MetaZip: A High-throughput and Efficient Accelerator for DEFLATE

Ruihao Gao[1,2], Xueqi Li[1,*], Yewen Li[1,2], Xun Wang[3,1], Guangming Tan[1,2]
[1]Institute of Computing Technology, Chinese Academy of Sciences [2] University of Chinese Academy of Sciences
[3]China University of Petroleum (East China)
[*]Corresponding authors (email: lixueqi@ict.ac.cn)

ABSTRACT

Booming data volume has become an important challenge for data center storage and bandwidth resources. Consequently, fast and efficient compression architecture is becoming the most fundamental design in data centers. However, the compression ratio (CR) and compression throughput are often difficult to achieve at the same time on existing computing platforms. DEFLATE is a widely used compression format in data centers, which is an ideal case for hardware acceleration. Unfortunately, Deflate has an inherent connection among its special memory access pattern, which limits a higher throughput.

In this paper, we propose MetaZip, a high-throughput and scalable data-compression architecture, which is targeted for FPGA-enabled data centers. To improve the compression throughput within the constraints of FPGA resources, we propose an adaptive parallel-width pipeline, which can be fed 64bytes per cycle. To balance the compression quality, we propose a series of sub-modules (e.g. 8-bytes MetaHistory, Seed Bypass, Serialization Predictor). Experimental results show that MetaZip achieves the throughput of 15.6GB/s with a single engine, which is 234×/2.78× than a CPU gzip baseline and FPGA based architecture, respectively.

ACM Reference Format:
Ruihao Gao, Xueqi Li, Yewen Li, Xun Wang, Guangming Tan. 2022. MetaZip: A High-throughput and Efficient Accelerator for DEFLATE. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC) (DAC '22), July 10–14, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3489517.3530450

1 INTRODUCTION

Data growth has become a key challenge for data centers, which require significant overhead in order to compute, store and transmit huge amounts of data. Taking genome data as an example, it is predicted that by 2025, 1 billion people will have their own genomes, generating up to 40 exabytes a year of genomics data [8]. Thus, data compression has become a urgent need to storage and memory cost savings. However, the overhead of existing data compression technologies is enormous. For example, in the analysis measured on more than 20,000 Google machines over a three-year period [10], compression took up more than one-quarter of all tax cycles. The computational burden motivates the acceleration of data compression applications in today's data centers.

Deflate is widely used in highly interconnected enterprise and cloud environments for lossless compression and is utilized in many data compression tools such as GZIP [6], ZLIB. Therefore, Deflate represents a realistic option for acceleration. In the past, software optimizations [9] provide speedup for Deflate by sacrificing compression ratio. Nevertheless, the throughput on existing CPU system is only about a hundred megabytes per second. In response to the shortcomings of software approaches, some hardware approaches, such as FPGA [3, 7, 11, 13], ASIC [2], were proposed. The industry-leading NXU accelerator [2] claimed 68 PCIe based compression cards with 4GB/s peak throughput are needed to match its performance on the largest z15 system topology with 20 processor chips. However, the hard-coded ASIC implementation is not being able to meet new compression algorithms. Due to the growing deployment of FPGAs in cloud data centers [12] and the flexibility of FPGAs, accelerating compression using FPGA is becoming a practical solution. However, previous studies used resource-intensive designs, which limited both the scalability and throughput.

In this paper, we propose MetaZip, which is targeted at FPGA-enabled data centers. In contrast to previous studies, each engine in MetaZip is designed to process 64 bytes/clock cycle @250MHz. To fully take advantage of FPGA resources, we propose the seed-extension scheme in MetaZip. MetaZip introduces several novel designs that maintain compression ratio while significantly improving throughput: 1) Seed Bypass balances the problem that the output of the hashSeeding phase is much larger than that of the extension phase, 2) 8 bytes metaHishtory effectively provides information for port resource allocation, which significantly reduce unnecessary memory access, 3) the ArbiterTree enables MetaZip to have adaptive throughput in the face of dynamically changing inputs in the real system. In addition, MetaZip contains a serialization predictor to avoid modifying the data flushed out, which is faithful to Deflate specification. The main contributions of this paper are:

- We propose MetaZip, a high-throughput and scalable compression architecture for Deflate. MetaZip is designed with the philosophy of maximizing throughput. We designed an adaptive parallel-width pipeline to overcome the bottleneck of static architecture.
- We propose series of sub-module, such as 8 bytes metaHistory, Seed Bypass mechanism, and serialization predictor to improve compression ratio at high throughput.
- Experimental results show that MetaZip achieves 15.6GB/s throughput with a single engine. By deploying MetaZip on a

This work was supported in part by China National Postdoctoral Program for Innovative Talents under Grant BX2021320, in part by NSFC under Grant T2125013.

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07.
https://doi.org/10.1145/3489517.3530450

machine with only 8 U280 accelerator cards, the system can provide 374.4GB/s throughput, which is 1.33× better than the industry-leading NXU system.

2 BACKGROUND AND MOTIVATION

2.1 Deflate Overview

Deflate is a widely used lossless compression data format defined in RFC1951 [5]. Although not clearly defined, the Deflate algorithm generally refers to a lossless compression algorithm that generates the specified format.

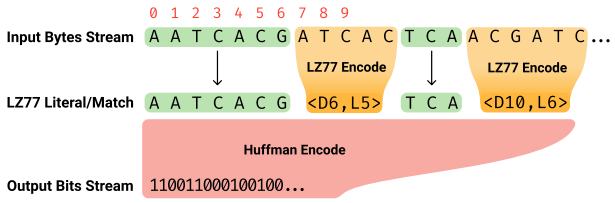

Figure 1: Deflate compression process.

The typical Deflate algorithm implementation consists of LZ77 encoding and Huffman encoding, as shown in Fig. 1. The LZ77 encoding phase uses the Lempel-Ziv compression algorithm [14] variants to convert the input byte stream into a series of literals or matches. A *literal* is identical to the input byte. A *match* is a $< Dist, Len >$ pair that indicates current input repeats *Len* bytes with input *Dist* bytes ago. In gzip, LZ77 encoding is implemented with a *hash table* and a 32KB *sliding window*. The hash value of the first 3 bytes input is taken to index the hash table and find the history addresses in the sliding window where a duplicated string might exist.

After that, all literals and matches are encoded again using the Huffman algorithm. Deflate supports fixed or dynamic Huffman codes. In the fixed mode, predetermined Huffman codes are embedded into the compressor to eliminate the overhead of building the Huffman tree and achieve higher speed.

To improve throughput, deflate algorithms can be parallelized in two dimensions: block-wise and byte-wise. The input file is split into blocks, and the blocks are assigned to multiple standalone threads in block-wise parallel mode. On the other side, the compressor is extended to accept *P* parallel bytes. The two parallel dimensions are orthogonal and can be combined to achieve higher throughput. However, it is essential to note that block-wise parallelism violates deflate streaming input and output requirements to some extent, so the scope of application will be narrowed.

2.2 Motivation and Challenges

According to our observations, the bottleneck of gzip performance is caused by frequent and fragmented data access. We profiled gzip using Intel VTune, and Table 1 lists the functions and CPU time proportions that account for the majority of execution time. In function 'longest_match', which takes up 65.7% execution time, gzip performs hash table lookups and byte-wise string prefix matching operations. Heterogeneous architectures with memory structure optimization are potential paths to speed up the deflate algorithm.

Table 1: Execution time breakdown of Gzip.

Function	CPU Time %	Description
longest_match	65.7%	LZ77 Encode
fill_window	9.9%	File read
flush_block	7.6%	Huffman Encode & file write
ct_tally	3.3%	Huffman Encode

Previous works have proposed some designs but are not efficient enough. [7] proposed a multi-banking hash table that operates at a double clock rate of the rest parts, which can take *P* bytes input and produce *P* history address candidates each cycle. However, after the well-designed hash table, *P* memory banks of *P*-bytes data width are needed to perform the match, which means the number of required BRAM is $O(P^2)$. Although the number of BRAM blocks in FPGA is increasing, our practice shows that the poorly-organized history memory will incur routing congestion and eventually lead to difficulty in timing convergence when $P \geq 32$. In addition, when the hash table depth $HTD > 1$, the hash table produces more candidate history addresses. Based on our experiments, arbitrary discarding candidate history address results in a lower compression rate.

