Accelerating Chaining in Genomic Analysis Using RISC-V Custom Instructions

Kisaru Liyanage*†, Hasindu Gamaarachchi*†, Hassaan Saadat*, Tuo Li‡§, Hiruna Samarakoon*†, Sri Parameswaran¶
*The University of New South Wales, Sydney, Australia. †Garvan Institute of Medical Research, Sydney, Australia.
‡State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, China.
§Zhongguancun Lab, China. ¶The University of Sydney, Sydney, Australia.

***Abstract*—This paper presents a method for designing custom instructions tailored to RISC-V processors, focusing on optimizing the chaining step of Minimap2 (a software tool used to analyze DNA data emanating from third-generation sequencing machines). This custom instruction design involves employing an architectural template within the Rocket Custom Coprocessor (RoCC) unit of Rocket Chip, an open-source hardware implementation of RISC-V ISA, aided by a heuristic algorithm that facilitates extracting custom instructions from high-level C code targeting the proposed architectural template. Two types of instructions are created in this work: complex computational instructions; and instructions that load static data apriori so that these data are not repeatedly brought in from the memory. The resulting custom instructions integrated into Rocket Chip demonstrate a speedup of up to $2.4\times$ in the chaining step of Minimap2 with no adverse impact on the final mapping accuracy compared to the original software. The acceleration of Minimap2's chaining stage on a RISC-V processor enhances its portability and energy efficiency, making third-generation DNA sequence analysis more accessible in various settings.**

***Index Terms*—chaining, minimap2, genomics, read mapping, custom instructions, RISC-V**

I. Introduction

DNA sequence analysis is revolutionizing medicine, agriculture, etc., given its ability to understand the underlying details of living organisms. Minimap2 [1] is the gold-standard software-tool used in the latest generation (third-generation) of DNA sequence analysis [2]. Traditionally, DNA sequence analysis software (including Minimap2) is executed on high-performance computing (HPC) systems due to its compute-intensive nature. However, recent research has emphasized the need and importance of running DNA analysis on embedded systems to enhance portability, cost-effectiveness and energy-efficiency [3]. Such small systems enable the deployment of DNA analysis in remote areas [4], [5].

Minimap2's workflow consists of three main stages: seeding, chaining, and alignment. Owing to the high efficacy of the chaining stage, Minimap2 can deliver an accurate-enough DNA sequence alignment output even after the first two stages [1]. Consequently, the alignment stage can be considered optional when run-time and/or energy-efficiency is critical (i.e. Minimap2 can be executed without the "-a" flag). However, when executed on portable embedded platforms, the end-to-end run-time of Minimap2 running only the first two stages is still high compared to executions on HPC systems. This is elaborated in Fig. 1 (a), which compares the run-time of

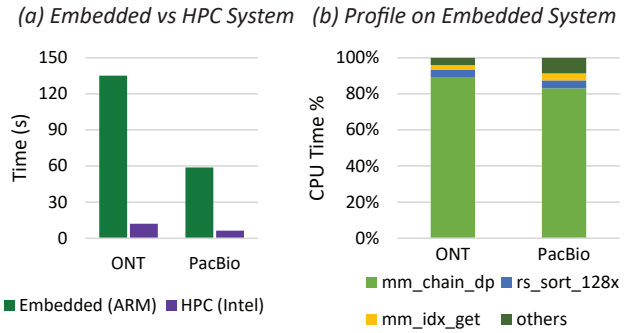

Fig. 1. Run-time profiling of Minimap2 on different systems. Note that an ARM processor was used instead of a RISC-V processor for profiling, due to the absence of a run-time profiling tool for RISC-V processors.

Minimap2 on an HPC system processor (Intel Xeon Gold 6154) vs an embedded system processor (ARM Cortex-A57) for two representative DNA sequencing datasets (ONT and PacBio datasets for Human NA24385 sample discussed in Section IV-B). Fig. 1 (a) shows that processing on an embedded processor makes Minimap2 nearly ten times slower than processing the same dataset on an Intel processor. *Therefore, there is a need to accelerate the chaining stage of Minimap2 for execution on embedded processors.*

In this paper, we aim to accelerate the chaining stage of Minimap2 using custom instructions in RISC-V. We focus on the chaining stage because our profiling experiment, as depicted in Fig. 1 (b), reveals that the *mm_chain_dp* chaining stage subroutine of Minimap2 becomes the bottle-neck on embedded processors. RISC-V is an open-source extendable ISA that is ideal for designing embedded system processors [6], while allowing customized instructions. Custom instructions can perform frequently performed operations in a given application tool faster while consuming less energy. They do add area, though this area in the context of the entire system is usually minimal. Therefore, in this work, we aim to extract custom instructions from the chaining step of Minimap2 for RISC-V processor.

