SALSA: A Domain Specific Architecture for Sequence Alignment

Lorenzo Di Tucci*†, Riyadh Baghdadi†, Saman Amarasinghe†, Marco D. Santambrogio*
*Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Milano, Italy,
†CSAIL - Massachusetts Institute of Technology, USA
{*lorenzo.ditucci, marco.santambrogio*}@polimi.it, {*lditucci, baghdadi,saman*}@csail.mit.edu

***Abstract*—The explosion of genomic data is fostering research in fields such as personalized medicine and agritech, raising the necessity of providing more performant, power-efficient and easy-to-use architectures. Devices such as GPUs and FPGAs, deliver major performance improvements, however, GPUs present notable power consumption, while FPGAs lack programmability. In this paper, we present SALSA, a Domain-Specific Architecture for sequence alignment that is completely configurable, extensible and is based on the RISC-V ISA. SALSA delivers good performance even at 200 MHz, outperforming Rocket, an open-source core, and an Intel Xeon by factors up to 350x in performance and 790x in power efficiency.**

***Index Terms*—Domain Specific Architecture, genomics, SALSA, Sequence Alignment, systolic arrays**

I. Introduction

The increasing amount of genomic data is empowering areas such as personalized medicine and agritech where drugs are selected based on the DNA of the patient and plants are engineered to be resistant to climate change [1]. In these regards, one of the most important tasks in the analysis of genomic data is sequence alignment, as it allows to find similarities and patterns among DNA sequences. The algorithms used in these fields are computationally intensive and the architectures on which they run need to process massive amounts of data [2]. General-purpose CPUs are inefficient in processing large genomic data due to their generality [3]. A better approach to analyze the big amount of genomic data, however, would be to create domain specific architectures that can process efficiently and in reasonable times huge data workloads, while keeping power consumption low [4]. The improvements of specific architectures in the genomic domain in terms of both performance and power consumption have been widely demonstrated in the state of art using both GPUs and FPGAs as hardware accelerators [3], [5]–[7]. GPUs offer massive performance and good programmability but they suffer from high power consumption. On the other hand, FPGAs offer a good ratio of performance over power consumption and unprecedented performance in this domain, with the drawback of harder programmability. In this paper, we propose the design and evaluation of SALSA, a Domain Specific Architecture (DSA) for sequence alignment. SALSA is designed to provide a good tradeoff between performance, power consumption, and programmability. SALSA is more programmable than FPGA implementations and consumes less energy than GPUs. The architecture of SALSA is based on Systolic Arrays (SAs) and is designed using Chisel HDL [8] It is programmable using RISC-V instructions [9], allowing users to add new instructions and, it is composed of multiple ALUs, where users can extend it by adding their own. While this paper is about accelerating sequence alignment applications using SALSA, we chose to base SALSA on SAs so that we can extend it, in the future, to other areas that can benefit from SAs too such as the ones where matrix multiplication, multiply and accumulate or convolution/correlation are involved [4]. The contribution of this paper is the design and evaluation of SALSA, a DSA for sequence alignments that is:

- Highly programmable: thanks to RISC-V ISA, with the possibility to extend the Instruction Set by introducing custom instructions;
- Customizable: it is possible to tune different parameters in the source code to generate architectures with different features like the number of processing elements, or the size of the memory port;
- Extensible: each Processing Element (PE) is composed of a general purpose ALU programmable via software, and more specific ALUs. The user can design and integrate custom ALUs inside SALSA.

II. Related Works

The State of the Art is dense of hardware acceleration of sequence alignment algorithms such as the Smith-Waterman or Needleman-Wunsch on CPU, GPU or FPGA [3], [10]–[12] with performance reaching billions of cell update per second (CUPS). In [3] an FPGA implementation of the Smith-Waterman algorithm is presented. The architecture can process single query to database alignment at a rate of more than 40 GCUPS, but is limited in processing a single version of the algorithm. [10], presents a non-reconfigurable heuristic accelerator integrable in Bowtie2 with notable performance increase with relation to well-known alignment algorithms. Darwin [13] is a co-processor for genomics exploiting FPGAs to speed up a non-optimal Smith-Waterman. This system as well is a hyper optimized version of an aligner that considerably differs from SALSA. The SAMBA accelerator [14] is based on systolic arrays and implements a Smith-Waterman algorithm with performance around 2 Mega CUPS, however, it is considered an ASIC, and users cannot modify the algorithm. The great majority of works done on this topic are implementations of a single algorithm that exploit the hardware to get the maximum performance, not allowing direct modifications. These implementations are hence hardly