Another approach to overcome the memory access challenge is the *NearCAM* based on comparator array and shift registers introduced by [2]. The NearCAM completely discarded the hash table and the history memory. It achieved efficient data reuse by redundant operation components. Nonetheless, NearCAM does not perform well in scalability. On the one hand, the number of comparators required by NearCAM is $O(P \cdot \text{SlidingWindowSize})$, and many of the comparator results are useless due to the lack of hash table guidance. On the other hand, the critical path length of NearCAM is $O(P)$, and it is not easy to be pipelined. These characteristics limit the scale of NearCAM and harm throughput and compress ratio. In fact, in previous work [2], along with the novel NearCAM, a traditional FarCAM which implemented with hash table and history memory likes [7] is also included.

3 METAZIP ACCELERATOR

3.1 Overview

The architecture of MetaZip is shown in Fig. 2(a). We hope to achieve higher throughput by expanding the parallel width of MetaZip. Since the increase in parallel width causes a dramatic increase in resource overhead for a single memory component, simply stacking more logic resources does not improve throughput effectively. To maximize the throughput, we designed the hash table with *MetaHistory* and *Seed Bypass* mechanism to reduce the contention for history memory. We also proposed *Serialization Predictor* and novel interconnection structure with *ArbiterTree* to maintain compression ratio at high throughput.

MetaZip is a non-blocking pipeline with 4 main phases. MetaZip can accept *P* bytes input per cycle. The input data firstly come to the ① *HashSeeding* phase for hash table lookup and update. The hashSeeding phase produces *seeds* corresponding to input. Unlike software and previous works, a seed contains not only a possible

