SPARROW: A Low-Cost Hardware/Software Co-designed SIMD Microarchitecture for AI Operations in Space Processors

Marc Solé Bonet*†
*Universitat Politècnica de Catalunya (UPC)

Leonidas Kosmidis†*
†Barcelona Supercomputing Center (BSC)

***Abstract*—Recently there is an increasing interest in the use of artificial intelligence for on-board processing as indicated by the latest space missions, which cannot be satisfied by existing low-performance space-qualified processors. Although COTS AI accelerators can provide the required performance, they are not designed to meet space requirements. In this work, we co-design a low-cost SIMD micro-architecture integrated in a space qualified processor, which can significantly increase its performance. Our solution has no impact on the processor's 100 MHz frequency and consumes minimal area thanks to its innovative design compared to conventional vector micro-architectures. For the minimum configuration of our baseline space processor, our results indicate a performance boost of up to 9.3× for commonly used AI-related and image processing algorithms and 5.5× faster for a complex, space-relevant inference application with just 30% area increase.**

I. Introduction and Design motivation

Space systems experience an increased interest in machine learning (ML) and artificial intelligence (AI), however, the performance requirements of modern machine learning workloads cannot be satisfied by the existing space processors. Moreover, space systems cannot use COTS accelerators because they are not designed to withstand radiation and comply with numerous other requirements to be *qualified* for space use.

Since space qualification is an extremely costly process, space computing industries rely significantly on design reuse for the production of new, higher performing hardware, to minimize these costs. In particular, the qualification of a modified version of an already qualified processor costs only a fraction of the qualification of a new system, since requirements and validation tests from the original design can be reused.

Space processors have small size to minimise the area that is susceptible to radiation. They are manufactured with older, mature and reliable node processes e.g. 60nm or 45nm which are less dense than modern node processes, resulting in less available transistors than modern processors in the same die area, especially when the design is implemented with a radiation hardened cell library or incorporates fault-tolerance features such as triple modular redundancy (TMR). This also implies lower clock speeds compared to consumer products.

In order to increase the AI processing capabilities of modern space processors, in this paper we present SPARROW, an open source hardware design and compiler support [13] for the space qualified space processor LEON3 [14] used in numerous space missions. SPARROW uses a short vector, also known as Single Instruction, Multiple Data (SIMD), microarchitecture.

II. Module Design Principles and Architecture

SPARROW adds an extension module for the SPARC v8 based LEON3 space processor. It extends the integer pipeline with additional short vector operations with focus on AI applications, without any performance cost in the rest of the operations of the base processor. To guarantee this, we maintain the cycle time and the pipeline depth of the unmodified LEON3 and include the module in parallel to its ALU.

A key design decision of SPARROW is the reuse of the integer register file to reduce the area overhead of the design. We notice that in conventional architectures the vector register file consumes a significant portion of the vector design. Therefore, our decision results in a SIMD design which is at least 20-30% smaller than any other vector design targeting embedded processors, as we confirm later with our experimental results.

Another important element of SPARROW's design is its hardware-software co-design for AI processing. By analysing the literature it is apparent that one of the most significant operations in ML is the dot product [4], primarily used in matrix multiplication, one of the most common operations, since it is used both for the implementation of fully connected layers, as well as for convolutions. Thus, the optimization of this operation was the starting point of the design of SPARROW's architecture resulting in our two stage architecture design.

In addition, recent studies have shown that reduced precision of computations involved in machine learning inference operations to 8-bit integers provides minimal reduction in the inference accuracy. As a consequence, almost every architecture designed for inference workloads nowadays operate with 8-bit integers or even smaller bit widths [15]. This is compatible with our choice to reuse the integer register file, which consists of 32-bit registers and therefore can accommodate 4 values which can be processed in a SIMD manner. Moreover, the decision to work only with 8-bit widths allows our architecture to take implementation choices that otherwise would be impossible to be implemented without a significant impact on the cycle time or the area of the SIMD module.

In summary, in the first stage we add support for 13 SIMD arithmetic and bitwise operations (add, sub, mul, max, min, shift, move b, and, or, xor, nand, nor, xnor). In the second stage the number is limited to 4 reduction operations (sum, max, min, xor). Arithmetic operations can be signed or unsigned and can optionally use saturation.

