Heterogeneous Genome Compression on Mobile Devices

Liangliang Chen[1], Xiaotong Wang[1], Ziqiang Liao[1] and Juncong Lin[1(✉)]

[1] School of Informatics, Xiamen University, Xiamen 361104, China
jclin@xmu.edu.cn

Abstract. In recent decades, sequencing techniques have undergone rapid development, leading to a significant reduction in costs. Concurrently, the popularity of personalized medicine and portable devices has surged. These advancements have precipitated an exponential increase in genomic data volume, posing formidable challenges in terms of storage and transmission. Compression emerges as a viable solution to address these challenges. To date, the bulk of research within this domain has concentrated on refining compression algorithms. However, investigations into hardware acceleration of these algorithms have been scant, primarily focusing on Graphics Processing Units (GPUs) or Field-Programmable Gate Arrays (FPGAs). Given the prevalence of multiprocessor system-on-chips (MPSoCs) in portable devices, there is a pressing need to design algorithms that can leverage computing resources effectively, enhancing both performance and energy efficiency, which are crucial for portable devices. To address these challenges, our study introduces a novel heterogeneous compression algorithm that significantly advances the performance and energy efficiency of genomic data processing on mobile devices. By implementing a dual-stage pipeline approach for Gzip and leveraging both OpenCL and OpenMP, our framework optimally utilizes the disparate computational resources of MPSoCs. The empirical results underscore a robust performance enhancement, achieving an average increase of 15.7% in data processing speed to conventional CPU-based methods. This substantial leap in efficiency alleviates the computational burdens typically associated with mobile genomic applications.

Keywords: Genomic Data, Compression, Heterogeneous Algorithms.

1 Introduction

With ongoing advancements in sequencing technologies and a marked decrease in sequencing costs, personalized medicine is evolving rapidly, paving the way for tailored treatments. This evolution generates an ever-increasing volume of genomic data, presenting formidable challenges in long-term storage—given that much of this data is unique and may be revisited in the future. Additionally, the transmission and access of large data volumes require substantial network bandwidth.

For example, the data produced by the 1000 Genomes Project in its first six months surpassed the volume of sequence data accumulated over 21 years in the NCBI

GenBank database [1]. Hence, efficient data compression becomes essential to support this growth.

Moreover, the introduction of portable nanopore sequencing devices, such as Oxford Nanopore Technologies' MinION device [2], has made DNA and RNA sequencing feasible in the field or clinic. The deployment of these devices across various contexts highlights their significant potential. However, a corresponding portable solution for genomic analysis in such scenarios remains underdeveloped, limiting the full realization of these devices' potential. Recent efforts have been made to develop a mobile toolkit for ONT-sequencing analysis [3]. Yet, these efforts primarily focus on the toolkit's functionality, with little attention to maximizing the computing resources of portable devices to enhance toolkit performance.

Meanwhile, computing platforms have universally shifted towards multi-core architectures. Portable devices, for example, often employ Mobile Application Processors as MPSoCs, incorporating a mix of general-purpose cores, GPUs, DSPs, non-programmable accelerators, and FPGAs. This diversity offers the potential for high performance, provided software can effectively harness all available resources by utilizing the most power-efficient cores. However, developing algorithms that run efficiently on such heterogeneous MPSoCs is challenging, and the development of heterogeneous multi-core systems has not been paralleled by advancements in software support, making programming for these complex systems a formidable task. For further details on the influence of architectures on bioinformatics, refer to section 2.2.