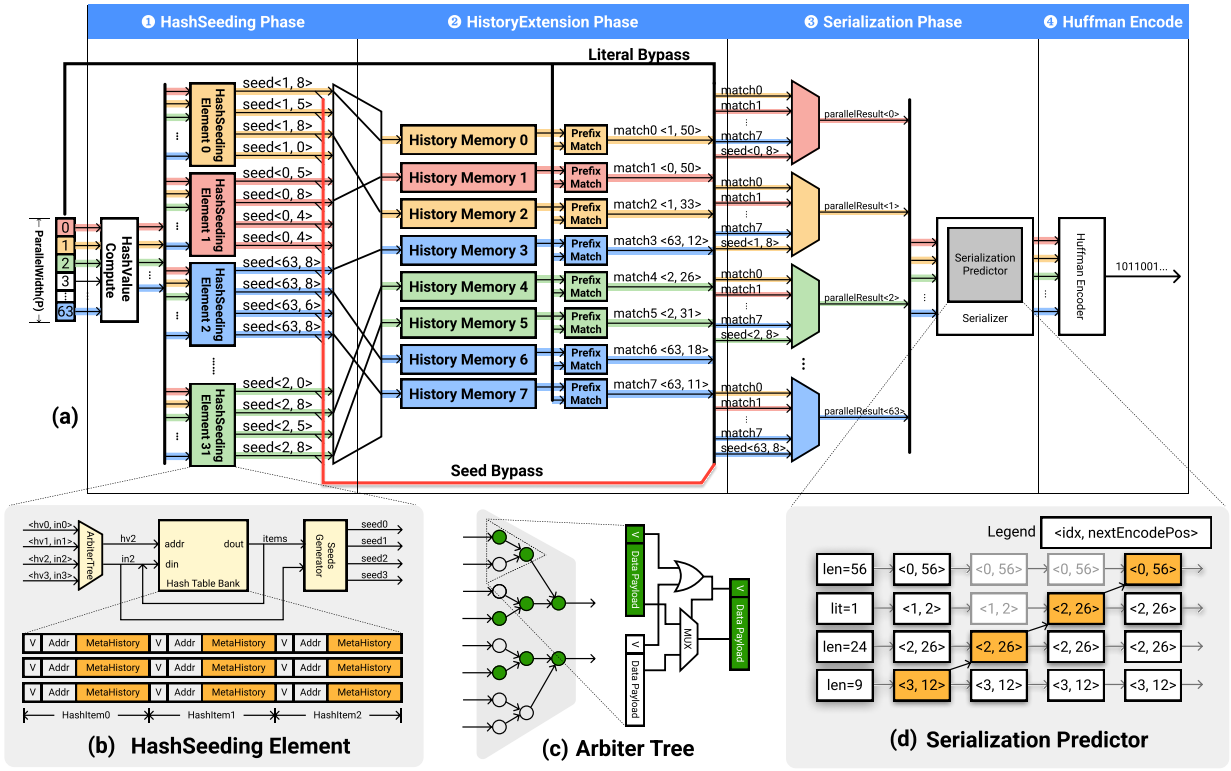

Figure 2: The architecture of MetaZip. (a) The architecture and the dataflow with in 4 phases. P input bytes are converted to seeds, matchs, and finally bit stream. (b) The architecture of the hashSeeding element and the hash table with metaHistory. (c) Circuit of the arbiterTree. (d) Dataflow in the serialization predictor.

history address, but also a locally matched length that we call *metaLength*. The number of seeds generated by the hashSeeding phase depends on the number of the hashSeeding elements(*HTP*) and the depth of the hash table(*HTD*). In a typical configuration, the hashSeeding phase can generate up to 256 seeds each cycle.

Next, seeds enter the ② *HistoryExtension* phase, in which they will guide history memories to pick up history string and match with current input. Duplicate history memories play the role of the sliding window in the LZ77 encoding and their ports need to be wide enough to accept *P* bytes input without loss. Such a wide data width can lead to inefficient use of memory resources in FPGA, so the number of history memory is minimal and less than the number of input seeds. An arbiter tree is employed before history memory banks to pick out seeds that may result in longer matching lengths.

After the historyExtension phase, the seeds have been extended into matches with full lengths. However, since matching results are generated in parallel, there may be overlaps between matches, which is not allowed by DEFLATE. ③ *Serialization* phase eliminates repeats among matches. Finally, the serialized result will be sent to the ④ *Huffman Encode* phase and be packed into the output bit stream.

3.2 HashSeeding Element with MetaHistory

The hashSeeding phase is formed by *HTP* hashseeding elements shown as Fig. 2(b). The whole hash table is an SRAM scratchpad addressed according to the hash value, partitioned horizontally and evenly into all hashSeeding elements. Each hashSeeding element receives all input hash values, and an arbiterTree picks up one that falls within the current address interval. The data organization of hash table is shown in Fig. 2(b). Rows of the hash table contain *HTD* hash items and are read out entirely and updated like a shift register. A hash item includes not only a history address where a match might occur but also a short history segment called *MetaHistory*.

Putting additional metadata into hash items is an effective way to improve compression performance. The most intuitive reason is that it gives one access to the hash table to fetch more information and merges several potential memory access into one. Previous work [4] and open source Vitis compression library [1] used this schemes to increase compression ratio. Furthermore, we found that additional metadata can play a more significant role in designing a high throughput DEFLATE accelerator since higher throughput increases the conflict between memory port requirements and the FPGA routing resource. So we add 8 bytes metaHistory for each hash item radically.

After hash table lookup, seedGenerator calculate metaLength for each seed according to metaHistory and current input. MetaLength is precisely the match length if there is a match of current input at the history address. Obviously, if the seed's metaLength is not longer than the metaHistory's length, MetaZip no longer needs to

allocate an memory port for the seed. This strategy can significantly reduce unnecessary memory access.