978-3-9819263-6-1/DATE22/©2022 EDAA

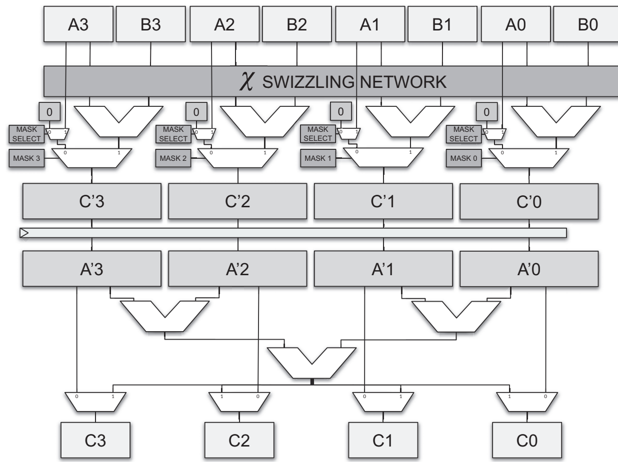

Fig. 1. Outline of the SPARROW module.

An outline of SPARROW is shown in Figure 1. In the first stage, the two input registers execute 4 operations in parallel in a conventional SIMD fashion, whilst being subject to traditional vector modifiers, such as swizzling or masking. The four components of the first stage result are either combined in reduction operations to produce up to a 32-bit result or passed to the module output. In both stages the result can be saturated and whenever one of the two stages is not used, it can be bypassed without penalty. Furthermore, when the result is used by the following instruction, the data can be bypassed directly to the integer pipeline saving a cycle.

A. The SPARROW Control Register

Unlike several SIMD extensions, SPARROW has full support for vector modifiers like masking and swizzling. These modifiers are not specified in the instruction, but they are instead configured by accessing a new special register `%scr` (SPARROW Control Register), which is accessed using the read and write instructions used for accessing the special registers in the SPARC v8 ISA. The 4 least significant bits in `%scr` identify the mask used, each bit corresponds to respective vector component. A set bit in the mask allows the component to be operated normally while a reset bit passes the masked value to the second stage. This masked value is selected using the 5th bit of `%scr`, and is, either the value 0, or the original value of the input component.

The next 16 bits of the SPARROW Control Register are used for swizzling control, eight for each source vector. Each pair of bits represents the component of the input register (0-3) which will occupy each position in the operation's source operand. The rest of the bits are reserved for future extensions.

B. SPARROW Stage 1: 2-Operand SIMD Operations

The first stage operations are those that require two source operands, the input registers after applying the reordering or component replication thanks to swizzling. Alternatively, the second operand can be an immediate value. Since all the bits for specifying the immediate in the ISA's integer instructions are used to select the opcode of each SPARROW stage, the immediate must be encoded in the 5 bits used for specifying the second input register. Thanks to our co-design approach, we encode the most frequently used values in AI inference applications, a common practice in embedded GPU ISAs [2]. Based on our analysis, in the AI domain such values are 0, 1, powers of two as well as their negatives and powers of two minus one, but the particular values are different depending on the instruction. The resulting operand vector contains the immediate value replicated in each of its components.

The operations are then performed in parallel for each pair of components. The result is stored in an extended length register (16 bits) to have higher precision if additional computation will be performed in the second stage. In most of the cases of AI algorithms the value is simply saturated in 8-bits. For this reason we offer a saturated version of each operation. For the rare cases that the user desires to preserve the upper 8-bits of each computation e.g. multiplication, this can be achieved by shifting the result. This allows to use SPARROW to perform also higher precision general purpose operations as it has been demonstrated in embedded GPUs [12].

If the selected operation for the first stage is a `nop`, only the second stage operations are executed. In this case the passed value is that of the first input register without swizzling but with the possibility to mask components.

The intermediate vector is the resulting vector after the first stage computations, if any, and with the mask applied. Depending on the mask selector bit, in `%scr`, the component is replaced with 0 or with the input vector component. In the later case, the sign is extended to match the larger size of the intermediate vector components.