In this paper, we realized a heterogeneous version of Gzip on smartphones to offer a portable yet powerful method for genomic data compression. To optimize the computational resource utilization on smartphone platforms, we have deconstructed the Gzip algorithm into two segments, integrating them within a pipeline to improve throughput. Furthermore, we allocated Gzip's computational tasks to heterogeneous computing units, each selected based on its compatibility with the specific computational requirements. By adjusting the voltage frequency of these computing units, we achieved a balanced execution time across different pipeline stages, thus enhancing performance and managing energy consumption and thermal output. Our experimental findings indicate that this heterogeneous implementation of Gzip yields, on average, a performance increase of approximately 15.7% compared to normal CPU execution. To facilitate reproducibility and further research, our source code is available under the MIT License at https://github.com/RookieTars/HeterogeneousGenomeGzip. Although we have already developed a prototype program, we are still working on a mobile application that incorporates our algorithm. This application is designed to enhance the usability of the algorithm and to facilitate further experimental extensions.

2 Related Works

2.1 Genome Data Compression

Genome data compression algorithms can be broadly classified into two categories: those designed for raw Next-Generation Sequencing (NGS) data, including FASTQ and SAM/BAM files, and those for compressed assembled genome data, such as FASTA

files. Within the realm of assembled genome compression, algorithms can be further divided into reference-based and non-reference-based methods. Reference-based methods, such as DNAzip [4], HiRGC [5], iDoComp [6], GeCO [7], and GDC [8, 9], utilize a reference genome to enhance compression efficiency. Conversely, certain algorithms [8–10] are specifically optimized for genome collections, leveraging similarities across entire collections to maximize redundancy reduction. On the other hand, non-reference-based methods [11–14] compress genome sequences directly, without the aid of external data. This approach was predominant until the introduction of referential genome compression algorithms by Brandon et al. [15]. Typically, reference-based methods outperform non-reference-based ones by utilizing similarities within the same species' genomes. For a deeper understanding of developments in this field, readers are encouraged to consult additional literature [16].

Genome data compression has primarily been implemented on CPUs, focusing mainly on the compression algorithms themselves. However, GPU adaptations such as CULZSS [17] and efforts by Patel et al. [18] have not met expected either compression ratios or performance due to inherent complexities. Limited research has been conducted on designing algorithms for heterogeneous processors. Our study introduces a novel approach to genome data compression on MPSoCs, optimizing task allocation across different processors to enhance computational efficiency.

2.2 Hardware Accelerated Bioinformatics

Despite advancements in CPUs through Moore's Law and multi-threading, there is still a significant gap between the processing capabilities of CPUs and the demands of bioinformatics analyses. This has led to the use of various hardware architectures, such as high-performance computing (HPC), FPGAs [19], multi-core devices, and GPUs [20], to accelerate bioinformatics algorithms. Meanwhile, as genome data acquisition devices become smaller [21], the need for efficient data compression on resource-limited portable devices grows, despite challenges in parallelizing compression schemes and coordinating between processors within an MPSoC. Our work focuses on the efficient use of heterogeneous processors within an MPSoC for compression tasks, avoiding the complexity of parallelizing on a single processor.

3 Background

3.1 Heterogeneity of MPSoC

Heterogeneity in MPSoCs manifests in two primary forms: performance heterogeneity and functional heterogeneity. Performance heterogeneity arises when cores sharing the same instruction-set architecture (ISA) but exhibiting different power-performance characteristics are integrated within the same system. This variation often results from distinct micro-architectural features, such as the difference between in-order and out-of-order cores. An example of this is the ARM big.LITTLE architecture, which, for instance, may combine high-performance, out-of-order ARM Cortex-A15 cores with energy-efficient, in-order ARM Cortex-A7 cores. On the other hand, functional

heterogeneity involves integrating cores of vastly different functionalities (and ISAs) on the same die, common in embedded systems. Such MPSoCs may include general-purpose CPU cores, GPU cores, Digital Signal Processor (DSP) blocks, and various hardware accelerators or Intellectual Property (IP) blocks (for instance, video encoders/decoders, imaging processing units, etc.). This approach caters to the performance needs within a tight power budget by assigning workloads to the most suitable processing units, whether they are GPUs, DSPs, or dedicated accelerators.