The contributions of this work are:

- for the first time, Minimap2's chaining step is accelerated on a RISC-V based application-specific instruction set processor (ASIP) by up to $2.4\times$ with custom instructions, without affecting the final mapping accuracy.
- a novel architectural template is introduced for designing

979-8-3503-4859-0/DATE24/© 2024 EDAA

custom instructions, accommodating complex instructions beyond the regular two-input limit.
- a heuristic algorithm is presented to identify and map suitable code segments from C code to the architectural template.
- custom instructions are extracted from Minimap2's chaining step and implemented in HDL on Rocket Chip configured on Xilinx Zynq UltraScale+ FPGA.

II. Related Work

Researchers have recognized Minimap2 as a key third-generation DNA sequence analysis tool and have worked to improve its performance using various platforms, including modern CPUs, GPUs, and FPGAs on HPC systems. Multiple works have successfully sped up Minimap2's isolated chaining step on FPGAs [7], [8] and GPUs [8], [9], with one FPGA-accelerated chaining method being integrated into Minimap2 itself [10]. On modern Intel CPUs, the seeding, chaining, and alignment steps have been accelerated using AVX-2 and AVX-512 instructions and integrated seamlessly into Minimap2 [11]. Additionally, researchers have accelerated the alignment step of Minimap2 on FPGAs using GACT-X aligner [12] and the KSW2z algorithm [13]. To the best of our knowledge, no previous work has attempted to enhance Minimap2 with custom instructions for embedded processors.

Various custom instruction-based ASIPs have been developed for earlier generations of DNA sequence analysis tools and algorithms. An ASIP using the Xtensa re-configurable processor [14] has improved the performance of Kalign [15] by 30%. The Smith-Waterman (SW) algorithm [16] has been accelerated $2.87\times$ on Altera Nios II soft-processor with custom instructions [17], and up to $160\times$ [18] on the same soft-processor operating at 3.1MHz clock frequency. The work in [19] has extended the Xilinx MicroBlaze ISA to accelerate biological sequence processing using compressed indexes, achieving speed-ups between $3.1 - 4.5\times$. A multi-core Single Instruction Multiple Data (SIMD) ASIP [20] based on MicroBlaze has achieved over a $30\times$ speedup in the SW algorithm. An automated system has extended Tensilica's Xtensa processors [21] to achieve a $2.1\times$ acceleration of the BWA-aln.

In the broader context of application domains, various generalized and automated techniques have emerged to derive custom instructions from software represented as data flow graphs (DFGs). One approach in [22] identifies maximal speedup convex sub-graphs in the DFG for custom instruction implementation. Another work [23] has accelerated candidate instruction enumeration for large DFGs. In [24], instruction sets are chosen based on patterns with DFG mapping for latency reduction. Automation in [25] utilizes a DFG design space exploration heuristic and a compiler sub-graph matching framework. However, these methods do not incorporate state maintenance (e.g. with registers) within the Application-Specific Functional Unit (AFU). Consequently, their applicability is limited to smaller applications with simpler custom instructions that rely on a restricted number of inputs, typically just two inputs supported by the processor's register file.

In contrast to previous genomics related ASIP designing work ([14], [17]–[21]), we use a novel architectural template that resides within the AFU and a heuristic algorithm to allow extracting complex custom instructions from high-level C code, targeting the template. In comparison to DFG-based automated custom instruction extraction work ([22]–[25]), our approach enables instructions to maintain state across different instructions. This allows for more than the standard two inputs, with additional inputs coming from the "RoCC Registers Unit", a set of private registers that maintains the state. These extra inputs, combined with the two from the processor's register file, enables the extraction of complex instructions with over 10 inputs in Minimap2's chaining step.

III. Methodology

This section outlines the methodology for extracting custom instructions from high-level C source code of Minimap2's chaining step. Section III-A details the proposed architectural template, Section III-B explains the heuristic algorithm for custom instruction extraction targeting the template, and Section III-C presents the resulting custom instruction design for Minimap2's chaining step.

