


Simodense: a RISC-V softcore optimised for exploring custom SIMD instructions

Philippou Papaphilippou , Paul H. J. Kelly, Wayne Luk
Department of Computing, Imperial College London, UK
{pp616, p.kelly, w.luk}@imperial.ac.uk

Abstract—Simodense is a high-performance open-source RISC-V (RV32IM) softcore, optimised for exploring custom SIMD instructions. In order to maximise SIMD instruction performance, the design’s memory system is optimised for streaming bandwidth, such as very wide blocks for the last-level cache. The approach is demonstrated on example memory-intensive applications with custom instructions. This paper also provides insights on the effectiveness of adding FPGA resources in general purpose processors in the form of reconfigurable SIMD instructions.

Index Terms—FPGA, RISC-V, softcore, SIMD, cache hierarchy, custom instructions, streaming, big data, sorting, prefix scan

I. INTRODUCTION

Modern general purpose processors (CPUs) support a wide range of single-instruction-multiple data (SIMD) instructions [1] as a way to accelerate applications that exhibit data-level parallelism. While there can be notable performance gains in certain applications [2], the instruction set extensions become bloated with overspecialised instructions [3], and sometimes it is difficult to express efficiently a parallel task using a fixed set of SIMD extensions [4].

As an alternative means to acceleration, FPGAs achieve unparalleled processing capabilities in specialised tasks [5], [6]. They have been getting attention for datacenter use, with numerous academic and industrial solutions focusing on processing big data [7]–[9]. However, the combination of specialisation and their placement in highly heterogeneous systems has high development and deployment costs. FPGAs are often left behind in terms of main memory bandwidth, leading to a bandwidth bottleneck for big data accelerators [7], [10], [11]. Also, high-end FPGAs are usually based on non-uniform memory access (NUMA) systems, and the communication techniques are mostly inconvenient and expensive. Examples include the high access latency of PCIe [12], HLS tools enforcing certain programming models, such as offloading before processing [13], and the need to reimplement cache hierarchies on FPGAs for improving the memory latency [11].

In combination with the openness of the RISC-V instruction set, and its support for custom instructions [14], this is a great time to start considering custom SIMD instructions on general purpose CPUs. Small FPGAs can be integrated, to implement custom instructions [3] and are demonstrated to improve the performance over existing extensions significantly [15], [16]. In the literature, the exploration of custom instructions on CPUs, and specifically SIMD, is rather limited.

In this paper, we present an open-source RISC-V softcore [17] that is optimised for exploring custom SIMD instructions. In order to achieve high throughput for streaming applications, the focus was given on the cache hierarchy and communication. The framework allows easy integration of custom vector instructions and evaluation in simulation and hardware. Finally, we evaluate examples of custom instructions and provide insights on introducing small FPGAs as execution units in CPUs. Our contributions are as follows:

- An open-source softcore framework to evaluate novel SIMD custom instructions, and design choices to maximise streaming performance (sections III,V-A).
- A set of non-standard instruction types to enable optional access to a high number of registers (section IV).
- A demonstration of the approach with novel SIMD instructions for sorting and prefix sum (section V-C).

II. RELATED WORK

Nguyen Dao et al. presented FlexBex [3], an open-source framework for embedding small FPGAs in a modified Ibex RISC-V core [18]. Although it provides a form of SIMD functionality, it is for embedded solutions and without a cache, and the operation is done on multiple 32-bit registers. This makes it limiting for benefiting memory-intensive applications. On a similar note, Ordaz et al. [16] developed a closed-source 128-bit wide SIMD engine shared between RISC-V cores [19]. Unlike our solution, the memory interface was much narrower than the SIMD engine, and both of these works mostly focus on the fabrication aspect and streaming performance.

There is a plethora of open-source softcores implementing the RV32IM or similar instruction set [20], from both industry and academia. Many have special features, such as out-of-order execution [21], [22], running Linux [23] and optimised arithmetic-logic units [24], [25]. Though, fundamental limitations, such as a 32-bit datapath, are usually hard-coded.

There are softcores that may have the potential to be adopted for exploring custom SIMD instructions [26], [27], but one major consideration is their use of higher-level languages and generators, which is known to be a problem for adoption [24].

There are also numerous FPGA-based vector processors and overlays, but are not suitable for exploring modular SIMD instructions for CPUs. Example limitations include the need to pass exclusive control to dedicated accelerator logic [28], fixed vector instruction sets [29]–[33], absence of base instruction set [34] or complete absence of instructions [35].