Fig. 1. Architecture of Snapdragon® 855

Recent mobile platforms, including smartphones, tablets, and wearable devices, incorporate both types of heterogeneity to optimize performance and energy efficiency. For instance, the Qualcomm® Snapdragon® 855 [22], utilized in our research, showcases this with its array of CPU cores: one high-performance Qualcomm® Kryo™ 485 Prime core operating at 2841 MHz, three medium-performance Qualcomm® Kryo™ 485 Gold cores at 2419 MHz, and four low-performance Qualcomm® Kryo™ 485 Silver cores at 1785 MHz, alongside an Adreno™ 640 GPU, as depicted in Fig. 1.

3.2 Dynamic Voltage-Frequency Scaling

The performance, energy consumption, and thermal behavior of a processor core are significantly influenced by its operating frequency. Research has demonstrated that the execution time T of a task on a core exhibits a relationship with frequency, represented as:

$$T = \alpha / f + \beta, \tag{1}$$

where f denotes the frequency, and α, β are constants derived via interpolation from runtime data at two extreme frequencies. Moreover, power consumption escalates quadratically with frequency increase:

$$P = ACV^2f + P_i, \tag{2}$$

in which A stands for the activity factor, C the capacitance, V the voltage, f the frequency and P_i the idle power consumption at the specified frequency. Additionally, the core's temperature rises more sharply as the frequency increases, though this relationship is less precisely defined. To manage core performance, energy efficiency, and prevent thermal throttling, Dynamic Voltage-Frequency Scaling (DVFS) algorithms adjust the operating frequency and corresponding voltage based on the current workload, representing one of the most widely adopted strategies.

4 Methods

To thoroughly explore the advantages of heterogeneous computing on MPSoCs, we have developed a specialized algorithm tailored for the Gzip compression method, taking into account its unique characteristics. Specifically, we segment the data into blocks and allocate them to idle processors. These blocks are then processed in parallel by different components in a pipeline structure. Ultimately, all processed blocks are merged to produce the final compressed output.

4.1 Distribute Tasks Transparently

Coordinating all cores to effectively process data blocks poses a significant challenge. Open Computing Language (OpenCL) [23] is an open standard designed to facilitate the development of parallel applications across heterogeneous multi-core architectures, including CPUs, GPUs, DSPs, and FPGAs. Device vendors supporting OpenCL are responsible for providing the necessary runtime software and compilation tools that enable the execution of OpenCL programs. Unfortunately, current mobile System-on-Chips (SoCs) often lack OpenCL support for ARM CPU cores [24]. While many academic projects might opt for an open-source solution like FreeOCL [25], it unfortunately does not offer robust support for the latest Android versions. Consequently, we have chosen to implement a uniform task scheduling system using OpenMP [26] for shared-memory multiprocessing on CPUs, while utilizing OpenCL for other components when possible, which makes the idle computing components get various kinds of tasks from the queue transparently possible.

4.2 Pipeline Organization

To enhance throughput, constructing an efficient pipeline for the Gzip algorithm presents another significant challenge. Gzip consists primarily of LZ77 and Huffman coding, each with unique characteristics that complicate full utilization of all components to accelerate the pipeline.

LZ77 is a dictionary-based compression algorithm that identifies the longest matching string within a sliding window and a lookahead buffer. It operates on the principle of reducing data redundancy by replacing certain substrings that appear elsewhere in

the data with references to their previous occurrences. According to the experiments we carried out, implementing LZ77 on GPUs using multi-threading has resulted in poorer performance compared to single-threaded CPU execution. This underperformance is due to several factors:

- **Data Transfer Limitations:** The slow speed of data transfer between the GPU and memory significantly affects the efficiency of GPU parallel computations, as continuous data retrieval from memory for comparison is required.
- **Overheads in GPU Processing:** While GPUs possess a greater number of processing units, the extensive one-by-one comparison operations required by LZ77 introduce considerable overheads in processor communication and collaboration.
- **GPU Inefficiency in Branch Operations:** GPUs are not well-suited for branch operations. The SIMD architecture within GPUs requires that each thread perform similar operations simultaneously. The extensive conditional logic in LZ77 means that different threads may need to perform different operations, causing some threads to wait for others, thus reducing parallel computing efficiency. In contrast, CPUs can better handle this issue through branch prediction.

