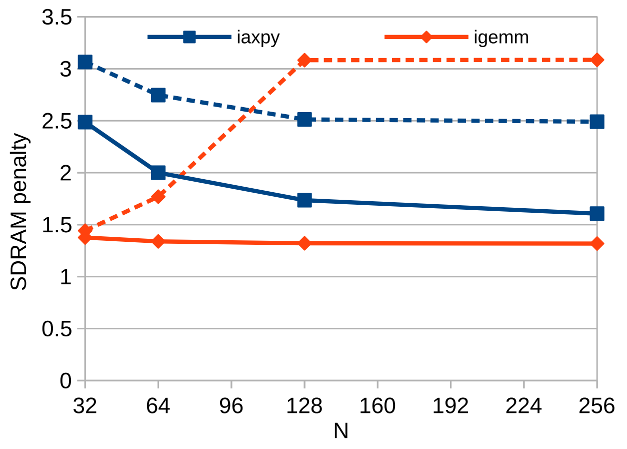


Fig. 15 SDRAM penalty in terms of performance, varying the size of the problem N for both *iaxpy* and *igemm* with $N_{\text{Lanes}} = 32$. Solid lines represent data for the vector processor, and dashed lines for the scalar processor. Points represent measured data ($N = 32$, 64, 128, 256).

the one of the scalar processor. Similarly to *igemm*, the resource efficiency of *iaxpy* for the SDRAM configuration is lower as an absolute value compared to the SRAM configuration, but it is higher compared to the respective baseline. In this case, the vector processor has a total resource utilization efficiency better than the scalar processor already for $N_{\text{Lanes}} > 8$, its peak is at $N_{\text{Lanes}} = 32$, and it is 1.46× the one of the scalar processor. The following decrease makes the efficiency of the vector processor lower than the one of the scalar processor for $N_{\text{Lanes}} = 128$. Furthermore, the peak of the DSP efficiency is at $N_{\text{Lanes}} = 16$ and reaches 0.46× the one of the scalar processor.

2. *Scalability in Terms of the Problem Size*

To show how increasing the size of the problem N increases the advantage of the vector processor over the scalar processor, we investigated how the two processors behave when N is increased above 32 for a vector processor with $N_{\text{Lanes}} = 32$. The results are shown in Fig. 13.

For both the SRAM and SDRAM configurations, the speed-up increases with the OI (proportional to N) for *igemm* until $N = 128$ (11.70×), after which it saturates. Figure 14 gives more insights on why this happens, considering the effective number of operations per CC ($\#OP/CC$) calculated as the number of operations ($2N^3 + N^2$ for *igemm* and $2N$ for *iaxpy*) for the SRAM configuration. For *igemm*, increasing the size of the problem slightly increases $\#OP/CC$ for the vector processor, as the overhead of calling the function is spread over more calculations. On the other hand, $\#OP/CC$ for the scalar processor decreases when N is increased because the matrices get larger, leading to an increase of cache misses. The same can be deduced by the SDRAM penalty shown in Fig. 15, which shows that in the case of *igemm* the SDRAM penalty increases for the scalar processor when going from $N = 32$ to $N = 128$, meaning that the L1D is less capable of masking the latency of the main memory as matrices get larger. The SDRAM penalty is instead roughly constant for the vector processor. It should also be noted that large problem sizes favor the vector processor even for *iaxpy*, increasing the gap in terms of SDRAM penalty as the problem size increases (even if the SDRAM penalty of the scalar processor does not increase).

IV. Related Work

Although the RVVE is still in the process of being standardized, it plays such a crucial role in state-of-the-art terrestrial applications that already several developments implementing the RVVE are described in literature. One of the most mature implementations described in literature is Xuantie-910 from Alibaba, a 16-core RISC-V processor supporting the RVVE [27]. Its FPGA prototypes are employed in data servers and a 12 nm Application-Specific Integrated Circuit (ASIC) clocked up to 2.5 GHz has been taped out. However, this processor clearly targets servers and HPC applications, which have totally different constraints compared to space embedded systems. Also a European consortium [36] is working on a RISC-V Vector accelerator for HPC applications and an ASIC has been recently taped out. Other companies like SiFive have announced commercial cores supporting the RVVE [37].