The memory banks in hashSeeding elements are implemented with LUTRAM rather than block RAM for following two reasons:

- **Resource utilization**: As the hash table is split horizontally, and when HTP is large, there are fewer rows in each hashSeeding element. Utilizing block RAM in this situation is wasteful, but LUTRAM is appropriate.
- **Timing burden**: Data read from the hash table need to be written back to the same address in the same cycle. Using LUTRAM can reduce the timing burden because no optional output register could be merged into the RAM block.

Finally, the complete hashSeeding phase can generate up to 256 seeds per cycle, providing 500 GB/s of memory bandwidth at a clock frequency of 250 MHz.

3.3 Seed Bypass

The hashSeeding phase produces far more seeds than the historyExtension phase can handle. According to the strategy in Section 3.2, only seeds with long enough metaLength can occupy memory ports in the historyExtension phase. To take full advantage of metaHistory, rather than drop too many seeds directly, we designed a *seed bypass* along with the historyExtension phase.

At the end of the hashSeeding phase, for each hashSeeding element, the seed with max metaLength is picked. Selected seeds are sent into seed bypass and forward to the end of the historyExtension phase and merge with match result produced by the historyExtension phase. If the metaLength of a seed is longer than the minimal match length allowed by DEFLATE (typically 3), the seed can be treated as a match result.

Seed bypass can increase compression from two perspectives:

- If a hashSeeding elements does not produce any seed long enough to participate in the extension, its result also has the chance to become the final match.
- If the hashSeeding phase produces too many seeds long enough for the historyExtension to handle them all, a nice enough seed can also be the final result.

4 OPTIMIZATION

4.1 Serialization Predictor

The DEFLATE algorithm requires the output to be serialized, meaning that the literal / match encoding positions for each output are continuous and do not overlap. This requirement ensures that the DEFLATE decompressor can be streamed without modifying the data flushed out. The serialization phase takes charge of eliminating overlap between the results of the historyExtension phase. As the length of match might span parallel windows, there are dependencies between parallel windows. However, when P is large, serialization of a single window cannot be completed in one cycle. This will lead to pipeline blocking and throughput reduction. Therefore, we designed a serialization predictor as shown in Fig. 2(d) to eliminate the impact of dependencies between parallel windows.

The Serialization Predictor consists of multiple stages. It takes match lengths(length=1, if literal) from the historyExtension phase as input. At the first stage, each length is added with index in parallel window and turned into *nextEncodePos* which indicates the subsequent position after current match. NextEncodePos is allowed to stride across parallel windows. Then, P stages are applied to calculate the suffix max nextEncodePos for each input and the index of the max value is also propagated forward. The output of the serialization predictor is serial of $< idx, nextEncodePos >$ pairs.

The serialize phase has a Global Next Encode Position (GNEP) register and a MUX selects Serialization Predictor Result (SPR) from the output of the serialization predictor according to GNEP. The selection of SPR can be completed in 1 cycle. In subsequent encoding, the next GNEP value will be SPR.nextEncodePos, provided that the corresponding match for SPR.idx is not overwritten.

Compared to trimming match length into single parallel window, the serialization predictor can make full use of previous phase results to improve the compression ratio without compromising the throughput.

4.2 ArbiterTree and Adaptive Throughput

In MetaZip, data arbitration is required in many places, e.g. the hashSeeding elements selects 1 hash value from P inputs, the historyExtension phase selects HMPs from many seeds for extension and the shuffled match need to be reordered before the serialization phase. Due to the wide data width, the priority encoder and MUX using pure combinational logic consume many LUT resources and have a poor timing performance. So, we use *ArbiterTree* shown as Fig 2(c) to settle down data arbitration issues.

The arbiterTree is organized as a binary tree. Each node in the arbiterTree has a valid bit and a data payload field. Assuming that the two children of a node P are L and R, then $P.valid = L.valid|R.valid$ and payload of P can be selected by a 2-to-1 MUX. ArbiterTree's layered structure makes it easy to be pipelined.