978-1-7281-7445-7/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPSW50202.2020.00033

comparable with SALSA where the goal is to provide an architecture that is more general and capable of providing a tradeoff among performance, programmability, and easiness to use, so that the user can create different alignment algorithms. Furthermore, SALSA allows modifications at its microarchitecture, allowing the user to define specific ALUs to exploit the systolic array and the insertion of new customized instructions. SALSA cannot be considered as a revolutionary architecture,but it implements features which makes it unique.

III. SALSA

SALSA top level architecture is visible in Figure 1. It is mainly composed of a pipeline with 4 stages: Fetch & Decode, Dispatch, Load/Store and Compute. Each stage is associated with a hardware unit that solves the specific task. Fetch & Decode takes the instructions from the host processor and delivers them to the Dispatcher that can then decide to send the instruction to the Load/Store or directly to the Compute stage. The Load/Store unit has a direct connection to the memory controller to perform single, as well as multi-load/store requests. While the Load/Store unit handles a memory request, it asserts a busy signal and it is not able to process any new memory request until completion. The computation and the I/O are completely detached, hence if there is no dependency among the operations, the Load/Store and Compute Unit can work in parallel. Among all the stages there are fifos to buffer the data: whenever a fifo gets full, a stalling mechanism is activated until the fifo is able to store new data.For example, whenever the Load/Store unit is unable to store data at the same frequency of the Compute unit, it asserts a stall signal and the computation is stalled until the Load/store unit returns available. The size of the registers, the interfaces among the different modules as well as the interface among the Load/Store unit and the memory controller are tunable by changing specific parameters in the Chisel code. In this way, depending on the memory controller, it is possible to have 512 or 32/64 bits transactions. The value depends on whether we are interfacing SALSA with a cache memory, as in a scenario where SALSA is used as a co-processor sharing the same chip, or directly with the DDR where SALSA would have direct access to memory. The datapath of the entire architecture has been reduced to map the genomic scenario, yet the user can program the architecture to use a bigger datapath. SALSA instructions can take more than a clock cycle per stage, unlike traditional RISC pipelines that require 1 clock cycle per stage.

A. The Compute Unit and Processing Element

The Compute Unit Figure 1 is composed of a PE Dispatcher, a set of PE Sub-dispatchers, a linear array of tightly coupled PEs, a set of global registers accessible in read mode by all the processing elements, a fifo buffer and a set of PE Sub-collectors that communicate with a PE Collector. New data values coming from the Load/Store Unit can be stored to a Global Register, to the fifo or to one of the registers that are inside a PE. Furthermore, an element could be loaded and broadcasted to all the PEs or a specific register of a

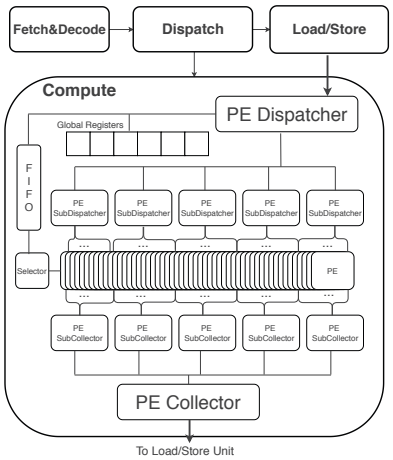

Fig. 1. Top Level architecture of SALSA with details of the CU.