A. Comparison with Other Implementations

The microarchitecture described in our work can be seen as a simplified version of Ara [24], a RISC-V vector processor based on the scalar Ariane [35] and implementing the RVVE with vector lanes. However, Ara aims at implementing (almost) all of the RVVE (excluding fixed-point and vector atomics [38]), while in our work we decided to implement only a reduced integer subset. For this reason (especially because of floating point units), the scaling of the resources required by Ara with the number of lanes is much worse: for instance, in our work the 16-lane version uses just 24.1% resource more than the 4-lane version, while for Ara the area increases by 212.6%. This allows us to reach massive DLP on a single core with limited resources. On the other hand, Ara proves that the complete RVVE, although very interesting for HPC applications, is overdimensioned for resource-constrained embedded applications.

The approach followed in our work is not the only possible way of implementing the RVVE. For example, in [39] Vicuna is described and benchmarked. Vicuna uses dedicated execution units for different instruction types that process several elements at once instead of vector lanes [39]. Also in this case, only integer and fixed-point instructions are implemented. Although a direct comparison with our implementation is not possible (because Vicuna operates on 8-bit vector elements instead of 32-bit vector elements), few considerations can be drawn. The wider memory interface employed in our case (128 bits) compared to Vicuna (32 bits) allows for faster execution of memory-bound algorithms. In case of *iaxpy* with $N = 65536$ on 8-bit integer data, Vicuna requires 524.9 μs for its execution. Multiplying by 4 to keep into account the higher memory traffic required, it can be estimated a required time of 2.1 ms for 32-bit elements. Our 256-lane implementation executes $N = 65536$ *iaxpy* in 678.8 μs. Furthermore, our modular approach and the subset of chosen instructions does not cause penalties on the maximum frequency (100 MHz) up to 256 lanes of 32 bits each (8192-bit datapath), while the largest version of Vicuna (1024-bit multiplier) has a 20% penalty over the smallest (32-bit multiplier). A similar penalty would be even larger for execution units operating on 32-bit data with the same DLP (4096-bit multiplier). However, an 8-bit 256×256 *igemm* executes in 8.3 ms on Vicuna. Assuming that it would be possible to implement a 32-bit version of Vicuna without any penalties on the frequency, an equivalent time of 33.3 ms can be estimated for 32-bit elements (x4). This is better than our 256 lanes implementation (81 ms), meaning that a 32-bit version of Vicuna would be faster than our implementation if it achieves a clock frequency higher than 33 MHz. A vector processor similar to Vicuna is described in [40]. In this case a 120×120 matrix multiplication executes on this processor capable of executing 8 OP/CC on 8-bit data in 3.0 *MCC*, equivalent to 12.0 *MCC* for 32-bit elements. Our implementation with the same potential number of OP/CC (8 lanes) executes a 120×120 *igemm* on 32-bit elements in 13.1 *MCC*. Although using a different FPGA, the implementation in [40] achieves only 50 MHz, confirming that this approach has worse scalability in terms of maximum clock frequency with the level of DLP implemented compared to ours. It is shown in Refs. [39,40] that operating on 8-bit elements is an interesting solution to speed up calculations and reduce memory traffic when the precision of 32 bits is not needed. Data from [41] show that 8 bit is usually the smallest factor that allows an acceptable loss of accuracy for DNNs.

Another possible approach is the one of Klessydra T13 [42], which is based on a nonstandard custom RISC-V vector extension. This extension is comparable to the subset implemented in our work (as also in this case the focus is on resource-constrained embedded applications, and, for instance, floating point operations are not implemented), but it is centered around a scratchpad memory instead of a VRF. Furthermore, the Klessydra platform is based on a interleaved multithreated microarchitecture, alternating instructions belonging to different execution threads in the stages of a single-issue in-order pipeline [42]. The data in [42] show that the single-core implementation using the maximum DLP available (8 elements) executes a 64×64 matrix multiplication on 32-bit values in 484.4 *kCC* for homogeneous workloads and 414.4 *kCC* for heterogeneous workloads,[¶] while our 64-lane implementations run the 64×64 *igemm* in 314.2 *kCC*. The Klessydra T13 core requires $+148.22\%$ the LUTs of the baseline scalar processor, while our 64-lane processor $+98.64\%$ over its baseline scalar processor.