A. Hardware Architectural Template for Custom Instruction Extraction

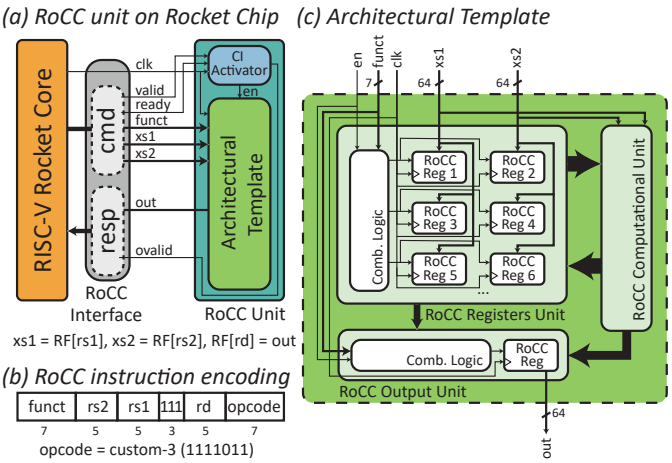

Fig. 2. Components of RoCC-based custom instruction hardware

Custom instructions for Minimap2 chaining step often require more than the conventional two inputs, and the specific input requirements can vary depending on the instruction. To tackle this challenge, we introduce a novel architectural template that extends the capabilities of traditional custom instruction implementation.

The architectural template is implemented within the Rocket Custom Coprocessor (RoCC) unit (Fig. 2 (a)). The RoCC unit interfaces with the Rocket Chip processor core via the RoCC interface. This interface includes the *cmd* sub-interface for input retrieval and the *resp* sub-interface for output transmission. The "CI Activator" module within the RoCC unit manages the template's activation. Custom instructions are

completed in a single clock cycle, ensuring efficient execution. The instruction format given in Fig. 2 (b) is used to access the architectural template within the RoCC unit and execute custom instructions on it. The instruction uses the opcode *custom-3* (binary encoding: "1111011") from the RISC-V ISA specification [26], and encodes two source registers *rs1* and *rs2* from the processor's register file (as opposed to values stored in RoCC Registers Unit) and a single destination register *rd*. Note that, the inputs needed for a particular instruction from the RoCC Registers Unit are directly encoded and are not operands of the custom instruction itself.

Fig. 2 (c) shows a detailed view of the architectural template. At its core, the template contains RoCC Registers Unit, a dedicated private memory unit. This unit can store additional inputs (discussed in Section III-B), augmenting the available inputs for custom instruction execution. To perform complex computations required by the custom instructions, the RoCC Computational Unit in the template concurrently accesses the stored inputs in RoCC Registers Unit (*RoCC Reg 1*, *RoCC Reg 2*, etc. in Fig. 2 (c)) alongside the regular two inputs retrieved from the register file of the Rocket Core (*xs1*, *xs2* in Fig. 2 (a) and (c)). This unique dual-unit input configuration effectively expands the number of input values accessible to a given custom instruction while adhering to the conventional two-input limitation of the already available processor register file. RoCC Computational Unit contains the hardware combinational logic for the C code segments extracted by the custom instruction extraction heuristic algorithm (Algorithm 1) explained later. Furthermore, the template includes RoCC Output Unit which is responsible for generating the appropriate output (*out* in Fig. 2 (a) and (c)) using values computed in the RoCC Computational Unit and/or the results stored in the RoCC Registers Unit. The output is connected back to the register file of the Rocket Core.

B. Custom Instruction Extraction Targeting Hardware Template

1) Overall Custom Instruction Extraction Process: In finding custom instructions, the first step is to profile the code and find loops within it, taking the most time. In the case of the chaining step, there is just one hot loop taking over 90% of the CPU time. Once the loop is identified, we extract C code blocks (code segments) to be converted to custom instructions. This step is done in Algorithm 1 (explained later). These code segments are manually converted to custom instructions.

Once we have these custom instructions, the original code is enhanced with the custom instructions, and the enhanced code's performance on the Spike ISA simulator [27] is recorded. Then, the identified loop in the previous paragraph is unrolled once and the C code blocks are again extracted. Custom instructions are again identified (similar to the description in the above paragraph). Once again, the enhanced code is simulated to examine for performance. If the performance is better, then instructions are used.

We continue to unroll iteratively until the performance is a maximum. Note, however, that unrolling it once gave the best performance for the chaining step.