One of the vital design characteristics of SPARROW is its multiplication implementation. Multiplication is traditionally a costly operation which requires multiple cycles to be executed and has a high area cost. Implementing a full 4-component SIMD multiplication in a single pipeline stage would impact the frequency of the original design and it would require to be split in multiple pipeline stages. However, since we are working on 8-bits operands, we implemented the logic which computes multiplication by explicitly performing a shift and addition for the 8 bits. This allows to fit the 8-bit multiplication with saturation within a single cycle without frequency impact.

C. SPARROW Stage 2: Reduction Operations

Second stage operations compute a single result by combining all the components of the intermediate vector. Reduction operations have a saturated and non-saturated version, too. The intermediate reductions are performed with an increased data width to avoid overflow. In the saturated case, the clamping is performed only to the final result, to avoid different results depending on the order of the components.

If there is no need for a reduction operation, a `nop` can be specified. In that case the result will be the output of the first stage with no modification whatsoever. Moreover, this result is bypassed to the integer pipeline and can be immediately used with no need to wait for an additional cycle.

III. Software Support for SPARROW

An important advantage of SPARROW compared to custom accelerators is the ability to reuse the existing qualified software stack of LEON3 i.e. the RTEMS real-time operating system or bare-metal space applications, which reduces both the cost and the effort of the development of a new compiler from scratch as well as its qualification cost later.

We added SPARROW support in the two most widely used compilers nowadays, gcc and llvm. We modified the `binutils` of Gaisler's `bcc-2.2.0` gcc-derivative compiler and the base `LLVM v13.0`. We program SPARROW in C, using inline assembly instructions wrapped in a C-preprocessor based library providing an interface similar to vector intrinsics for conventional SIMD extensions such as SSE or NEON.

IV. Evaluation

A. Hardware utilization

SPARROW has been synthesized and implemented for the Zynq UltraScale+ ZCU102 FPGA. Since there is no design from Gaisler's GRLIB GPL 2021.2 library for Zynq Ultrascale+, we have created a top-level design using the minimum components for a functional processor. This is the smallest, microcontroller-like configuration of the LEON3 with 8KB direct-mapped instruction and data caches, clocked at 100MHz. We also implemented the same design without caches for a fair comparison with another vector processor implementation for embedded systems in the literature [9].

According to the resource utilization results in Table I, SPARROW has a very small relative increase over LEON3 when it is implemented on an FPGA. There is an increment of only 26% over the baseline LEON3 design when implemented with caches and 30% without caches. In absolute terms, SPARROW uses only 2500 LUTs and 200 FF. In comparison, Johns and Kazmierski [9] present a vector unit implementation for a RISC-V embedded processor, which doubles the resource utilisation over its baseline. Unlike SPARROW, they implement vector operations up to 32-bit, not only 8-bit ones, but since they adhere to the RISC-V vector specification, a separate vector register file is need. As another indication of SPARROW's cost, the entire floating point unit (GRFPU) for LEON3 from Gaisler's GRLIB [6] costs 4600 LUTs and 2 BRAM blocks, and its area-optimised one (GRFPUlite) has comparable cost with ours (2000 LUTs and 2 BRAM blocks).

Notice that the relative area overhead of SPARROW does not change between the cache-featuring and the cache-less implementations on an FPGA, because by default BRAM blocks are used for the cache and the register files. That is, the key advantage of SPARROW, which is the integer register file reuse, is less evident in default FPGA implementations and more significant in ASIC implementations.

However, radiation-hardened-by-design space FPGAs such as the Xilinx Virtex 5 V5QV only offer radiation hardening for LUT-RAMs, while BRAM blocks only have ECC protection. Therefore, implementing the cache and the register files using LUT-RAMs can increase LEON3's reliability [14]. In order to evaluate SPARROW's resource overhead in that scenario and to give a rough indication of its relative area overhead in the case of an ASIC implementation, we have implemented our designs using LUT-RAMs, too. In this case the relative cost of SPARROW over the baseline LEON3 is 16% when the cache is present, and 25% otherwise. Note that our baseline processor is the smallest LEON3 configuration, so with a larger one, our relative hardware overhead is expected to be even smaller.

Moreover, in order to show the exact hardware savings of SPARROW thanks to the reuse of the integer register file, we implemented a version of both LEON3 and SPARROW using an extra register file. The cost of the extra vector register file would be around 310 LUTs, 240 LUT-RAMs and 43 FFs. This cost corresponds to 12% of the SPARROW cost in LUTs and 21% in FFs. Therefore, our hardware savings thanks to this decision are consistent with the overhead of the vector register file in ASIC implemented vector processors [1] [10].