Our implementation can be compared also with RISC-V implementations using nonstandard packed SIMD extensions, like RI5CY [20]. This 32-bit core implements custom-packed SIMD instructions to operate on four 8-bit elements or two 16-bit elements with a single instruction [20]. Furthermore, the core implements other instructions to speed up matrix multiplications and other kernels composed by loops of computations (e.g., hardware loops and postincrement addressing modes) [20]. When comparing the area required for RI5CY (40.7 kGE) with the one required for Zero-Riscy (18.9 kGE), there is an area overhead of 115.3% [42]. The frequency instead is reduced to 91.4 from 117.2 MHz [42].

Data for RI5CY in [42] show that for a 64×64 matrix multiplications our 64-lane implementation is around 4.3× faster than RI5CY operating on 32-bit values (314.3 *kCC* against 1.4 *MCC*). This means that our 64-lane implementation in this case is roughly as computationally capable as four RI5CY cores and therefore the eight cores in PULP can be potentially outperformed with a dual-core implementation of our vector processor. However, for smaller matrix multiplications, this advantage is reduced. For a 16-bit 32×32 matrix multiplication in [43] a single-core RI5CY takes 41.9 *kCC*, while for our 32-lane implementation it takes 70.9 *kCC*. Even adjusting for the different data element size (x2), our processor would be just 1.18× faster. An approach to improve the performance of our implementation for small matrices is to implement the *vmadd* and *vmacc* instructions in a single clock cycle. This would have little or no impact for a small number of lanes in terms of maximum clock frequency, roughly halving the execution time. The results of the comparison of this work with similar implementations are summarized in Table 1.

B. Benchmarking Methodology

Considering the methodology employed in the benchmarking, the mentioned related works do not investigate the effects of a fast and a slow memory on the vector processor (Platzer and Puschner [39] provide performance only for an SRAM), some of them [24,42] not even providing details on the memory used. On the other hand, in our work we prove how cacheless vector processors have an increased advantage over cached scalar processors when dealing with relatively slow, high-latency memories (Fig. 9).

Although the benchmarking in this work was carried out in depth and with a detailed analysis of the results, it lacks a direct connection to the workloads required in space applications. For this reason, another benchmark was carried out in [11]. In [11], the processor was employed to speed-up the inference of CloudNet, a fully convolutional network (FCN) [44] for cloud detection; i.e., its output is a mask of the same size as the input image indicating the pixels covered with clouds. When employing 128 lanes, the processor described in this work is 19.6× faster than the scalar baseline [11].

[¶]Given the interleaved multithreated microarchitecture of the Klessydra cores, data from [42] are reported both in the case the threads are based on the same algorithm operating on different data (homogeneous workload) and in the case where different algorithms are executed as different threads (heterogeneous workload).

V. Conclusions

State-of-the-art processors for space embedded systems are based on simple microarchitectures, because of the small footprint required, low power available, and presence of ionizing radiation. In this paper the implementation of DLP to enable compute-intensive workloads in next-generation space embedded systems with limited resources was described.

The resources required for the hardware implementation of the selected RVVE subset increase roughly linearly with the number of lanes (i.e., its computational capabilities), with no noticeable penalty in the maximum clock frequency achievable compared to the baseline scalar processor up to 128 lanes. Given a certain size of the problem, the hardware implementation of the selected integer RVVE subset shows good scalability with the number of lanes, especially for compute-intensive workloads (*igemm*), having resource utilization efficiency above the one of the scalar processor for all configurations with more than four lanes, achieving up to 23.0× the performance of the scalar processor with only 4.3× the resources of the scalar prototype, and increasing resource utilization efficiency up to 6.5× the one of the scalar processor. Furthermore, for a fixed configuration, the performance of the vector processor scales better than the performance of the scalar processor with the increase of the problem size.

Although this work shows that applying DLP in next-generation processors is feasible and effective for certain workloads, further work is needed to investigate whether an L1 vector data cache can speed the execution for relatively small number of lanes or small problem size, whether other subsets of the RVVE are fit for implementations with small footprint, and which RVVE instructions not implemented yet are detrimental to the scaling in terms of number of lanes when considering resource efficiency and maximum achievable frequency. Finally, also the implementation of operations on smaller vector elements is an interesting subject to investigate to increase performance when lower precision is acceptable.