III. SOFTCORE MICROARCHITECTURE

The proposed softcore supports the RISC-V 32-bit base integer instruction set (RV32I v. 2.1), plus the “M” extension for integer multiplication and division [14]. The novel features of our approach include a series of design choices to: (1) enable high-performance for custom vector instructions and streaming applications; (2) allow efficient implementation on recent FPGAs by enhancing the block RAM (BRAM) organisation and the behaviour of the inter-chip communication.

A. Cache hierarchy optimisations

On the first level, there is an instruction cache (IL1) and a data cache (DL1), and on the second level there is a unified last-level cache (LLC). LLC responds to requests from both IL1 and DL1, and accesses the entire address range. It communicates in bursts to DRAM through an interconnect such as AXI. Figure 1 provides a high-level view of the data communication throughout the cache hierarchy.

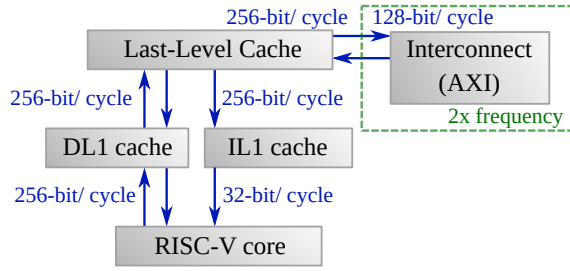


Fig. 1. Data movement in an example cache configuration

All caches use the writeback policy, with the exception of the IL1 cache, where writing is not needed. While DL1 and LLC are set-associative caches, the IL1 is direct-mapped for fast lookup of the next instruction, to avoid a stall on instruction hits, or contributing to the critical path.

The LLC is implemented in block memory (BRAM), to accommodate the high capacity of the last level. The IL1 is implemented in registers for a reduced latency, in order to be able to provide the successor instruction immediately on the next cycle, if it is a hit. The DL1 is implemented in BRAM by default, although changing the implementation directive to registers yields similar performance and utilisation results, due to its relatively low size.

At the set-associative caches (DL1 and LLC), each block can be allocated into multiple possible ways, represented by different parallel block RAM sections. The block replacement policy for these caches is not-recently-used (NRU). It uses one bit of meta-information per block [36], but closely resembles the Least-Recently-Used (LRU) eviction policy. The choice of a replacement policy can be crucial to the performance of streaming applications, due to the wide data blocks and the reduced cache space on the FPGA.

A series of design choices are presented for optimising the performance and applicability for our purposes.

1) *Level-1 block size*: One optimisation is to set the block size of the DL1 to be equal to the **vector register width**, such as 256 bits. This is because a wider block size would require an additional read on each write, from the cache of the higher level, so that the entire block becomes valid. When the data are from vector registers and are properly aligned, there is no need to wait for fetching that block on a write miss, because the whole block will contain new information.

The IL1 uses the same block size for easier arbitration between DL1 blocks, at the cache of the higher-level (LLC). Additionally, since IL1 is direct-mapped, using a wider-block than 32-bits is also beneficial to performance, as it can also be seen as a natural way of prefetching.

2) *LLC block size*: An important feature for increased streaming performance is **very wide blocks** for LLC, such as 8192-bit wide. This is in contrary to today’s CPUs with a 512-bit (64-byte) block size. The idea is that on write-back to/ fetch from main memory, it achieves a higher speed because of longer bursts. Longer bursts are shown to have significant impact on the overall throughput, such as in heterogeneous systems with AXI [37], and this is especially useful for streaming. Associating entire LLC blocks with bursts was a convenient and practical organisation choice, because of interconnect protocol limitations, such as for not crossing the 4KB address boundary in AXI [38].

3) *LLC strobe functionality*: A naive implementation of the LLC in BRAM, would be to read the (wide) blocks in their entirety in a single cycle, as in the DL1 case. However, BRAM is organised in chunks of certain width and length, such as 36-bit wide. With a LLC of just a single wide-enough block, the BRAM capacity of the FPGA can be exceeded, or stagnate timing performance. For this reason, the proposed LLC stores blocks in consecutive BRAM locations of **narrower size**. The tag array only stores the tags of entire blocks. Figure 2 visualises the indexing and strobe action of the LLC cache, where M is the number of ways, N is the number of sub-blocks in a way, and B is the number of sub-blocks per block.