Algorithm 1 Custom Instruction Extraction Algorithm

Inputs:
C code of hot loop
Line-by-line CPU time profiling information
Output:
C code segments for custom instructions

1: *Find candidate code segments in C code*
2: *Pick the non-overlapping set of code segments with the highest total CPU time*
3: *Sort code segments top to bottom based on their location in C code*
4: **for** *each candidate code segment in C code* **do**
5: *Identify and remove stationary inputs that can be pre-loaded to RoCC Registers Unit*
6: *Identify and remove inputs already available in RoCC Registers Unit*
7: $C_{inputs} \leftarrow$ *No. of remaining inputs*
8: **if** $C_{inputs} \leq 2$ **then**
9: *Select code segment for custom instruction implementation*
10: *Combine outputs within the maximum bit-width of 64 bits*
11: *Find the critical output and make it the output of instruction*
12: *Store non-critical outputs in RoCC Registers Unit*
13: **end if**
14: **end for**
15: **return** *Selected C code segments for custom instructions*

2) Instruction Extraction from Hot Loop Body: Algorithm 1 gives the heuristic algorithm used to extract custom instructions from the hot loop's C code. The first step involves identifying potential code segments within the C code that can be converted into custom instructions (Line 1). From the identified candidate code segments, the algorithm selects a set of segments with the highest total CPU time (based on line-by-line profiling information) that do not overlap with each other (Line 2). Non-overlapping selection ensures that the same part of the code is not optimized multiple times unnecessarily. The selected code segments are then sorted in the order they appear in the original C code, from top to bottom (Line 3).

Next, the algorithm iterates through each candidate code segment within the C code. For each segment, it performs a series of operations. First, it identifies and removes stationary inputs, that can be pre-loaded to the RoCC Registers Unit before the execution of the loop (Line 5). Next, it checks and removes inputs that are already available in RoCC Registers Unit (Line 6), stored by the previously chosen custom instructions as non-critical outputs (described later in this paragraph). The algorithm then calculates the number of remaining inputs, denoted as C_{inputs} (Line 7), and checks if this count is within the processor's register file input constraint of two (Line 8) as these remaining inputs have to be taken in from processor's register file. If the constraint is met, the code segment is chosen for custom instruction implementation (Line 9). If feasible, the outputs produced by the segment are combined within the maximum bit-width of the output (i.e. 64) to reduce the number of outputs (Line 10). Next, the critical output, immediately needed in the subsequent code, is identified and used as the custom instruction's output (Line 11). Non-critical outputs are stored in the RoCC Registers Unit for future use by subsequent custom instructions (Line 12).

After iterating through all the candidate code segments, the algorithm returns the set of code segments selected for custom instruction implementation (Line 15).

C. Custom Instructions Mapped to Architectural Template for Minimap2's Chaining

Employing the methods in Section III-B, with the source code of chaining step (i.e. *mm_chain_dp* function) of Minimap2 and the line-by-line fine-grained run-time profiling information obtained for the source code as inputs, custom instructions were extracted for chaining step of Minimap2. Fig. 3 shows the optimised C source code that includes the extracted custom instructions (custom instructions are highlighted in green).

The optimised C code contains ten custom instructions targeted for the proposed architectural template: five (instructions 0-4) purely for loading stationary inputs into the RoCC Registers Unit and five (instructions 5-9) for computations with non-stationary input loading. It should be noted that instructions 0-4 that load the stationary inputs to the RoCC Registers Unit were inserted at appropriate locations in the C code, before the the hot loop starts. Instructions 5-9 have been inserted inside the hot loop, replacing C code segments chosen by Algorithm 1.

The RoCC Computational Unit consisted of three sub compute modules to facilitate computations performed with instructions 5-9. The initial checks and computations performed by instructions 5 and 6 are performed with a single sub compute module. Within this sub compute module, several floating point multiplications are performed and performing these multiplications with floating point multipliers added significant area and time overhead to the digital circuit. To optimise these floating point multiplications, fixed-point multipliers were used. A second sub module was used for the $2\times$ parallel remaining score computation and *max_f*, *max_j* update performed by instruction 7. A third sub module was used for *max_skip* heuristic handling performed by instructions 8 and 9.