Acknowledgments

This work was supported by Cobham Gaisler AB and the European Space Agency (under the Networking/Partnering Initiative program). The authors would like to thank Jan Andersson and Nils-Johan Wessman for their support and assistance in this work.

References

[1] Andersson, J., Hjorth, M., Johansson, F., and Habinc, S., "LEON Processor Devices for Space Missions: First 20 Years of LEON in Space," *2017 6th International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, IEEE, New York, 2017, pp. 136–141.
https://doi.org/10.1109/SMC-IT.2017.31

[2] Andersson, J., "Development of a NOEL-V RISC-V SoC Targeting Space Applications," *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, IEEE Computer Soc., Los Alamitos, CA, 2020, pp. 66–67.
https://doi.org/10.1109/DSN-W50199.2020.00020

[3] "GRLIB IP Core User's Manual," Ver. 2020.2, Cobham Gaisler AB, June 2020, https://www.gaisler.com/products/grlib/grip.pdf.

[4] Gal-On, S., and Levy, M., "Exploring CoreMark–A Benchmark Maximizing Simplicity and Efficacy," *The Embedded Microprocessor Benchmark Consortium*, Whitepaper, 2012, https://www.eembc.org/techlit/articles/coremark-whitepaper.pdf.

[5] Di Mascio, S., Menicucci, A., Gill, E., Furano, G., and Monteleone, C., "Leveraging the Openness and Modularity of RISC-V in Space," *Journal of Aerospace Information Systems*, Vol. 16, No. 11, 2019, pp. 454–472.
https://doi.org/10.2514/1.I010735

[6] Furano, G., Tavoularis, A., and Rovatti, M., "AI in Space: Applications Examples and Challenges," *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, IEEE, New York, 2020, pp. 1–6.
https://doi.org/10.1109/DFT50435.2020.9250908

[7] Giuffrida, G., Diana, L., de Gioia, F., Benelli, G., Meoni, G., Donati, M., and Fanucci, L., "CloudScout: A Deep Neural Network for On-Board Cloud Detection on Hyperspectral Images," *Remote Sensing*, Vol. 12, No. 14, 2020, Paper 2205.
https://doi.org/10.3390/rs12142205

[8] Cappellone, D., Di Mascio, S., Furano, G., Menicucci, A., and Ottavi, M., "On-Board Satellite Telemetry Forecasting with RNN on RISC-V Based Multicore Processor," *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, IEEE, New York, 2020, pp. 1–6.
https://doi.org/10.1109/DFT50435.2020.9250796

[9] Di Mascio, S., Menicucci, A., Gill, E., Furano, G., and Monteleone, C., "On-Board Decision Making in Space with Deep Neural Networks and RISC-V Vector Processors," *Journal of Aerospace Information Systems*, Vol. 18, No. 8, 2021, pp. 553–570.
https://doi.org/10.2514/1.I010916

[10] Abdelouahab, K., Pelcat, M., Sérot, J., and Berry, F., "Accelerating CNN Inference on FPGAs: A Survey," 2018, http://arxiv.org/abs/1806.01683.

[11] Di Mascio, S., "Spin-in of RISC-V Processors in Space Embedded Systems," Ph.D. Thesis, Delft Univ. of Technology, Delft, The Netherlands, 2022.
https://doi.org/10.4233/uuid:e515547e-62bc-4893-b299-87c1286b5d55

[12] Sze, V., Chen, Y., Yang, T., and Emer, J. S., "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proceedings of the IEEE*, Vol. 105, No. 12, 2017, pp. 2295–2329.

[13] Mathieu, M., Henaff, M., and LeCun, Y., "Fast Training of Convolutional Networks Through FFTs," arXiv preprint arXiv:1312.5851, 2013.

[14] Schwank, J. R., Shaneyfelt, M. R., and Dodd, P. E., "Radiation Hardness Assurance Testing of Microelectronic Devices and Integrated Circuits: Radiation Environments, Physical Mechanisms, and Foundations for Hardness Assurance," *IEEE Transactions on Nuclear Science*, Vol. 60, No. 3, 2013, pp. 2074–2100.
https://doi.org/10.1109/TNS.2013.2254722

[15] Dodds, N. A., *Single Event Latchup: Hardening Strategies, Triggering Mechanisms, and Testing Considerations*, Vanderbilt Univ., Nashville, TN, 2012, Chaps. I and II, https://etd.library.vanderbilt.edu/etd-110320 12-225718.