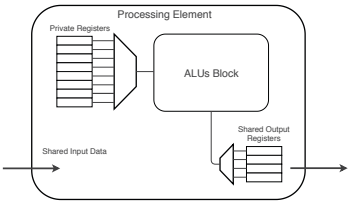

Fig. 2. Details of the PE inside the SA.

single PE. To route data values to internal register there is a *PE Dispatcher*. To avoid routing congestion, between the connection among the PE Dispatcher and the final register there is a PE Sub-dispatcher serving 32 PEs. The fifo buffer can be programmed to provide every clock cycle new data to the first PE of the SA that then, depending on how the computation is carried on, can be propagated each clock cycle to the neighboring PE. The fifo can provide a specific bitwidth of the input variable through a *data selector*, that fetches a new data element from the fifo and stores it into a local register. Every clock cycle, it selects the bitwidth chosen and sends it to the first PE. All the PEs are connected with shared registers and can be configured to perform the same operation in parallel each clock cycle. The operation can be executed using the general purpose ALU, or one of the more specialized ALUs that can be designed and integrated by the user. Custom ALUs speed up the execution of well-known computational patterns perfming multiple operation within the clock cycle. Every clock cycle, the SA can produce an output that is then collected by a series of *PE Sub-collectors* - each Sub-collector collects from 32 PEs - that communicate with a *PE Collector* that prepares the transactions for the Load/Store Unit. The PE, observable in Figure 2 contains all the logic to perform the mathematical operation and to move the output data to the neighboring PE. The PE can read the values shared from the PE on its left and can store the value that needs to be passed to the PE on the *Shared Output* registers. The General Purpose ALU can read and store in any register but the *Global*, as are

read only. At every point of the computation, it is possible to store data to the memory by storing values into the last N - where N is tunable at design time - registers into the private bank and by asserting those registers to valid. In this way, the *PE Sub-Collectors* know that there is data to store and collect the data for the Load/Store Unit. To make SALSA programmable and extensible, we have based its instructions on the extension created for the ROCC Interface [15], that uses the opcode space set aside for non-standard extensions in the RISCV ISA manual [9]. As an example, to perform a 64 bits load from the memory, it is necessary to specify the pointer of the variable using 40 bits, the destination PE, the destination register inside the PE and the type of the register. It can be shared, private or global. It is possible to use the ROCC instruction format to define the operations on the user-defined ALUs. In SALSA, we have designed 3 application specific ALUs to perform sequence alignment. We have ALUs to perform the Smith-Waterman Algorithm, the Needleman-Wunsch, and an affine version of the Smith-Waterman. These ALUs differs from the general purpose one as they solve a specific problem and are able to perform multiple operations within the clock cycle. To add specific ALUs, the user needs to specify which are the inputs and outputs of the ALU (see Figure 2) and specify the operations performed, that can be single, or multi-cycle, and the operation will be repeated in parallel by each PE in a lock-step way. Every time the new ALU has valid data, it will have to store it to the specific register and assert the valid signal, the architecture will take care of the rest. In this way, the users can define new algorithm exploiting the existing architecture and new instructions that exploit a combination of both specific and general purpose ALUs. For testing purposes, the ALUs are all physically implemented in SALSA. In the future we plan in making them dynamically reconfigurable.

IV. Experimental Settings and Results

A. Experimental Settings

SALSA has been designed using Chisel HDL [8]. To test the architecture, we have integrated SALSA with Rocket [16] an open source RISCV core. SALSA can be integrated in other systems, or adapted to be used as a standalone architecture. Rocket has been taped out in different commercial processes, reaching frequency over 1.65 GHz in IBM 45nm SOI [15]. Rocket infrastructure limits SALSA transaction to be at most 64 bits with no memory burst. This is limiting factor for memory bound applications, notice however that this is not a limitation of SALSA, but the testing infrastructure. SALSA has been generated with 160 PEs. 16 32-bits *Global* registers, the fifo that inputs the value to the first PE is capable of hosting 128 64-bits values. Each PE features 20, 32-bits *Private* registers, and the last 5 are reserved for the outputs, and 6 32-bits registers serving as *Shared output* ones. The synthesis and implementation of SALSA have been done with Firesim [17] 1.5 testing the designs on a single AWS EC2 F1 instance. The target frequency have been set to 200 MHz. We have compared the same applications using SALSA and Rocket on 4 examples, namely Smith-Waterman with constant and affine gap penalty, Needleman-Wunsch and MaxScore. All these algorithms benefit from a hardware implementation based on SA sharing a similar computational pattern, however, they can suffer from being memory bound. The maxScore, is instead compute bound as it just requires to store a singular value as output, and is a valid benchmark for SALSA real capabilities. The performance of SALSA has been also compared to an Intel Xeon E3 with a maximum frequency of 3.70GHz and power consumption of 90W [18]. As a software baseline, we used SeqAn 2.4 [19], a C++ optimized library, considered to be state of the art in sequence alignment. The power consumption for the system composed of SALSA and Rocket on Firesim, has been benchmarked using a specific API provided by AWS and is 11W considering the full FPGA board. The power efficiency has been obtained dividing the performance in GCUPS (Giga Cell Update per Seconds) by the power value described above. GCUPS is calculated as in [3].