Line No.	Code
1	int32_t unused;
2	ROCC_INSTRUCTION(0, unused, max_dist_x, max_dist_y);
3	ROCC_INSTRUCTION(1, unused, bw, is_cdna);
4	ROCC_INSTRUCTION(2, unused, n_segs, gap_scale_fp);
5	ROCC_INSTRUCTION(3, unused, avg_qspan_scaled_fp, temp);
6	for (i = 0; i < n; ++i) {
7	ROCC_INSTRUCTION(4, unused, a[i].x, a[i].y);
8	uint64_t ri = a[i].x;
9	int64_t max_j = -1;
10	int32_t q_span = a[i].y>>32&0xff;
11	int32_t max_f = q_span, n_skip = 0, min_d;
12	while (st < i && ri > a[st].x + max_dist_x) ++st;
13	if (i - st > max_iter) st = i - max_iter;
14	uint64_t maxj_maxf = (max_j << 32) | max_f;
15	for (j = i - 1; j >= st; j -= 2) {
16	uint8_t init_checks_passed;
17	ROCC_INSTRUCTION(5, unused, a[j].x, a[j].y);
18	ROCC_INSTRUCTION(6, init_checks_passed, a[j - 1].x, a[j - 1].y);
19	if (!init_checks_passed) continue;
20	ROCC_INSTRUCTION(7, maxj_maxf, f[j], f[j - 1]);
21	uint8_t max_skip_break;
22	ROCC_INSTRUCTION(8, max_skip_break, t[j], unused);
23	if (max_skip_break) break;
24	if (p[j] >= 0) t[p[j]] = i;
25	ROCC_INSTRUCTION(9, max_skip_break, t[j - 1], unused);
26	if (max_skip_break) break;
27	if (p[j - 1] >= 0) t[p[j - 1]] = i;
28	}
29	max_j = (int32_t)(maxj_maxf >> 32);
30	max_f = (int32_t)maxj_maxf;
31	f[i] = max_f, p[i] = max_j;
32	v[i] = max_j >= 0 && v[max_j] > max_f? v[max_j] : max_f;
33	}

Fig. 3. Modified C code with custom instructions for chaining

IV. Experimental Setup

A. Implementation

To measure realistic run-time performance in terms of clock cycles, the custom instructions were implemented on hardware using Verilog HDL and were integrated into Rocket Chip. The customized Rocket Chip was synthesized onto the FPGA available on Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit, and performance gains were measured in terms of clock cycles. Fine-grained optimisations to improve the hardware area and performance were performed while meeting the timing constraints. Both the original Rocket Chip and the one with custom instructions met the critical path timing constraint of 10ns (100MHz clock speed). The Rocket Chip had a single-core, 64-bit, in-order processor that implements the RV64GC ISA and included a 16-kilobyte L1 data cache, utilizing a 4-way set-associative configuration with cache blocks of 64 bytes. Prior to hardware implementation, the Spike RISC-V ISA simulator was used for functional verification and instruction-level performance measurements. On both Spike and Rocket Chip, Minimap2 (the original version and the one with custom instructions) was run on RISC-V processor via a proxy kernel.

B. Performance Comparison

Performance benchmarks were conducted using a wide range of real third-generation DNA sequencing datasets (see Table I): different sequencing platforms, Oxford Nanopore Technologies (ONT) and Pacific Biosciences (PacBio), which are the two major third-generation sequencing platforms; and, different species, NA12878 and NA24385 human data and Zymo metagenomic data. For ONT data, benchmarks were performed on all combinations of currently existing pore-chemistries (R9.4.1, R10.4.1 4kHz and R10.4.1 5kHz) and available basecalling models (fast, high and super accuracy). Benchmarks were conducted by executing both the original Minimap2 and the optimised Minimap2 on 30,000 DNA sequencing reads from each dataset, with the seed and chain stages in Minimap2 enabled. For experiments done for human genome alignment, DNA sequencing reads were mapped against separate chromosomes extracted from hg38 human reference genome due to the 2 GB memory limitation on the Rocket Chip system on FPGA. For Zymo experiments, DNA sequencing reads were mapped against the full Zymo reference.

C. Accuracy Comparison

To validate the accuracy of our solution, 30,000 in silico ONT DNA sequencing reads were generated by basecalling

TABLE I
DETAILS OF THE DATASETS USED FOR BENCHMARKING

Species	Sample Name	Sequencing Platform	Chemistry	Data source / accession
Human	NA12878	Oxford Nanopore Technologies PromethION	R9.4.1	Sequence Read Archive: SRR22186402
Human	NA24385	Oxford Nanopore Technologies PromethION	R10.4.1 4 kHz	Sequence Read Archive: SRR23215366
Human	NA24385	Pacific Biosciences Revio	HiFi	In-house data
Metagenomic	Zymo	Oxford Nanopore Technologies GridION	R10.4.1 5 kHz	European Nucleotide Archive: ERR11759632

the ONT raw signal data generated using *squigulator* [28], [29] from the chromosome 10 of hg38 human reference genome. Basecalling was performed using *ONT Guppy* 6.5.7 through the *buttery-eel* [30] wrapper under the super accuracy model. These basecalled DNA sequencing reads were mapped to the chromosome 10 of hg38 human reference genome using the original Minimap2 and our optimised Minimap2. The results were evaluated using *paftools mapeval* utility.