[16] Di Mascio, S., Menicucci, A., Gill, E., Furano, G., and Monteleone, C., "Open-Source IP Cores for Space: A Processor-Level Perspective on Soft Errors in the RISC-V era," *Computer Science Review*, Vol. 39, Feb. 2021, Paper 100349.
https://doi.org/10.1016/j.cosrev.2020.100349

[17] Peleg, A., and Weiser, U., "MMX Technology Extension to the Intel Architecture," *IEEE Micro*, Vol. 16, No. 4, 1996, pp. 42–50.
https://doi.org/10.1109/40.526924

[18] Thakkur, S., and Huff, T., "Internet Streaming SIMD Extensions," *Computer*, Vol. 32, No. 12, 1999, pp. 26–34.
https://doi.org/10.1109/2.809248

[19] Doolan, D. C., Tabirca, S., and Yang, L. T., "Mobile Parallel Computing," *2006 Fifth International Symposium on Parallel and Distributed Computing*, IEEE, New York, 2006, pp. 161–167.

[20] Gautschi, M., Schiavone, P. D., Traber, A., Loi, I., Pullini, A., Rossi, D., Flamand, E., Gürkaynak, F. K., and Benini, L., "Near-Threshold RISC-V Core with DSP Extensions for Scalable IoT Endpoint Devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 25, No. 10, 2017, pp. 2700–2713.
https://doi.org/10.1109/TVLSI.2017.2654506

[21] Dabbelt, D., Schmidt, C., Love, E., Mao, H., Karandikar, S., and Asanovic, K., "Vector Processors for Energy-Efficient Embedded Systems," *Proceedings of the Third ACM International Workshop on Many-Core Embedded Systems*, Assoc. for Computing Machinery, New York, 2016, pp. 10–16.
https://doi.org/10.1145/2934495.2934497

[22] Stephens, N., Biles, S., Boettcher, M., Eapen, J., Eyole, M., Gabrielli, G., Horsnell, M., Magklis, G., Martinez, A., Premillieu, N., Reid, A., Rico, A., and Walker, P., "The ARM Scalable Vector Extension," *IEEE Micro*, Vol. 37, No. 2, 2017, pp. 26–39.
https://doi.org/10.1109/MM.2017.35

[23] Sato, M., Ishikawa, Y., Tomita, H., Kodama, Y., Odajima, T., Tsuji, M., Yashiro, H., Aoki, M., Shida, N., Miyoshi, I., Hirai, K., Furuya, A., Asato, A., Morita, K., and Shimizu, T., "Co-Design for A64FX Manycore Processor and 'Fugaku'," *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, New York, 2020, pp. 1–15.
https://doi.org/10.1109/SC41405.2020.00051

[24] Cavalcante, M., Schuiki, F., Zaruba, F., Schaffner, M., and 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*, Vol. 28, No. 2, 2020, pp. 530–543.
https://doi.org/10.1109/TVLSI.2019.2950087

[25] "Working Draft of the Proposed RISC-V V Vector Extension," GitHub, Inc., https://github.com/riscv/riscv-v-spec [retrieved 7 Jan. 2021].

[26] Lee, Y., Ou, A., Schmidt, C., Karandikar, S., Mao, H., and Asanovic, K., "The Hwacha Microarchitecture Manual, Version 3.8.1," EECS Dept., Univ. of California TR UCB/EECS-2015-263, Berkeley, CA, 2015, https://aspire.eecs.berkeley.edu/wp/wp-content/uploads/2016/02/EECS-2015-263.pdf.