Building on a similar approach, we attempted to optimize the Huffman encoding segment of Gzip using GPUs. Huffman encoding in Gzip compresses the array of length and distance pairs resulting from LZ77 by counting the frequency of each character, sorting characters from least to most frequent, constructing a Huffman binary tree, and performing variable-length encoding from the leaf nodes to the root node. Characters that occur more frequently are assigned shorter codes, while less frequent characters receive longer codes, thus minimizing the overall data size for effective compression.

We decomposed the Huffman encoding algorithm into three distinct phases: frequency counting, Huffman tree construction, and code allocation. The most time-consuming aspects of this stage are the frequency counting and the code generation phases. We implemented these phases on the mobile GPU and assessed their performance relative to CPU-based implementations with Qualcomm® Snapdragon® 855 [22].

Table 1. Execution times of Huffman encoding's time-consuming phases on CPU and GPU

Size of the tested file	Frequency counting on CPU (ms)	Frequency counting on GPU (ms)	Code generation on CPU (ms)	Code generation on GPU (ms)
4.47 KB	0.007760	7.569063	0.006562	0.142031
32.0 KB	0.036042	8.636772	0.041511	0.123646
73.7 KB	0.074844	8.573855	0.102448	0.432917
8.51 MB	9.103074	11.291147	7.580574	4.102240
10.3 MB	12.900314	10.694636	9.128022	4.927865

The results presented in Table 1 indicate that when dealing with smaller input files, using GPUs does not offer a performance advantage over CPUs; however, as the size of the input files increases, the performance benefits of GPUs become increasingly significant, even surpassing those of CPUs. These outcomes suggest that Huffman

encoding could benefit from parallelization, but the input size should be sufficient. Because the gains may be offset by additional overheads such as data sharing and synchronization when the data processed by each thread is too small. Nonetheless, the input for the Huffman algorithm in Gzip, being data processed by LZ77's dynamic sliding window, typically does not exceed 32KB. Therefore, executing the Huffman algorithm on a GPU does not achieve the intended performance optimization.

Based on the analysis and tests conducted, due to the complex data dependencies and branching operations inherent in the Gzip algorithm, utilizing GPUs does not effectively optimize algorithm performance and compression efficiency. We executed the entire Gzip compression algorithm on the CPU and proposed a heterogenous architecture approach to better leverage the various cores of the CPU.

As shown in Fig. 2, we organize the cores of the MPSoC into two pipeline stages. As data sharing among different types of cores are usually larger than that among same cores (the sharing of data between CPU and GPU are much larger), our main principle for pipeline organizations is to cluster the same type of cores together. Correspondingly, we divide the Gzip algorithm into two stages: LZ77 is whole put on the super core, while the Huffman coding is put on the stage consist of four small cores and three big cores taking input from buffer, which LZ77 generated. The reason is that LZ77 is pure serial job. And the Huffman coding is independent and suitable for SIMD parallelization, meaning that we can divide file into blocks and dispatch them independently and transparently to idle cores in the stage.

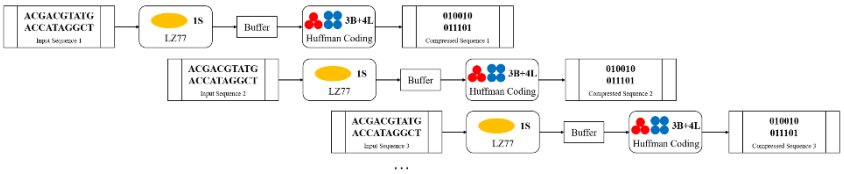

Fig. 2. Pipelined Gzip algorithm

We finally determine the optimal frequency for each type of core to increase core utilization using DVFS, by addressing the following energy minimization problem:

$$\arg\min_{f_S, f_B, f_L} \omega_t \cdot \|T_{LZ} - T_H\|^2 + \omega_p \cdot \left(E_S T_S^2 + E_B T_B^2 + E_L T_L^2\right), \tag{3}$$

where T_{LZ} and T_H are the processing time of two pipeline stages (LZ77 and Huffman coding) respectively. E_S, E_B and E_L are the energy consumption of super core, big core, and little core respectively while T_S, T_B and T_L similarly stand for processing time for cores. ω_t and ω_p are weights which we attribute equal constants to performance and power efficiency. Since LZ77 runs on super core and Huffman encoding runs on big and little cores, we have $T_S = T_{LZ}$ and $T_B = T_L = T_H$ to simplify the problem. In addition, target frequencies relate to processing times and energy consumptions with Equation (1), Equation (2) and power Equation (4):

$$P = W / t \tag{4}$$

5 Results and Discussions

We implemented heterogeneous compression methods with OpenCL for the GPU part and OpenMP for the CPU part. We conducted a series of experiments using a Google Pixel® 4 smartphone with Qualcomm® Snapdragon® 855 SoC [22] as the hardware platform for convenience. In this section, we show how heterogeneous design of the algorithms can improve in running time and power efficiency. In addition, we explored the reason that had influence on energy efficiency. We compare our heterogeneous algorithm with its normal CPU counterparts. We emphasis more on the running time and energy efficiency, both of which are important for portable devices and also our main motivations of this work.

5.1 Test Data

For the pipelined Gzip algorithm, we prepared two groups of genome data. The first group is of small size, comprising yeast [27] and fruit fly genomes. We downloaded three yeast strains (s.paradoxus, yeast_S288C and yeast_AWRI796) and D. melanogaster (dm6) genomes in FASTA format, with sizes not exceeding 1 GB. The third group contains considerably larger datasets, with three files containing human gene data in different testing ages (hg17, hg19 and hg38), one with dog gene data inside (canFam6) and one containing mouse gene data (mm39). The data pertaining to these advanced species all exceed 1 GB in size. The detail sizes of these files are documented in Table 2.

Table 2. Details of test data

Size group	Species	Gene dataset	File size
Small	Yeast	s.paradoxus	11.2 MB
		yeast_S288C	11.6 MB
		yeast_AWRI796	7.46 MB
	Fruit fly	dm6	139.8 MB
Large	Mouse	mm39	2.59 GB
	Dog	canFam6	2.19 GB
	Human	hg17	2.93 GB
		hg19	2.98 GB
		hg38	3.05 GB

5.2 Performance and Energy Efficiency Improvements of Heterogeneous Gzip

For the Gzip algorithm, we compared the heterogeneous algorithm with normal CPU algorithm on both performance and energy consumption.

Results of performance is shown in Fig. 3. Comparing the processing times for nine gene datasets, it is evident that the heterogeneous algorithm enhances the compression

performance of gene datasets 15.7% on average compared to the normal CPU algorithm with the same compression ratio. For smaller gene datasets, such as the yeast and fruit fly genomes, which are less than 1 GB in size, the performance improvement ranges between 19.6% and 21.3%. In contrast, for larger gene datasets like the human and mouse genomes, exceeding 1 GB, the improvement is more modest, averaging between 9.3% and 12.5%. Therefore, the algorithm significantly boosts performance for smaller gene datasets, while the gains for larger gene datasets are relatively smaller. This discrepancy may be attributed to longer runtime associated with larger files leading to increased temperature and subsequent performance degradation of the SoC. To enhance performance for large-scale datasets, further refinement of the algorithm's design and implementation is required, considering cooling solutions in future research.