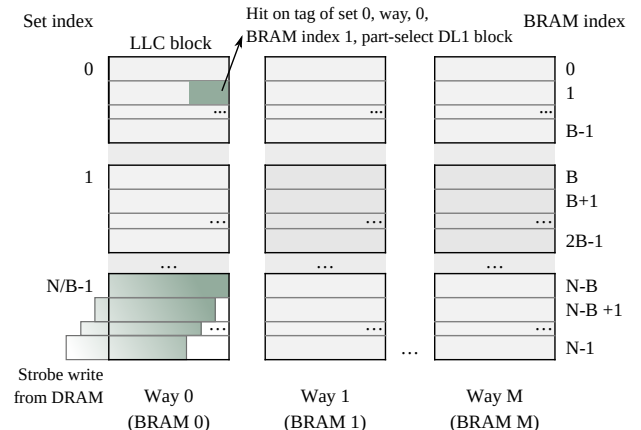


Fig. 2. Last-level cache organisation

There is no overhead in access latency by using sub-blocks, as it still takes a single cycle to read an I/DL1-sized block

from LLC. Another advantage of this technique is that, on fetch, the requested I/DL1 block can be provided before a read burst from DRAM finishes, since the LLC blocks are stored progressively.

4) *Flushing writes*: One consideration is that we need to be able to observe all memory modifications this softcore makes, at least when used as a co-processor. Thus, it also features a recursive *flush-writes* command. This essentially “cleans” all blocks upon request, one-by-one without erasing the caches, and all writes propagate to DRAM.

5) *Doubling the frequency of the interconnect*: In contrast to the timing characteristics of this softcore, as well with other well-known softcores [20], the operating frequency of the interconnect on FPGAs can be relatively much higher [39]. Given that the port data widths are rather narrow (e.g. 128 bits/ cycle), this directly impacts the throughput for streaming applications. This optional (platform-specific) optimisation involves setting double rate for the interconnect, to **emulate double data width** by fetching or writing twice per cycle, and saturate [37] the bandwidth more easily (see Figure 1).

B. Main core

The core has a single pipeline stage, even though more advanced instructions such as pipelined vector instructions have their own pipeline. Almost all instructions in RV32I consume 1 cycle and the result is available on the immediately next cycle. In practice, this has a similar effect to operand forwarding in pipelined processors, as consecutive dependent instructions are executed sequentially without stalls.

Having a **single pipeline stage**, so that most instructions complete in a single cycle, is useful for simplifying the dependency checks. The output of simple instructions such as add, addi, etc. is not tracked for dependencies. Of course, there are alternative approaches, but the current implementation mapped well in our evaluation platform and facilitated the SIMD functionality rather efficiently.

The load and store instructions are handled by the cache system independently in their own pipeline resulting in different amounts of latency. In order to monitor the data dependencies in multi-cycle instructions such as the “M” extension, SIMD tasks and loads, there is a ready bit per register that influences the stall decisions.

The implementation of the SIMD instructions follow a template that allows a variable pipeline length, abstracted through a *ready* signal for when the result is available.

IV. CUSTOM SIMD INSTRUCTIONS

In order to support adding and using SIMD instructions on Simodense, we propose new instruction types that refer to the vector registers and corresponding HDL (Verilog) code templates for implementing the instructions in hardware.

In addition to the RV32IM standards we include two experimental instruction types for supporting the custom SIMD instructions. The proposed instruction types I’ and S’ are variations of the types I and S respectively.

Type I’ provides access to the register operands of the I-type, that is one source and one destination 32-bit register. It also provides access to two source (*vrs1* and *vrs2*) and two destination (*vrd1* and *vrd2*) vector registers. Type S’ exchanges the space used by *vrs2* and *vrd2* for access to an additional 32-bit source register *rs2*. The proposed variations are summarised in Figure 3.

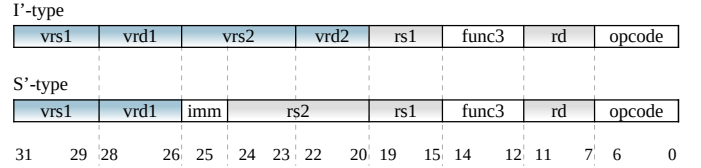


Fig. 3. Two variations of the I and S instruction types

As shown in Figure 3, the fields for the vector registers are 3 bits wide, which sets the maximum number of vector registers to 8. Vector register 0 **corresponds to 0**, as in the base set. It is useful for the proposed **many**-register instruction types, because not all register operands may need to be accessed at once. In software, this can be achieved with aliasing the unused operands with register 0, and was **not** supported in “V”, as vector 0 originally represents a register.