B. Performance

We compare with [9] which is the vector processor design in the literature closest to ours, which implements the vector extension of RISC-V in a microcontroller. Although their design supports vector operations up to 32-bit per element (one 32-bit, 2 16-bit or 4 8-bit operations in a single cycle), they only evaluated their proposal with 8-bit programs: matrix multiplication, grayscale conversion of an RGB image and an edge detection filter. Matrix multiplication and the convolution filter are AI-related, while the grayscale conversion can be part of inference processing. The authors were kind enough to provide us with information and their software implementations in RISC-V assembly using vector extensions in order to perform a fair comparison. Their design runs at 50MHz which is half of SPARROW's and does not feature a cache.

Our results are shown in Table II. Due to space constraints we provide results only with the bcc toolchain, since llvm results are similar. For each experiment we report the number of processor cycles and the obtained speed-up compared to the LEON3 baseline. Our speed-up is higher than [9] for matrix multiplication (5.8×) thanks to our co-design, while the others are similar (grayscale 2.7× and filter 3.2×). Note that the results in [9] are obtained in simulation whereas ours on an FPGA. SPARROW obtains its performance boost over these 8-bit workloads using only a fraction of the hardware cost of [9] while operating at double frequency. When the cache is used, the execution time is reduced considerably in both SPARROW and baseline configuration, but their relative speedups are in the same ranges, so they are omitted for space reasons.

For an approximate comparison with the performance benefit provided by SIMD architectures such as ARM's NEON over their scalar baselines, we computed the 2nd degree polynomial equation presented in [8]. Jie and Kapre [8] mention that this operation executed with NEON with high loop trip counts over uncached 8-bit data provides a speed-up of 3.7× over the scalar code on an ARM A9 hardcore on an FPGA. SPARROW provides a speed-up of 4.5× with respect to LEON3 and 17.25× when saturation arithmetic is used for both designs.

TABLE I
RESOURCE UTILIZATION COMPARISON WITH RESPECT TO THE BASELINE FOR DEFAULT FPGA IMPLEMENTATION

	Zynq Ultrascale+ Available	LEON3		SPARROW	
		Cache enabled	**Cache disabled**	**Cache enabled**	**Cache disabled**
LUT	274080	9333 (3.41%)	8709 (3.18%)	11792 (4.3%)	11251 (4.11%)
LUTRAM	144000	292 (0.2%)	292 (0.2%)	292 (0.2%)	292 (0.2%)
FF	548160	6346 (1.16%)	6145 (1.12%)	6553 (1.2%)	6353 (1.16%)
BRAM	912	9.5 (1.04%)	4 (0.44%)	9.5 (1.04%)	4 (0.44%)

TABLE II
PERFORMANCE RESULTS WITHOUT CACHE

Program	**Data size**	**LEON3**	**SPARROW**	**Speed-up**
Matrix Mult.	120×120	117,310,797	12,632,882	9.3×
Grayscale	256×256	3,221,620	876,633	3.67×
Filter	256×256	39,019,342	11,855,218	3.3×
Cifar-10	32×32	3,784,906	649,618	5.83×
Polynomial	2048	88,145	19,537	4.5×

Although the previously presented benchmarks are good examples of ML applications, they don't exhibit any data reuse – which can further show the benefit of SPARROW – nor are relevant for space. For this reason we have ported a complex space relevant inference application based on CIFAR-10 from the open source GPU4S Bench benchmark suite [7] which has been recently released [11] [5]. Its layers include convolution, relu, max pooling and matrix multiplication over 32×32 images. The obtained speed-up is 5.5× for the entire inference chain with the cache and 5.8× without the cache.

It is worth noting that some SPARROW speedups are higher than 4× which is its short vector width. The reason is the additional reduction operations and saturation options, which implement multiple operations with a single instruction.

V. RELATED WORK

To our knowledge, the reuse of the integer file for short vector operations is a unique feature and a key contributor to SPARROW's low resource cost. Other low-cost vector architectures for embedded systems – but not AI-specific – are the series of works of J. Rose's [16] [17] and G. Lemieux's [3] [18] students. However, they only focus on designs FPGAs-based vector processors, exploiting FPGA features to achieve low cost. On the other hand, SPARROW targets both FPGA and ASIC implementations, as the original LEON3.