B. Results

Table I provides a comparison in terms of performance of SALSA, Rocket and an Intel Xeon E3 processor in executing the 4 applications presented above. The first column reports the name of the algorithm and the second one the dataset size of the two sequences (No. of characters). The execution time of SALSA is reported in column three. We have considered that Rocket, can be synthesized at a frequency of more than 1 GHz, as expressed in [16]. Hence, in columns 4 and 5, we have reported the speedups of SALSA compared to Rocket, estimating for Rocket a frequency of 1 and 3 GHz respectively. The last column shows the speedup of SALSA against SeqAn running on an Intel Xeon E3, with a peak frequency of 3.7 GHz. The comparison shows how SALSA is faster than Rocket, in all the applications benchmarked, with the maximum performance speedup in the Smith Waterman Affine and the MaxScore algorithms. Note that the benchmark not only takes into consideration the computation time but also the time needed to store the results back to cache. Table I shows also how SALSA outperforms the CPU in terms of performance, even with these small datasets. To understand the benefits of SALSA compared to state of the art software implementations, consider that our implementation, as configured here and considering the short inputs provided, reaches performance up to 8 GCUPS in the max score example. SeqAn can obtain around 0.05 GCUPS with a single core. Under the assumption of independent sequences and a 40-cores machine, SeqAn would obtain 2 GCUPS. Except for the MaxScore algorithm that does not suffer from the memory badwidth limitation, and where SALSA is able to process data at the rate of around 8 GCUPS, the limiting factor is the cache size offered by Firesim, limiting the storage size to 64 bits per transaction and forcing SALSA to stall the computation. In these regards, as SALSA is configurable, is enough to change a parameter to obtain a wider channel to massively store the results. The tests performed with Firesim and Rocket are to demonstrate that even in a scenario with

TABLE I
PERFORMANCE AND POWER EFFICIENCY OF 4 APPLICATIONS EXECUTED ON SALSA, ROCKET AND USING SEQAN ON AN INTEL XEON E3 V2

Algorithm	Dataset	SALSA exec. time @ 200Mhz [μs]	Salsa Performance Improvement			Power Eddiciency[GCUPS/W]		
			w.r.t. Rocket @ 1 GHz	w.r.t. Rocket @ 3 GHz	w.r.t. CPU @ 3.7 GHz	SALSA	CPU	Improvement
MaxScore	32x64	0.8	**101.66x**	**33.89x**	**47.64x**	231.28	0.59	**389.8x**
MaxScore	64x128	1.045	**350.07x**	**116.69x**	**96.74x**	712.66	0.9	**791.5x**
SmithWaterman	32x64	16.84	**4.99x**	**1.66x**	**2.38x**	11.06	0.56	**19.5x**
SmithWaterman	64x128	37.01	**9.86x**	**3.28x**	**3.31x**	20.12	0.74	**27.13x**
NeedlemanWunsch	32x64	16.71	**4.85x**	**1.62x**	**1.99x**	11.14	0.68	**16.34x**
NeedlemanWunsch	64x128	37.00	**9.56x**	**3.19x**	**2.76x**	20.13	0.89	**22.61x**
SmithWatermanAffine	32x64	17.8	**8.62x**	**2.87x**	**2.55x**	10.45	0.5	**20.93x**
SmithWatermanAffine	64x128	39.48	**17.45x**	**5.82x**	**4.04x**	18.86	0.57	**33.09x**

such limitation, SALSA is capable of outperforming CPUs both in performance and power efficiency. Observing the end of Table I, in fact it is clear how SALSA is always more power efficient than the Intel processor with improvements up to 791 times in the MaxScore example demonstrating as a DSA like SALSA is beneficial not onl by means of performance, but also power efficiency. SALSA would benefit from an architecture different from Rocket. It could be used in a datacenter as a coprocessor, attached to the main processor with PCIe and with direct access with the main memory.