The official draft RISC-V “V” vector extension has not been followed, as it targets high-end/ hardened processors. Moreover, supporting hundreds of intrinsics and more vector registers would also be contrary to the idea of having small reconfigurable regions as instructions. An interesting feature in the “V” specification is the ability to chain vector instructions, hence the need for a high number of registers. Since our solution enables custom instruction **pipelines of arbitrary length**, a lower number of registers was considered satisfactory.

Custom instruction templates are provided inside the soft-core codebase for adding low-level user code for SIMD instruction implementations in Verilog [17].

V. EVALUATION

The exploration is divided in three parts according to the outcomes of each set of experiments: (V-A) justifies important design choices related to streaming performance, (V-B) shows that the performance is still acceptable when no SIMD instruction is used and (V-C) explores the behaviour and efficiency of example novel custom SIMD instructions.

The evaluation platform is Ultra96, which features the Xilinx UltraScale+ ZU3EG device. The FPGA on the device shares the same 2GB DDR4 main memory with the 4 ARM cores. The ARM cores run Linux, but the kernel address space is manually configured to end at the 1GB mark, so that the other 1GB is dedicated to the FPGA, that includes the softcore¹.

A. Design Space Exploration

The target application is memory copying (*memcpy()*), as its performance is (indirectly) detrimental to big data processing and related evaluations have a long history in HPC applications [40]. *memcpy()* here is manually implemented

¹Source available: <https://github.com/pphilippos/simodense>

with the custom instructions for load vector and store vector, instead of a library implementation using base registers. The data length is 256 MiB, in order to surpass the cache sizes.

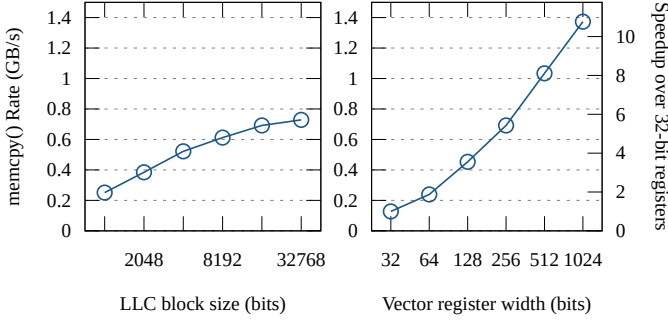


Fig. 4. Memcpy() read and write (bidirectional) throughput for different last-level cache block sizes (left) and vector register widths (right)

Figure 4 (right) illustrates the impact of the vector register size on `memcpy()`. The 1024-bit softcore achieved a `memcpy()` rate of 1.37 GB/s. Though, we opt for 256-bit (VLEN) registers, with a rate of 0.69 GB/s, as 512-bit and beyond seemed more challenging to route efficiently when incorporating more complex custom instructions. These designs used a 16384-bit-wide LLC block.

One other experiment (Figure 4 left) measures the impact of the block size in last-level cache (LLC). Wider LLC blocks seem to be a considerable contributor to memory performance, as they relate to the burst size. All implementations reached timing closure for a frequency 150 MHz, except the 1024-bit configuration that was clocked at 125 MHz. Table I summarises the selected baseline configuration for the remainder of the evaluation.

TABLE I
SELECTED CONFIGURATION

IL1		DL1		LLC		VLEN (bits)	f_{max} (MHz)		
sets	block (bits)	sets	ways	block (bits)	sub-blocks				
64	256 (=2KiB)	32	4 (=4KiB)	32	4	16384 (=256KiB)	32	256	150

B. Performance as a RV32IM core

In order to show that there is no significant bottleneck when compared with other non-SIMD cores, we overview some other works with a similar specification. Table II presents common benchmark metrics alongside previously reported numbers using FPGAs. Note that this is not for direct comparison, as each work used a different FPGA architecture, core/cache configuration and compilation environment.

Additionally, the performance of Simodense is measured for memory-intensive situations, without the use of SIMD. STREAM [40] is an established benchmark suite for measuring memory performance, especially in HPC. Figure 5 shows the obtained throughput in MB/s for each of the 4 kernels.