The ArbiterTree enables MetaZip to have adaptive throughput. MetaZip is designed as a non-blocking pipeline with stable throughput. However, when MetaZip is integrated into a system, the external bandwidth may be dynamic. When the speed of input or output cannot satisfy the full throughput of the pipeline, we can reduce parallel width by marking some input bytes as invalid. Since the arbiterTree decouples the data width between different phases, a narrow input can still fill up the data lane and result in a higher compression ratio with less conflict on memory ports. This adaptability is on-the-fly and does not require reconfiguration of FPGA.

5 EXPERIMENTAL RESULTS

5.1 Experimental Setup

To evaluate the throughput and compression ratio of MetaZip, we have fully implemented the hashSeeding phase, historyExtension phase, and serialization phase with Chisel. The key design parameters are listed in Table 2. We choose fixed Huffman encoding in MetaZip since it is friendly to FPGA.

We synthesized and implemented MetaZip in Xilinx Vivado v2020.1 and the target board is Xilinx Alveo U280 Data Center Accelerator Card. We were able to close the timing of MetaZip at 250MHz. We built an emulator with chiseltest and Verilator, which can produce traces of the serialization phase output. The output of the MetaZip emulator is piped into a static Huffman encoder that can generate files in standard DEFLATE format. We use Silesia

Table 2: Parameters configuration of MetaZip.

Parameter	Value
Max Parallel Width (P)	64 bytes
Hash Value Width	12 bits
Hash Table Depth (HTD)	8
MetaHistory Length	8 bytes
# of HashSeeding Elements (HTP)	32
# of History Memory (HMP)	8
Max Match Length	64 bytes

Table 3: Throughput and compression ratio comparisons.

Design	Throughput	Compression Ratio		
		Silesia	Canter.	Cal.
MetaZip(P=64)	15.6GB/s@250MHz	1.68	1.65	1.60
MetaZip(P=32)	7.8GB/s@250MHz	2.06	1.94	1.89
MetaZip(P=16)	3.9GB/s@250MHz	2.34	2.36	2.08
MetaZip(P=8)	1.9GB/s@250MHz	2.49	2.50	2.14
Gzip-fastest	68MB/s	2.74	3.24	2.63
Microsoft [7]	5.6GB/s@175MHz	1.90	2.56	2.09
IBM NXU [2]	19.5GB/s@2.5GHz	2.38	2.89	2.68
Vitis [1]	300MB/s@300MHz	2.70	-	-

corpus, Canterbury corpus and Calgary corpus as the benchmark for the compression ratio evaluation.

5.2 Analysis of Experimental Results

The evaluation results of MetaZip is listed in Table 3. The result of gzip is measured on a machine with a 2.10GHz Intel Xeon CPU E5-2620 v4 CPU and 128GB RAM. The results of related works are extracted from [7][2][1].

Throughput. Because MetaZip is a non-blocking pipeline that can handle 64 bytes of input per cycle at a clock frequency of 250MHz, the maximum throughput of single engine is 15.6 GB/s. If external blocking is taken into account, the adaptive throughput can vary from 1.9GB/s to 15.6GB/s.

The throughput of MetaZip is 234× higher than single thread gzip. Maximum throughput improves by 2.78× compared to [7], the design took high throughput as the same target and implemented on FPGA. The Vitis compression library is designed in block-wise parallel mode, so single-engine throughput is limited, typically with 8 engines in a kernel for 2.9 GB/s throughput and 5.3× slower than MetaZip. Since the parallel modes are orthogonal, one possible solution is to replace the engine of Vitis compression library with MetaZip for higher throughput.

The high single-engine throughput of [2] benefits from the 2.5GHz ASIC implementation, although the parallel width is only 8 bytes. [2] achieved 280GB/s throughput in the largest z15 system by deploying 20 processor chips with NXU. Based on the resource utilization, at least 3 MetaZip accelerators can be implemented on a single Alveo U280 card. MetaZip can provide 374.4GB/s throughput by leveraging 8 U280 cards. However, installing 8 dual-width PCIe accelerator cards in a 4U chassis is much easier than 20 processors.

Table 4: Configurations of optimization mechanism.