V. RESULTS

A. Performance Comparison

The run-time of the chaining step in terms of clock cycles executed on the customized Rocket Chip (configured on the FPGA) for different datasets (Table I) are in Fig. 4. In Fig. 4, the purple bars (*minimap2*) represent the run-time taken by the chaining step of the original Minimap2 software, which does not utilize custom instructions. The green bars (*minimap2_rocc*) depict the run-time of the chaining step of the Minimap2 software incorporating custom instructions (i.e. running the modified C code in Fig. 3). Fig. 4 (a)-(d) compares the run-time performance for mapping human genome datasets. Fig. 4 (e) compares the run-time performance for mapping a Zymo metagenomic dataset.

In particular, Fig. 4 (a) is for sequencing data generated using ONT R10.4.1 4kHz chemistry. The labels *sup*, *hac* and *fast* on the x-axis refer to the three basecalling methods available for ONT data (explained in Section IV-B). The experimental results show that *minimap2_rocc* is $1.7\times$ faster than *minimap2* for all the cases. Fig. 4 (b) shows the performance comparison for data generated using ONT R9.4.1 chemistry (legacy ONT flowcell chemistry). The results obtained show that *minimap2_rocc* is $1.5\times$ faster than *minimap2*. Fig. 4 (c) depicts the performance for PacBio HiFi data, with *minimap2_rocc* outperforming *minimap2* by $1.9\times$. Note that for the comparisons in Fig4 (a)-(c), the DNA sequencing reads originated from chromosome 10 and were mapped to the chromosome 10 reference.

Fig. 4 (d) delves into a more granular analysis, comparing performance for mapping reads from the NA24385 sample (ONT R10.4.1 4kHz chemistry basecalled using super accuracy model) to each chromosome in the human genome. Notably, *minimap2_rocc* consistently showcases enhanced performance, achieving speeds $1.6-1.9\times$ faster than *minimap2* when mapping chromosome reads to their corresponding chromosome references.

Shifting the focus to non-human data, Fig. 4 (e) presents the run-time performance comparison for Zymo metagenomic data sequenced on ONT R10.4.1 5 kHz chemistry. This data is representative of the most recent sequencing technology update from ONT where the sampling frequency was increased to 5 kHz for enhanced sequencing accuracy (sampling rate was 4 kHz previously). *minimap2_rocc* showcases an increased speedup of $2.3-2.4\times$ faster compared to *minimap2* for all the three basecalling models, when aligning reads to the complete Zymo metagenome reference.

B. Accuracy Comparison

Table II compares the accuracy of the final mapping outputs generated by the original Minimap2 and optimised Minimap2. Original Minimap2 is running on the Rocket Chip configured on the FPGA without custom instructions (*minimap2*), while the optimised Minimap2 utilizes custom instructions to accelerate the chaining step (*minimap2_rocc*). The mapping