On the same FPGA, we place PicoRV32 [41], as a drop-in replacement that supports AXI (Lite). Although it was not designed for performance, it achieves high operating frequencies

TABLE II
INDICATIVE COMPARISON IGNORING SIMD

	DMIPS/MHz	Coremark/MHz	f_{max}	FPGA architecture
RVCoreP/radix-4 [24]	1.25	1.69	169	Xilinx Artix-7
RVCoreP/DSP [24]	1.4	2.33	169	Xilinx Artix-7
RSD/hdiv [21]	2.04	N/A	95	Zynq
BOOM/hdiv [21], [22]	1.06	N/A	76	Zynq
Taiga [19], [20]	>1	2.53	~200	Xilinx Virtex-7
Simodense	1.47	2.26	150	Zynq UltraScale+

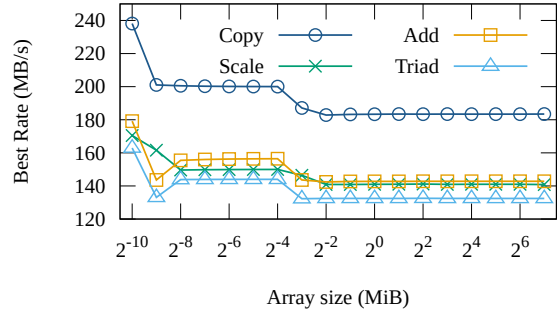


Fig. 5. Adapted STREAM benchmark results, no SIMD

(300 MHz in our platform), partly mitigating for its low IPC [20]. It does not have a cache, although this does not directly impact memory bandwidth, as the data reuse is practically zero. (The steps in Figure 5 are from reusing data from the initialisation). The results of PicoRV32 were 4.8, 3.6, 4.4 and 4 MB/s for Copy, Scale, Add and Triad consistently across the array size range. This makes our approach **38x** faster for Copy at 183.4 MB/s, or **144x** faster if we consider the 256-bit `memcpy()` performance. This also highlights the importance of optimising communication for streaming applications.

C. Custom SIMD instruction use cases

1) *Sorting (32-bit integers)*: Sorting is a widely applicable big data application. Existing SIMD intrinsic solutions are based on algorithms such as sorting networks [42], radix sort [43], mergesort [4], quicksort [2] and combinations.

The algorithm of our solution is merge sort, with the help of sorting networks [42] for introducing parallelism. The sorting networks are parallel and pipelinable algorithms for sorting a list of N values. In each parallel step there is a number of compare-and-swap (CAS) units, that collectively sort the entire input list, as the input moves along the network. Sorting networks were adapted for both software [2], [4] and hardware [44]–[47] solutions for sorting arbitrarily long input.

Figure 6 introduces the new custom instructions for sorting. In order to accelerate sorting in the softcore for arbitrary-sized input, a sorting network is used first to sort the entire list first in small chunks (function `c2_sort()`), as in [2]. Then, a traditional recursive merge sort approach is performed, but instead of merging each two sublists by comparing one element by one, it uses a parallel merge block (function `c1_merge()`). The merge block (the last $\log_2(N)$ layers of odd-even mergesort) is to merge two already-sorted lists together, as demonstrated in a numerical example of Figure 6. In our implementation, we add one more stage in the beginning of `c1_merge()` to enable

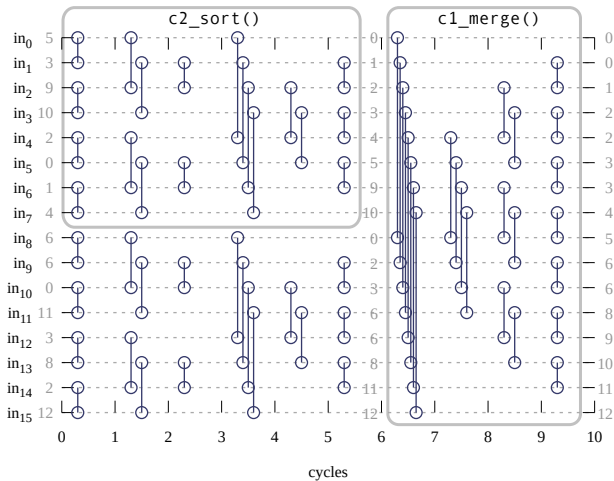


Fig. 6. Two new custom instructions based on odd-even mergesort

merging arbitrarily long lists progressively, and the algorithm is inline with the intrinsics merge algorithm [4].