#	MetaHistory	Seed Bypass	Serialization Predictor
1	0B	×	✓
2	3B	✓	✓
3	8B	×	✓
4	8B	✓	×
default	8B	✓	✓

Compression ratio. Figure 3 depicts the relationship between MetaZip compression ratio and Parallel width, with a significant negative correlation between them. This is because memory ports are more stressed as parallel width increases. Thanks to the features of adaptive throughput, if a higher compression ratio is required for a particular scenario, we can increase compression ratio by actively limiting throughput.

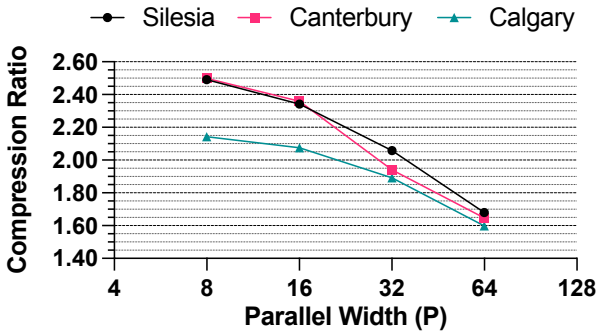

Figure 3: Effect of parallel width on compression ratio.

Due to dynamic Huffman encoding, previous works [1, 2] achieve higher compression ratio than MetaZip, however the cost is much higher implementation complexity and latency. Additional experiments show that MetaZip can provide a compression ratio of 2.948 on silesia corpus when using dynamic Huffman encoding with a 8 bytes parallel window. Gzip also uses static Huffman coding, but the compression ratio is high because there is no memory port contention in the single thread software implementation.

5.3 Performance of Optimization Mechanism

The metahistory, seed bypass and serialization predictor mechanisms are designed to increase compression ratio while providing high throughput. This section evaluates the effect of these mechanisms on compression ratio by 4 different configurations listed in Table 4. The compression ratio of Silesia corpus on four experimental configurations and default is shown in Figure 4.

MetaHistory. A comparison of configurations 1, 2, and default reveals the role of MetaHistory in improving compression ratio. Using no metahistory or only 3 bytes MetaHistory will downgrade average compression ratio 32.3% and 26.4%. Note that using 3 bytes history improves compression ratio by only 8.5% compared to no

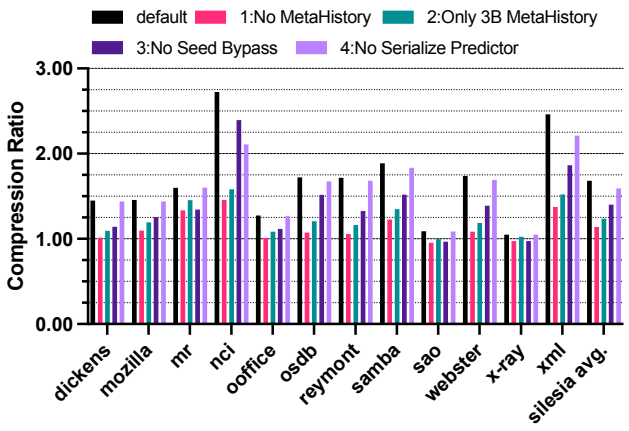

Figure 4: Effect of different optimization mechanisms on the compression ratio.

Table 5: Power consumption breakdown of MetaZip.

	Dynamic/W	%	Static/W	%
Total = 15.464W (100%)	12.066	78.0	3.398	22.0
HashSeeding	5.907	38.2	-	-
HistoryExtension	5.790	37.4	-	-
Serialization	0.369	2.40	-	-

metaHistory, and when metaHistory extends to 8 bytes, the compression ratio increases by 47.6%. The reason for this result is that longer MetaHistory can better filter invalid seeds caused by hash conflicts and produce seeds with longer metaLength.

Seed Bypass. An experimental comparison of the default configuration and configuration 1 shows that seed bypass can increase the average compression ratio by 20% when using the same 8 bytes metahistory. It is worth noting that test cases such as xml are more sensitive(increased 32.0%) to the seed bypass mechanism because their matching lengths are more distributed within 8 and more seeds can be converted through seed bypass.