V. CONCLUSIONS

This paper presented the design and evaluation of SALSA, a DSA for sequence alignment. SALSA provides a good tradeoff between performance, power consumption and programmability. The Instruction Set Architecture (ISA) is based on RISC-V, the architecture has been designed using Chisel HDL and it is highly extensible, parametrizable and customizable, with the possibility to modify the microarchitecture by design specific ALUs and instructions. Comparisons in applications coming from the genomic domain, with a real-world processor called Rocket and SeqAn, run on an Intel Xeon E3 demonstrated how SALSA is up to 790 times more power efficient and up to 60 times more performant. We are confident that increasing the sizes of the input variables and filling the systolic array, the performance of SALSA would increase considerably, however, due to the limitation imposed by Rocket cache size, in this test we kept the input sizes relatively small being well aware of the memory limitations. Although SALSA is presented here as a DSA for sequence alignment, its architecture based on SA can be efficiently tuned to work well in other scenarios such as, for example, Neural Network (NN). Furthermore, thanks to its customizability, SALSA can be configured to be used in SoCs, on the Edge or even in data centers for HPC workloads.

REFERENCES

[1] G. S. Ginsburg and J. J. McCarthy, "Personalized medicine: revolutionizing drug discovery and patient care," *TRENDS in Biotechnology*, vol. 19, no. 12, pp. 491–496, 2001.

[2] G. England, "The 100,000 genomes project," *The*, vol. 100, pp. 0–2, 2016.

[3] L. Di Tucci, K. O'Brien, M. Blott, and M. D. Santambrogio, "Architectural optimizations for high performance and energy efficient smith-waterman implementation on fpgas using opencl," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017.

[4] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture." *Commun. ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[5] A. Zeni, M. Crespi, L. Di Tucci, and M. D. Santambrogio, "An fpga-based computing infrastructure tailored to efficiently scaffold genome sequences," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 333–333.

[6] L. Alawneh, M. Shehab, M. Al-Ayyoub, and Y. Jararweh, "A gpu-based smith-waterman approach for genome editing," in *16th International Conference on Information Technology-New Generations (ITNG 2019)*. Springer, 2019, pp. 347–352.

[7] H. Zou, S. Tang, C. Yu, H. Fu, Y. Li, and W. Tang, "Asw: Accelerating smith–waterman algorithm on coupled cpu–gpu architecture," *International Journal of Parallel Programming*, vol. 47, no. 3, pp. 388–402, 2019.

[8] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*. IEEE, 2012, pp. 1212–1221.

[9] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The risc-v instruction set manual. volume 1: User-level isa, version 2.0," CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES, Tech. Rep., 2014.

[10] Y.-T. Chen, J. Cong, J. Lei, and P. Wei, "A novel high-throughput acceleration engine for read alignment," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2015, pp. 199–202.

[11] S. Sarkar, T. Majumder, A. Kalyanaraman, and P. P. Pande, "Hardware accelerators for biocomputing: A survey," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. IEEE, 2010, pp. 3789–3792.

[12] P. D. Vouzis and N. V. Sahinidis, "Gpu-blast: using graphics processors to accelerate protein sequence alignment," *Bioinformatics*, vol. 27, no. 2, pp. 182–188, 2010.

[13] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 199–213.

[14] P. Guerdoux-Jamet and D. Lavenier, "Samba: hardware accelerator for biological sequence comparison," *Bioinformatics*, vol. 13, no. 6, pp. 609–615, 1997.

[15] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[16] Y. Lee, "RISC-V "Rocket Chip" SoC Generator in Chisel," https://riscv.org/wp-content/uploads/2015/01/riscv-rocket-chip-generator-workshop-jan2015.pdf, 2015, [Online; accessed 21-August-2019].

[17] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra *et al.*, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 29–42.

[18] "Intel® xeon® processor e3-1505m v5," https://ark.intel.com/content/www/us/en/ark/products/89608/intel-xeon-processor-e3-1505m-v5-8m-cache-2-80-ghz.html, accessed: 2019-09-13.

[19] A. Döring, D. Weese, T. Rausch, and K. Reinert, "Seqan an efficient, generic c++ library for sequence analysis," *BMC bioinformatics*, vol. 9, no. 1, p. 11, 2008.