The performance of the resulting mergesort is compared against non-vectorised code on the softcore, running at 150 MHz, as well as on the ARM A53 core, running at 1.2 GHz. The baseline is *qsort()* from C’s standard library. The obtained speedup is **12.1x** and 1.8 times over the *qsort()* on the softcore and ARM respectively, for 64 MiB random input. Comparison with more optimised code such as multi-threaded NEON-based for ARM, as well as other SIMD algorithms [2], [48] would be appropriate as future work.

2) *Prefix sum*: Another fundamental operator is prefix sum, and has numerous applications in databases, including in radix hash joins and parallel filtering [49]. The prefix sum for a series of values is the cumulative sum up to each value inclusive, (i.e. $out_k = \sum_{i=0}^k in_i$ for $k \in \{0, 1, \dots, N - 1\}$), where N is the number of inputs. Its serial implementation is trivial and easy for compiling efficient code.

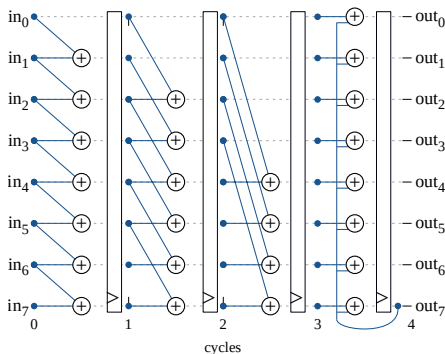


Fig. 7. New SIMD instruction for prefix sum

Figure 7 presents our custom instruction for the task. A widely-used algorithm for parallelising prefix sum is from Hillis and Steele [50], and is used in recent SIMD-based software [49]. The first $\log_2 N$ steps contain a pipelined version of this algorithm, plus one additional stage that adds the cumulative sum of the previous batch, that also happens to be the cumulative sum of the entire input up to that batch.

In this way, it can calculate the prefix sum of an arbitrarily long input in a pipelined and non-blocking way. For 64 MiB input, vectorising prefix sum yielded a speedup of **4.1x** over the serial version, though it had 0.4x the speed of ARM A53.

VI. DISCUSSION

One of the most useful insights from such exploration is about the reduction in the number of instructions and cycles required for a task. For instance, if we look at the *c2_sort* instruction, it is able to sort a list of 8 32-bit elements in 6 cycles. In contrast, a sorting network implementation of only 4 32-bit inputs in older Intel processors required 13 SIMD instructions and 26 cycles [4]. This **13x** and **4.3x** reduction of instructions and cycles respectively, while solving a bigger problem, is due to the unavailability of such specialised instructions. Even with the latest AVX-512 intrinsics [1], for each layer of compare-and-swap (CAS) units, a pair of separate instructions *min* and *max* are required, as well as a few calls of *shuffle* that permute the inputs for correct alignment [4].

Simodense aims at exploration. It is shown to perform well on real hardware, and does not assume 1-cycle-latency memories for instructions or data, a convention often found in other works [20]. However, sometimes dedicated FPGA accelerators are faster, such as with a sorter that achieves up to 49x speedup on the same platform [47]. This gap is expected and relates to the presence of instructions, in general. Many accelerators try to “internalise” processing, but this is not easily achievable on CPUs. Non-memory-intensive tasks can still achieve higher speedups, depending more exclusively on the FPGA’s capabilities.

CPUs operate at higher frequencies than FPGAs, and this is the reason why NEON-based memcpy() implementations can achieve high bandwidth on ARM [51]. Given that isolated or out-of-context FPGA designs can run much faster than when integrated in bigger systems [24], hardening of all communication could further close this gap.

VII. CONCLUSION

This softcore, in combination with the proposed optimisations and instruction types can be used to explore novel high-performance SIMD instructions. The provided methodology provides the ability to develop advanced SIMD instructions with a few lines of code, and minimise the instruction count for increased performance. It is demonstrated that custom SIMD instructions can provide an order of magnitude of speedup over serial implementations for memory-intensive applications. The availability of small reconfigurable regions as instructions in future generations of CPUs would be a more efficient use of silicon and processing cycles, and also simplify designs and solve the main memory bottleneck found in today’s FPGA-based datacenter accelerators.

ACKNOWLEDGMENT

This research was sponsored by dunnhumby. The support of Microsoft and the United Kingdom EPSRC (grant number EP/L016796/1, EP/I012036/1, EP/L00058X/1, EP/N031768/1 and EP/K034448/1), European Union Horizon 2020 Research and Innovation Programme (grant number 671653) is gratefully acknowledged.