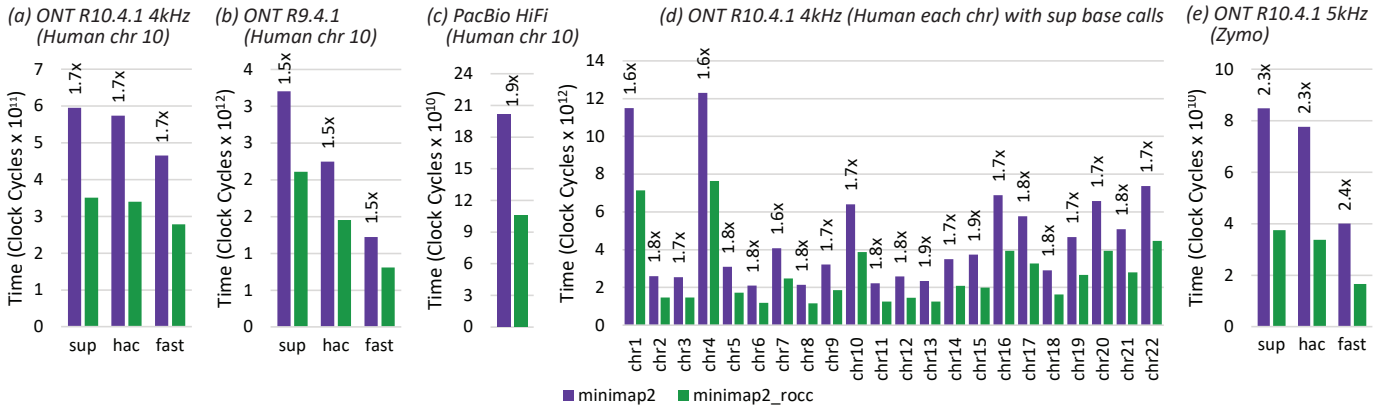

Fig. 4. Performance comparison performed on customized Rocket Chip configured on FPGA

TABLE II
FINAL MAPPING OUTPUT ACCURACY COMPARISON

MAPQ_T	*minimap2*		*minimap2_rocc*	
	No. reads MAPQ $\geq$ MAPQ_T	**No. wrong mappings**	**No. reads MAPQ $\geq$ MAPQ_T**	**No. wrong mappings**
60	29931	0	29931	0
3	36	1	36	1
0	33	22	33	22

MAPQ: Mapping Quality, MAPQ_T: Mapping Quality Threshold

TABLE III
FPGA RESOURCE UTILISATION OF CUSTOMIZED ROCKET CHIP

	LUTs	**FFs**	**DSPs**	**BRAMs**
Customized Rocket Chip	35832	18306	19	49
RoCC Unit	910	611	4	0

accuracy statistic in Table II are identical, thus, demonstrating that *minimap2_rocc* achieves the same level of accuracy as *minimap2*.

C. FPGA Resource Utilisation

Table III presents the FPGA resource utilization of the customized Rocket Chip, which includes the RoCC unit designed for Minimap2 chaining custom instructions. The LUT, FF and DSP usage of RoCC unit are 2.54%, 3.34% and 21.05% respectively when compared to the total area utilized by the customized Rocket Chip. Note that the RoCC unit does not utilise any BRAMs.

VI. CONCLUSION

This work contributes to advancing the third-generation DNA sequence analysis on embedded systems by accelerating the chaining stage of the gold-standard third-generation sequence analysis tool - Minimap2 through custom instructions on embedded RISC-V processors. Leveraging the open-source Rocket Chip that implements RISC-V ISA and an architectural template within the Rocket Custom Coprocessor (RoCC) unit, we introduced a heuristic algorithm to extract and implement custom instructions tailored for Minimap2's chaining step. Our FPGA-based experiments demonstrated a speedup of up to $2.4\times$ in the chaining stage while maintaining final mapping accuracy. This work not only showcases the potential of RISC-V for genomics but also minimizes the gap between high-performance and embedded systems, potentially advancing genomics research and healthcare applications.

VII. ACKNOWLEDGMENT

This research is supported by an Australian Government Research Training Program (RTP) Scholarship. We thank our colleague H. Dow for providing the VM configured with RISC-V development environment. We also thank our colleagues I. Deveson and Y. Perera for their continuous support during the research.

REFERENCES

[1] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 05 2018.

[2] H. Gamaarachchi *et al.*, "Featherweight long read alignment using partitioned reference indexes," *Scientific reports*, vol. 9, pp. 1–12, 2019.

[3] S. Y. Ko, L. Sassoubre, and J. Zola, "Applications and challenges of real-time mobile dna analysis," in *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*, 2018, pp. 1–6.

[4] J. Quick, P. Ashton, S. Calus, C. Chatt *et al.*, "Rapid draft sequencing and real-time nanopore sequencing in a hospital outbreak of salmonella," *Genome biology*, vol. 16, no. 1, pp. 1–14, 2015.

[5] S. L. Castro-Wallace *et al.*, "Nanopore dna sequencing and genome assembly on the international space station," *Scientific Reports*, vol. 7, no. 1, p. 18022, Dec 2017.

[6] E. Cui, T. Li, and Q. Wei, "Risc-v instruction set architecture extensions: A survey," *IEEE Access*, vol. 11, pp. 24 696–24 711, 2023.

[7] K. Liyanage *et al.*, "Cross layer design using hw/sw co-design and hls to accelerate chaining in genomic analysis," *IEEE TCAD*, pp. 1–1, 2023.