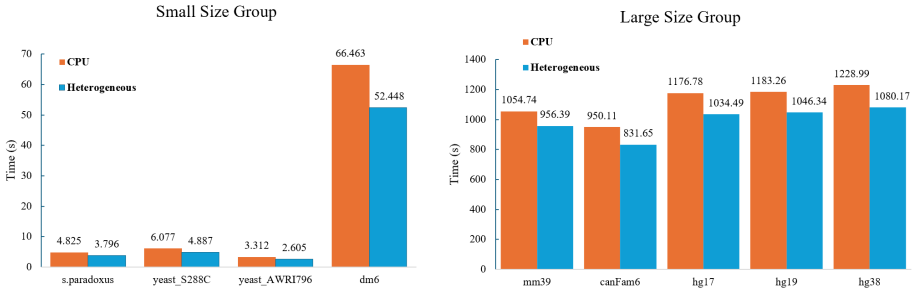

Fig. 3. Test results of performance on small and large size genome data

Energy consumption test results from Fig. 4 reveal significant differences in the energy efficiency improvements provided by Gzip's heterogeneous algorithm across various file sizes. For smaller data files, such as the yeast and fruit fly genomes, energy consumption unexpectedly increased, with an increase ranging between 7.8% and 17.4%, 13.7% on average. Conversely, for larger genomic files, such as those of mouse, dog, and humans, there was a minor reduction in energy consumption, falling within the range of 1.5% to 3.1%.

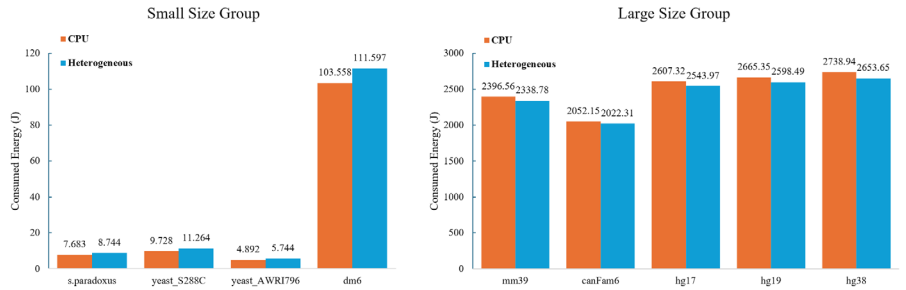

Fig. 4. Test results of energy consumption on small and large size genome data

5.3 Exploration for the Reason of Extra Energy Consumption and Discussion

To explore why our method consumes more energy than traditional CPU approaches when processing small size group, and to further investigate the user experience enhancements it offers, we conducted experiments focusing on temperature—a critical factor for mobile devices.

We selected one typical file each from both small and large size groups and monitored the SoC temperature changes while compressing these files on the device. During the experiments, the initial SoC temperature of the device was maintained at 30°C, and the device operated under ambient room temperature. We tested the dm6 and hg19 genomic files, and the results are shown in Fig. 5.

For the dm6 file, the SoC temperature of our method remained consistently higher than that of the traditional CPU method from the start to the end of the program execution, and the temperature difference increased over time. Our method consumed more energy with small size files, which we infer was largely due to increased heat generation. In contrast, for the hg19 file, our method's SoC temperature was mostly higher than the traditional CPU's, but the temperature difference was not too significant. The time difference in energy consumption between the traditional CPU method and ours was considerable, unlike with the dm6 file, which provided the traditional CPU method with more time to consume energy in generating heat, offsetting the energy efficiency benefits of the lower temperature of large size files.

On mobile devices, balancing performance, energy consumption, and temperature is crucial. In our approach, the energy efficiency and temperature performance of small files did not match those of traditional CPU methods. For scenarios where energy efficiency and thermal performance are prioritized, it may be advisable to forego the minor performance gains in compressing small files. Finding the balance between performance, energy consumption, and temperature in our method is essential, particularly in identifying the threshold at which energy gains are optimized for file size. This will be a focus of our future work.