REFERENCES

- [1] Intel (R), "Intel intrinsics guide." [Online]. Available: <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>
- [2] B. Bramas, "A novel hybrid quicksort algorithm vectorized using AVX-512 on Intel Skylake," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 8, no. 10, pp. 337–344, 2017.
- [3] N. Dao, A. Attwood, B. Healy, and D. Koch, "Flexbex: A risc-v with a reconfigurable instruction extension," in *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2020, pp. 190–195.
- [4] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient implementation of sorting on multi-core simd cpu architecture," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1313–1324, 2008.
- [5] H. Nakahara, Z. Que, and W. Luk, "High-Throughput Convolutional Neural Network on an FPGA by Customized JPEG Compression," in *28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2020, pp. 1–9.
- [6] K. Tatumura, A. Dixon, and H. Goto, "FPGA-Based Simulated Bifurcation Machine," in *29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 59–66.
- [7] M. Owaida, D. Sidler, K. Kara, and G. Alonso, "Centaur: A framework for hybrid CPU-FPGA databases," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2017, pp. 211–218.
- [8] K. Kara, Z. Wang, C. Zhang, and A. Gustavo, "doppioDB 2.0: hardware techniques for improved integration of machine learning into databases," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 1818–1821, 2019.
- [9] P. Papaphilippou and W. Luk, "Accelerating database systems using FPGAs: A survey," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 125–130.
- [10] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2014, pp. 151–160.
- [11] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras, "FPGA-based Multithreading for In-Memory Hash Joins," in *CIDR*, 2015.
- [12] J. Stuecheli, W. J. Starke, J. D. Irish, L. B. Arimilli, D. Dreps, B. Blaner, C. Wollbrink, and B. Allison, "IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI," *IBM Journal of Research and Development*, vol. 62, no. 4/5, pp. 8–1, 2018.
- [13] V. Kathail, "Xilinx vitis unified software platform," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 173–174.
- [14] A. Waterman and K. Asanovic, "The RISC-V instruction set manual, volume I: Unprivileged ISA document, version 20191214-draft," *RISC-V Foundation, Tech. Rep.*, 2020.
- [15] J. R. G. Ordaz and D. Koch, "soft-NEON: A study on replacing the NEON engine of an ARM SoC with a reconfigurable fabric," in *27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2016, pp. 229–230.
- [16] Ordaz, Jose Raul Garcia and Koch, Dirk, "A Soft Dual-Processor System with a Partially Run-Time Reconfigurable Shared 128-Bit SIMD Engine," in *29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2018, pp. 1–8.
- [17] P. Papaphilippou, P. H. Kelly, and W. Luk, "Extending the RISC-V ISA for exploring advanced reconfigurable SIMD instructions," *arXiv preprint arXiv:2106.07456*, 2021.
- [18] [www.lowrisc.org](https://github.com/lowRISC/ibex), "Ibex RISC-V." [Online]. Available: <https://github.com/lowRISC/ibex>
- [19] E. Matthews and L. Shannon, "Taiga: A new risc-v soft-processor framework enabling high performance cpu architectural features," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–4.
- [20] C. Heinz, Y. Lavan, J. Hofmann, and A. Koch, "A catalog and in-hardware evaluation of open-source drop-in compatible risc-v softcore processors," in *2019 International Conference on ReConfigurable Computing and FPGAs (ReConFig)*. IEEE, 2019, pp. 1–8.
- [21] S. Mashimo, A. Fujita, R. Matsuo, S. Akaki, A. Fukuda, T. Koizumi, J. Kadomoto, H. Irie, M. Goshima, K. Inoue *et al.*, "An Open Source FPGA-Optimized Out-of-Order RISC-V Soft Processor," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 63–71.
- [22] K. Asanovic, D. A. Patterson, and C. Celio, "The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor," University of California at Berkeley Berkeley United States, Tech. Rep., 2015.
- [23] J. Miura, H. Miyazaki, and K. Kise, "A portable and linux capable risc-v computer system in verilog hdl," *arXiv preprint arXiv:2002.03576*, 2020.
- [24] M. A. Islam, H. Miyazaki, and K. Kise, "RVCoreP-32IM: An effective architecture to implement mul/div instructions for five stage RISC-V soft processors," *arXiv preprint arXiv:2010.16171*, 2020.