[8] L. Guo, J. Lau, Z. Ruan, P. Wei, and J. Cong, "Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between fpga and gpu," in *2019 IEEE FCCM*, 2019, pp. 127–135.

[9] H. Sadasivan, M. Maric, E. Dawson *et al.*, "Accelerating minimap2 for accurate long read alignment on gpus," *Journal of Biotechnology and Biomedicine*, vol. 6, p. 13–23, 01 2023.

[10] K. Liyanage *et al.*, "Efficient end-to-end long-read sequence mapping using minimap2-fpga integrated with hardware accelerated chaining," *Scientific Reports*, vol. 13, no. 1, p. 20174, Nov 2023.

[11] S. Kalikar, C. Jain, M. Vasimuddin, and S. Misra, "Accelerating minimap2 for long-read sequencing applications on modern cpus," *Nature Computational Science*, vol. 2, no. 2, pp. 78–83, Feb 2022.

[12] C. Teng *et al.*, "Adapting the gact-x aligner to accelerate minimap2 in an fpga cloud instance," *Applied Sciences*, vol. 13, no. 7, 2023.

[13] A. Zeni *et al.*, "On the genome sequence alignment fpga acceleration via ksw2z," in *2023 IEEE ISCAS*, 2023, pp. 1–5.

[14] A. Gkogkidis *et al.*, "Exploration study on configurable instruction set for bioinformatics applications," in *2019 PACET*, 2019, pp. 1–6.

[15] T. Lassmann *et al.*, "Kalign–an accurate and fast multiple sequence alignment algorithm," *BMC bioinformatics*, vol. 6, no. 1, pp. 1–9, 2005.

[16] T. Smith *et al.*, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[17] J. Chiang, M. Studniberg *et al.*, "Hardware accelerator for genomic sequence alignment," in *2006 International Conference of the IEEE Engineering in Medicine and Biology Society*, 2006, pp. 5787–5789.

[18] I. T. Li, W. Shum, and K. Truong, "160-fold acceleration of the smith-waterman algorithm using a field programmable gate array (fpga)," *BMC Bioinformatics*, vol. 8, no. 1, p. 185, Jun 2007.

[19] N. Sebastio, P. Flores, and N. Roma, "Optimized asip architecture for compressed bwt-indexed search in bioinformatics applications," in *HPCS*, 2014, pp. 527–534.

[20] N. Neves, N. Sebastião, D. Matos *et al.*, "Multicore simd asip for next-generation sequencing and alignment biochip platforms," *IEEE Transactions on VLSI Systems*, vol. 23, no. 7, pp. 1287–1300, 2015.

[21] V. Gnanasambandapillai, J. Peddersen, R. Ragel *et al.*, "Finder: Find efficient parallel instructions for asips to improve performance of large applications," *IEEE TCAD*, vol. 39, no. 11, pp. 3577–3588, 2020.

[22] K. Atasu, L. Pozzi *et al.*, "Automatic application-specific instruction-set extensions under microarchitectural constraints," *International Journal of Parallel Programming*, vol. 31, no. 6, pp. 411–428, Dec 2003.

[23] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2004, p. 69–78.

[24] J. Cong, Y. Fan *et al.*, "Application-specific instruction generation for configurable processor architectures," in *ACM FPGA*, 2004, p. 183–189.

[25] N. Clark, H. Zhong, and S. Mahlke, "Automated custom instruction generation for domain-specific processor acceleration," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1258–1270, 2005.

[26] "riscv-spec-v2.2.pdf," https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf, (Accessed on 09/01/2023).

[27] "Spike, a risc-v isa simulator," https://github.com/riscv-software-src/riscv-isa-sim, (Accessed on 09/08/2023).

[28] H. Gamaarachchi, J. M. Ferguson, H. Samarakoon *et al.*, "Squigulator: simulation of nanopore sequencing signal data with tunable noise parameters," *bioRxiv*, pp. 2023–05, 2023.

[29] H. Gamaarachchi, H. Samarakoon, S. P. Jenner *et al.*, "Fast nanopore sequencing data analysis with slow5," *Nature biotechnology*, vol. 40, no. 7, pp. 1026–1029, 2022.

[30] H. Samarakoon, J. M. Ferguson, H. Gamaarachchi, and I. W. Deveson, "Accelerated nanopore basecalling with slow5 data format," *Bioinformatics*, vol. 39, no. 6, 2023.