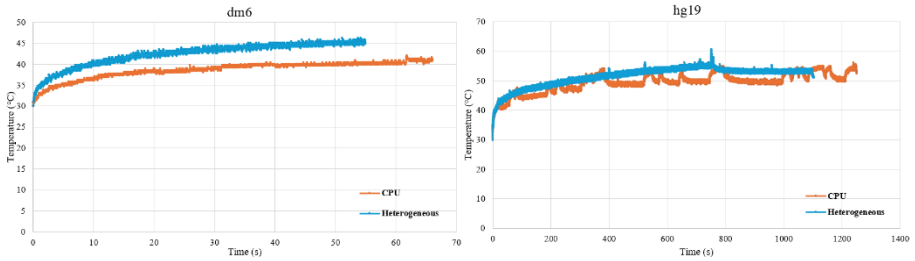

Fig. 5. Test results of temperature on typical small and large size genome data

6 Conclusion

In this paper, we investigate how the heterogeneous design of compression algorithm can improve the performance in both running time. Specifically, we design a pipelined

Gzip compression method by organizing processors into a two-stage pipeline, dividing the algorithms into two steps, mapping the two steps to the two stages of the pipeline, and finally adjusting the frequency of processors to make the processing time of each stage be consistency.

Experiments show that the heterogeneous Gzip algorithm gains average 15.7% improvement against the normal CPU counterparts in running time.

Acknowledgments. We sincerely appreciate the support of the National Natural Science Foundation of China (62077039) and the Fundamental Research Funds for the Central Universities (20720230106) to this work.

References

1. Pennisi, E.: Will Computers Crash Genomics? Science. 331, 666–668 (2011). https://doi.org/10.1126/science.331.6018.666.
2. Jain, M., Olsen, H.E., Paten, B., Akeson, M.: The Oxford Nanopore MinION: delivery of nanopore sequencing to the genomics community. Genome Biol. 17, 239 (2016). https://doi.org/10.1186/s13059-016-1103-0.
3. Samarakoon, H., Punchihewa, S., Senanayake, A., Hammond, J.M., Stevanovski, I., Ferguson, J.M., Ragel, R., Gamaarachchi, H., Deveson, I.W.: Genopo: a nanopore sequencing analysis toolkit for portable Android devices. Commun. Biol. 3, 1–5 (2020). https://doi.org/10.1038/s42003-020-01270-z.
4. Christley, S., Lu, Y., Li, C., Xie, X.: Human genomes as email attachments. Bioinformatics. 25, 274–275 (2009). https://doi.org/10.1093/bioinformatics/btn582.
5. Liu, Y., Peng, H., Wong, L., Li, J.: High-speed and high-ratio referential genome compression. Bioinformatics. 33, 3364–3372 (2017). https://doi.org/10.1093/bioinformatics/btx412.
6. Ochoa, I., Hernaez, M., Weissman, T.: iDoComp: a compression scheme for assembled genomes. Bioinformatics. 31, 626–633 (2015). https://doi.org/10.1093/bioinformatics/btu698.
7. Pratas, D., Pinho, A.J., Ferreira, P.J.S.G.: Efficient Compression of Genomic Sequences. In: 2016 Data Compression Conference (DCC). pp. 231–240 (2016). https://doi.org/10.1109/DCC.2016.60.
8. Deorowicz, S., Grabowski, S.: Robust relative compression of genomes with random access. Bioinformatics. 27, 2979–2986 (2011). https://doi.org/10.1093/bioinformatics/btr505.
9. Deorowicz, S., Danek, A., Niemiec, M.: GDC 2: Compression of large collections of genomes. Sci. Rep. 5, 11565 (2015). https://doi.org/10.1038/srep11565.
10. Wandelt, S., Leser, U.: FRESCO: Referential Compression of Highly Similar Sequences. IEEE/ACM Trans. Comput. Biol. Bioinform. 10, 1275–1288 (2013). https://doi.org/10.1109/TCBB.2013.122.
11. Chen, X., Li, M., Ma, B., Tromp, J.: DNACompress: fast and effective DNA sequence compression. Bioinformatics. 18, 1696–1698 (2002). https://doi.org/10.1093/bioinformatics/18.12.1696.
12. Korodi, G., Tabus, I.: An efficient normalized maximum likelihood algorithm for DNA sequence compression. ACM Trans. Inf. Syst. 23, 3–34 (2005). https://doi.org/10.1145/1055709.1055711.
13. Duc Cao, M., Dix, T.I., Allison, L., Mears, C.: A Simple Statistical Algorithm for Biological Sequence Compression. In: 2007 Data Compression Conference (DCC'07). pp. 43–52. IEEE, Snowbird, UT, USA (2007). https://doi.org/10.1109/DCC.2007.7.