However, the most important feature of SPARROW is that it is created on top of an already qualified processor, which allows its incremental qualification at a low cost. On the other hand, none of the aforementioned technologies have any prior qualification credit and therefore their qualification for space can have a very high cost, if it is at all possible.

VI. CONCLUSIONS

SPARROW is a short SIMD microarchitecture which accelerates AI applications for the LEON3 space processor at low hardware cost, for both ASIC and FPGA implementations. This is achieved by reusing the integer register file which provides 30% smaller hardware overhead than conventional vector processors. Our design shows great results up to 9.3×, higher than similar vector designs for embedded microcontrollers [9] and embedded processors such as ARM's NEON, at a fraction of their hardware overhead compared to their baseline scalar designs. When saturation is needed, the speed-up can be over 17×. Finally, we obtained similar high performance in a space relevant inference benchmark, with more than 5× speed-up.

ACKNOWLEDGMENTS

We thank Johns and Kazmierski for providing us the RISC-V assembly version of their software for a fair comparison with [9]. This work was supported by the European Space Agency (ESA) through the GPU4S (GPU for Space) project, the Spanish Ministry of Economy and Competitiveness under grants PID2019-107255GB and FJCI-2017-34095 (Spanish State Research Agency / http://dx.doi.org/10.13039/501100011033), the HiPEAC Network of Excellence and a first prize in Xilinx's University Open Hardware Competition 2021 in the student category.

REFERENCES

[1] K. Asanović. *Vector Microprocessors*. PhD thesis, University of California at Berkeley, 1997.
[2] Broadcom. *VideoCore IV 3D Architecture Reference Guide*, 2013.
[3] C. H. Chou et al. VEGAS: Soft Vector Processor with Scratchpad Memory. In *FPGA*, 2011.
[4] D. Liu et al. PuDianNao: A Polyvalent Machine Learning Accelerator. In *ASPLOS*, 2015.
[5] D. Steenari et al. OBPMark (On-Board Processing Benchmarks) - Open Source Computational Performance Benchmarks for Space Applications. In *ESA/DLR/CNES European Workshop on On-Board Data Processing (OBDP)*, 2021. http://dx.doi.org/10.5281/zenodo.5638577.
[6] Gaisler. GRLIB IP Core Performance and Resource Utilization. https://www.gaisler.com/products/grlib/grlib_area.xls.
[7] I. Rodriguez et al. GPU4S Bench: Design and Implementation of an Open GPU Benchmarking Suite for Space On-board Processing. Technical Report UPC-DAC-RR-CAP-2019-1. https://www.ac.upc.edu/app/research-reports/public/html/research_center_index-CAP-2019,en.html.
[8] S. J. Jie and N. Kapre. Comparing Soft and Hard Vector Processing in FPGA-based Embedded Systems. In *FPL*, 2014.
[9] M. Johns and T. J. Kazmierski. A Minimal RISC-V Vector Processor for Embedded Systems. In *FDL*, 2020.
[10] C. Kozyrakis. *Scalable Vector Media Processors for Embedded Systems*. PhD thesis, University of California at Berkeley, 2002.
[11] L. Kosmidis et al. GPU4S: Major Project Outcomes, Lessons Learnt and Way Forward. In *DATE*, 2021.
[12] M. M. Trompouki and L. Kosmidis. Towards General Purpose Computations on Low-end Mobile GPUs. In *DATE*, 2016.
[13] M. Solé and L. Kosmidis. SPARROW source code repository, 2021. https://gitlab.bsc.es/msolebon/sparrow.
[14] M. W. Learn. Evaluation of the LEON3 Soft-Core Processor Within a Xilinx Radiation-Hardened Field-Programmable Gate Array. Technical Report SAND2012-0454, Sandia National Labs, April 2011.
[15] N. P. Jouppi et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*, 2017.
[16] P. Yiannacouras et al. VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors. In *CASES*, 2008.
[17] P. Yiannacouras et al. Fine-Grain Performance Scaling of Soft Vector Processors. In *CASES*, 2009.
[18] A. Severance and G. Lemieux. VENICE: A Compact Vector Processor for FPGA Applications. In *FPT*, 2012.