Serialization Predictor. In Experiment 4, serialize predictor was disabled. To produce the right results, we serialize using a naive approach, which is to trim match length to ensure that it does not cross parallel windows. The results show that the naive method can reduce the average compression ratio by 5.66%.

5.4 Power and area breakdown

The power and area of MetaZip are estimated by Vivado and listed in Table 5 and Table 6. MetaZip requires about twice as much area as [7]. With a 2.78x increase in throughput, the area did not grow scale quadratically as [7] predicted, which means MetaZip has better scalability in area. Thanks to the low power feature of FPGA, the total power of MetaZip is 15.5W, only 55.7% of 28W NXU in [2]. And the energy/GB of MetaZip is 0.98J/GB, 4.28x lower than NXU. MetaZip has advantages over ASIC in terms of energy consumption.

Table 6: Resource utilization results of MetaZip.

	Avail.	HashS.	HistoryE.	Serial.	Total
LUT	1303K	110K	118K	16K	244K
	100%	8.47%	9.05%	6.36%	18.7%
LUTRAM	600K	49K	2K	12K	64K
	100%	8.23%	0.475%	2.00%	10.7%
FF	2607K	54K	133K	8K	196K
	100%	2.07%	5.13%	0.326%	7.53%
BRAM	2016	0	128	0	128
	100%	0	6.35%	0	6.35%

6 CONCLUSION

In this work, we propose a FPGA-based accelerator for Deflate, which adopts 8 bytes metaHistory, seed bypass, serialization predictor mechanisms to provide high throughput. Experimental results show that MetaZip with single engine can provide 15.6GB/s throughput, which is 234×/2.78× than a CPU gzip baseline and FPGA based architecture. By leveraging the adaptive throughput design, the compression ratio of MetaZip can be ranging from 1.68 to 2.49 according to the dynamic workloads in the real system.

REFERENCES

[1] 2021. Vitis Data Compression Library. https://xilinx.github.io/Vitis_Libraries/data_compression/2021.2/benchmark.html

[2] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J Starke, et al. 2020. Data compression accelerator on IBM power9 and z15 processors: Industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–14.

[3] Seungdo Choi, Youngil Kim, Daeyong Lee, Sangjin Lee, Kibin Park, Yun Heub Song, and Yong Ho Song. 2019. Design of FPGA-based LZ77 compressor with runtime configurable compression ratio and throughput. *IEEE Access* 7 (2019), 149583–149594.

[4] Seungdo Choi, Youngil Kim, and Yong Ho Song. 2018. False history filtering for reducing hardware overhead of FPGA-based LZ77 compressor. *Journal of Systems Architecture* 88 (Aug. 2018), 110–119. https://doi.org/10.1016/j.sysarc.2018.06.001

[5] Peter Deutsch. 1996. RFC1951: Deflate compressed data format specification.

[6] Peter Deutsch et al. 1996. GZIP file format specification version 4.3. (1996).

[7] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. 2015. A scalable high-bandwidth architecture for lossless compression on fpgas. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 52–59.

[8] Robert Gebelhoff. 2015. Sequencing the genome creates so much data we don't know what to do with it. *The Washington Post* (2015), 1–3.

[9] Vinodh Gopal, J Guilford, E Ozturk, G Wolrich, and W Feghali. 2011. High performance DEFLATE compression on intel architecture processors. *Technical Report 326352-001* (2011).

[10] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169.

[11] Andrew Martin, Damir Jamsek, and K Agarawal. 2013. FPGA-based application acceleration: Case study with gzip compression/decompression streaming engine. *ICCAD Special Session C* 7 (2013), 2013.

[12] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. 2015. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper* 2, 11 (2015), 1–4.

[13] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. 2018. High-throughput lossless compression on tightly coupled CPU-FPGA platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 37–44.

[14] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on information theory* 23, 3 (1977), 337–343.