14. Kuruppu, S., Beresford-Smith, B., Conway, T., Zobel, J.: Iterative Dictionary Construction for Compression of Large DNA Data Sets. IEEE/ACM Trans. Comput. Biol. Bioinform. 9, 137–149 (2012). https://doi.org/10.1109/TCBB.2011.82.
15. Brandon, M.C., Wallace, D.C., Baldi, P.: Data structures and compression algorithms for genomic sequence data. Bioinformatics. 25, 1731–1738 (2009). https://doi.org/10.1093/bioinformatics/btp319.
16. Hernaez, M., Pavlichin, D., Weissman, T., Ochoa, I.: Genomic Data Compression. Annu. Rev. Biomed. Data Sci. 2, 19–37 (2019). https://doi.org/10.1146/annurev-biodatasci-072018-021229.
17. Ozsoy, A., Swany, M., Chauhan, A.: Pipelined Parallel LZSS for Streaming Data Compression on GPGPUs. In: 2012 IEEE 18th International Conference on Parallel and Distributed Systems. pp. 37–44 (2012). https://doi.org/10.1109/ICPADS.2012.16.
18. Patel, R.A., Zhang, Y., Mak, J., Davidson, A., Owens, J.D.: Parallel lossless data compression on the GPU. In: 2012 Innovative Parallel Computing (InPar). pp. 1–9 (2012). https://doi.org/10.1109/InPar.2012.6339599.
19. Salamat, S., Rosing, T.: FPGA Acceleration of Sequence Alignment: A Survey, http://arxiv.org/abs/2002.02394, (2020). https://doi.org/10.48550/arXiv.2002.02394.
20. Nobile, M.S., Cazzaniga, P., Tangherloni, A., Besozzi, D.: Graphics processing units in bioinformatics, computational biology and systems biology. Brief. Bioinform. 18, 870–885 (2017). https://doi.org/10.1093/bib/bbw058.
21. Wilson, C.N., Musicha, P., Beale, M.A.: Genomic epidemiology on the move. Nat. Rev. Microbiol. 21, 69–69 (2023). https://doi.org/10.1038/s41579-022-00836-4.
22. Snapdragon 855 Mobile Platform | Qualcomm, https://www.qualcomm.com/products/mobile/snapdragon/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-855-mobile-platform, last accessed 2024/04/06.
23. OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems, https://www.khronos.org//, last accessed 2024/04/06.
24. Jo, G., Jeon, W.J., Jung, W., Taft, G., Lee, J.: OpenCL framework for ARM processors with NEON support. In: Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing. pp. 33–40. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2568058.2568064.
25. FreeOCL: Multi-platform implementation of OpenCL 1.2 targeting CPUs, https://goo.gl/qWL1Eg, last accessed 2024/03/26.
26. Home - OpenMP, https://www.openmp.org/, last accessed 2024/04/06.
27. Mewes, H.-W., Albermann, K., Bähr, M., Frishman, D., Gleissner, A., Hani, J., Heumann, K., Kleine, K., Maierl, A., Oliver, S., Pfeiffer, F., Zollner, A.: Overview of the yeast genome. Nature. 387, 7–65 (1997). https://doi.org/10.1038/42755.