[27] Chen, C., Xiang, X., Liu, C., Shang, Y., Guo, R., Liu, D., Lu, Y., Hao, Z., Luo, J., Chen, Z., Li, C., Pu, Y., Meng, J., Yan, X., Xie, Y., and Qi, X., "Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-Bit High Performance RISC-V Processor with Vector Extension : Industrial Product," *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, New York, 2020, pp. 52–64. https://doi.org/10.1109/ISCA45697.2020.00016

[28] Rico, A., Joao, J. A., Adeniyi-Jones, C., and Van Hensbergen, E., "ARM HPC Ecosystem and the Reemergence of Vectors: Invited Paper," *Proceedings of the Computing Frontiers Conference*, Assoc. for Computing Machinery, New York, 2017, pp. 329–334. https://doi.org/10.1145/3075564.3095086

[29] Batten, D., Jinturkar, S., Glossner, J., Schulte, M., and D'Arcy, P., "A New Approach to DSP Intrinsic Functions," *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, IEEE, New York, 2000, pp. 1–10. https://doi.org/10.1109/HICSS.2000.926967

[30] "RISC-V Vector Extension Intrinsic Document," GitHub, Inc., https://github.com/riscv/rvv-intrinsic-doc [retrieved 7 Jan. 2021].

[31] "KCU105 Board–User Guide (UG917)," Xilinx, San Jose, CA, 2019, Chap. 1, https://www.xilinx.com/support/documents/boards_and_kits/kcu105/ug917-kcu105-eval-bd.pdf.

[32] Williams, S., Waterman, A., and Patterson, D., "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Communications of the ACM*, Vol. 52, No. 4, 2009, pp. 65–76. https://doi.org/10.1145/1498765.1498785

[33] "BLAS (Basic Linear Algebra Subprograms)," http://www.netlib.org/blas/ [retrieved 7 Jan. 2021].

[34] Nakata, M., *Basics and Practice of Linear Algebra Calculation Library BLAS and LAPACK*, Springer Singapore, Singapore, 2019, pp. 83–112. https://doi.org/10.1007/978-981-13-6194-4_6

[35] Zaruba, F., and 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*, Vol. 27, No. 11, 2019, pp. 2629–2640. https://doi.org/10.1109/TVLSI.2019.2926114

[36] Labarta, J., "The RISC-V Vector Processor in EPI," *3rd RISC-V Meeting*, Paris, France, March 2021, https://open-src-soc.org/2021-03/media/slides/3rd-RISC-V-Meeting-2021-03-31-11h30-Jes%C3%BAs-Labarta.pdf.

[37] "SiFive Intelligence X280," SiFive, https://www.sifive.com/cores/intelligence-x280 [retrieved 15 Sept. 2021].

[38] Cavalcante, M., "Ara v2.0: RISC-V Vector Processor Implementation in GlobalFoundries 22FDX," *RISC-V Summit Proceedings*, San Jose, CA, Dec. 2019, https://riscv.org/wp-content/uploads/2019/12/12.11-16.40a-Ara-v2.0-RISC-V-Vector-Processor.pdf.

[39] Platzer, M., and Puschner, P., "Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation," *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, EUROMICRO, Sankt Augustin, Germany, 2021.

[40] Johns, M., and Kazmierski, T. J., "A Minimal RISC-V Vector Processor for Embedded Systems," *2020 Forum for Specification and Design Languages (FDL)*, IEEE, New York, 2020, pp. 1–4. https://doi.org/10.1109/FDL50818.2020.9232940

[41] Guo, K., Zeng, S., Yu, J., Wang, Y., and Yang, H., "[DL] A Survey of FPGA-Based Neural Network Inference Accelerators," *ACM Transactions on Reconfigurable Technology and Systems*, Vol. 12, No. 1, 2019, pp. 1–26. https://doi.org/10.1145/3289185

[42] Cheikh, A., Sordillo, S., Mastrandrea, A., Menichelli, F., Scotti, G., and Olivieri, M., "Klessydra-T: Designing Vector Coprocessors for Multithreaded Edge-Computing Cores," *IEEE Micro*, Vol. 41, No. 2, 2021, pp. 64–71. https://doi.org/10.1109/MM.2021.3050962

[43] Flamand, E., Rossi, D., Conti, F., Loi, I., Pullini, A., Rotenberg, F., and Benini, L., "GAP-8: A RISC-V SoC for AI at the Edge of the IoT," *2018 IEEE 29th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, IEEE, New York, 2018, pp. 1–4. https://doi.org/10.1109/ASAP.2018.8445101

[44] Shelhamer, E., Long, J., and Darrell, T., "Fully Convolutional Networks for Semantic Segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 39, No. 4, 2017, pp. 640–651. https://doi.org/10.1109/TPAMI.2016.2572683

E. Atkins
Associate Editor