- [25] E. Matthews, A. Lu, Z. Fang, and L. Shannon, "Rethinking integer divider design for fpga-based soft-processors," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 289–297.
- [26] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, 2020.
- [27] F. Kermarrec, S. Bourdeauducq, H. Badier, and J.-C. Le Lann, "LiteX: an open-source SoC builder and library based on Migen Python DSL," in *OSDA 2019, colocated with DATE 2019 Design Automation and Test in Europe*, 2019.
- [28] H.-C. Ng, C. Liu, and H. K.-H. So, "A soft processor overlay with tightly-coupled fpga accelerator," 2016.
- [29] P. Yiannacouras, J. G. Steffan, and J. Rose, "Vespa: portable, scalable, and flexible fpga-based vector processors," in *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, 2008, pp. 61–70.
- [30] J. Yu, G. Lemieux, and C. Eagleston, "Vector processing as a soft-core cpu accelerator," in *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, 2008, pp. 222–232.
- [31] A. Severance and G. Lemieux, "Venice: A compact vector processor for fpga applications," in *2012 International Conference on Field-Programmable Technology*. IEEE, 2012, pp. 261–268.
- [32] M. Johns and T. J. Kazmierski, "A minimal risc-v vector processor for embedded systems," in *2020 Forum for Specification and Design Languages (FDL)*. IEEE, 2020, pp. 1–4.
- [33] C. Chen, X. Xiang, C. Liu, Y. Shang, R. Guo, D. Liu, Y. Lu, Z. Hao, J. Luo, Z. Chen *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 *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 52–64.
- [34] A. Severance and G. G. Lemieux, "Embedded supercomputing in fpgas with the vectorblox mxp matrix processor," in *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 2013, pp. 1–10.
- [35] M. A. Sarkisla and A. Yurdakul, "Simdify: Framework for simd-processing with risc-v scalar instruction set," in *2021 Australasian Computer Science Week Multiconference*, 2021, pp. 1–10.
- [36] T. UltraSPARC, "Supplement to the ultrasparc architecture 2007," 2006.
- [37] K. Manev, A. Vaishnav, and D. Koch, "Unexpected Diversity: Quantitative Memory Analysis for Zynq UltraScale+ Systems," in *International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 179–187.
- [38] A. Holdings, "Amba axi and ace protocol specification," *Tech. rep.* 2011.
- [39] A. Interconnect, "v2. 1 logicore ip product guide," *PG059, Xilinx, December*, vol. 20, 2017.
- [40] J. D. McCalpin, "Stream benchmark," *Link: www.cs.virginia.edu/stream/ref.html# what*, vol. 22, 1995.
- [41] C. Wolf, "PicoRV32-a size-optimized risc-v cpu," 2019.
- [42] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 1968, pp. 307–314.
- [43] Intel (R), "Intel(r) integrated performance primitives: Developer reference, volume 1: Signal processing. (accessed on 11/01/2021)." [Online]. Available: <https://software.intel.com/content/www/us/en/develop/documentation/ipp-dev-reference/top/volume-1-signal-and-data-processing.html>
- [44] W. Song, D. Koch, M. Luján, and J. Garside, "Parallel hardware merge sorter," in *Field-Programmable Custom Computing Machines (FCCM), 24th Annual International Symposium on*. IEEE, 2016, pp. 95–102.

- [45] R. Kobayashi and K. Kise, "Face: Fast and customizable sorting accelerator for heterogeneous many-core systems," in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE, 2015, pp. 49–56.
- [46] E. A. Elsayed and K. Kise, "High-performance and hardware-efficient odd-even based merge sorter," *IEICE Transactions on Information and Systems*, vol. 103, no. 12, pp. 2504–2517, 2020.
- [47] P. Papaphilippou, C. Brooks, and W. Luk, "An Adaptable High-Throughput FPGA Merge Sorter for Accelerating Database Analytics," in *2020 30th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2020, pp. 65–72.
- [48] P. Papaphilippou, C. Brooks, and W. Luk, "FLiMS: Fast Lightweight Merge Sorter," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 78–85.
- [49] W. Zhang, Y. Wang, and K. A. Ross, "Parallel Prefix Sum with SIMD," *Algorithms*, vol. 5, p. 31.
- [50] W. D. Hillis and G. L. Steele Jr, "Data parallel algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.
- [51] "A.3.2. cortex-a8 data memory access," in *NEON Programmer's Guide*. arm, 2013, pp. 5–7. [Online]. Available: <https://developer.arm.com/documentation/den0018/a>