Assessing the Overhead of Offloading Compression Tasks

Laura Promberger
laura.promberger@cern.ch
CERN
Geneva, Switzerland
Heidelberg University
Heidelberg, Germany

Rainer Schwemmer
rainer.schwemmer@cern.ch
CERN
Geneva, Switzerland

Holger Fröning
holger.froening@ziti.uni-heidelberg.de
Heidelberg University
Heidelberg, Germany

ABSTRACT

Exploring compression is increasingly promising as trade-off between computations and data movement. There are two main reasons: First, the gap between processing speed and I/O continues to grow, and technology trends indicate a continuation of this. Second, performance is determined by energy efficiency, and the overall power consumption is dominated by the consumption of data movements. For these reasons there is already a plethora of related works on compression from various domains. Most recently, a couple of accelerators have been introduced to offload compression tasks from the main processor, for instance by AHA, Intel and Microsoft. Yet, one lacks the understanding of the overhead of compression when offloading tasks. In particular, such offloading is most beneficial for overlap with other tasks, if the associated overhead on the main processor is negligible. This work evaluates the integration costs compared to a solely software-based solution considering multiple compression algorithms. Among others, High Energy Physics data are used as a prime example of big data sources. The results imply that on average the `zlib` implementation on the accelerator achieves a comparable compression ratio to `zlib level 2` on a CPU, while having up to 17 times the throughput and utilizing over 80 % less CPU resources. These results suggest that, given the right orchestration of compression and data movement tasks, the overhead of offloading compression is limited but present. Considering that compression is only a single task of a larger data processing pipeline, this overhead cannot be neglected.

CCS CONCEPTS

• **Computer systems organization** → **Reconfigurable computing**; • **Information systems** → **Data compression**.

KEYWORDS

Compression, Offloading, Accelerator Card, Overhead, Measurement

1 INTRODUCTION

The current computational landscape is dominated by increasingly costly data movements, both in terms of energy and time, while computations continuously decrease in these costs [10, 13]. Furthermore, the amount of data generated and processed, preferably as fast and interactive as possible, is growing dramatically. As a consequence, especially for big data applications, it becomes more and more promising to trade computations for communication in order to diminish the implications of data movements. Overall run time benefits should be possible, even though this might substantially increase the amount of computations. Of particular interest in this context is the recent introduction of dedicated accelerators for compression tasks, including accelerators provided by AHA [12], Intel QuickAssist [14] or Microsoft's Project Corsica [22].

Most published works about compression utilizing accelerators are focusing on the compression performance, including work on GPU [4], addressing SIMD in general [18, 25, 27], and FPGAs [7, 9, 17, 24]. The common part of all these works is the focus on improving compression performance, albeit for various reasons. However, it is commonly neglected to assess the additional integration costs for the entire system. Only if the overhead of orchestrating compression tasks on accelerators and the associated data movements is as small as possible, one can truly speak of an off-loaded task. Otherwise, the host is not entirely free to perform other, hopefully overlapping, tasks.

Thus, the present work will assess these integration costs by evaluating the overhead associated with orchestrating offloaded compression tasks, as well as assessing obtained compression throughput and ratio by comparing it to well-known general-purpose compression algorithms solely run on the host. The data sets used are a representative set of standard compression corpora, and additionally, as a prime example of big data, High Energy Physics (HEP) data sets from the European Organization of Nuclear Research (CERN) are used. For these data sets the following will be assessed: compression time, ratio, overhead and power consumption of a compression accelerator.

In particular, this work makes following contributions:

(1) Proposing and implementing a well-defined, methodical, multi-threaded benchmark;
(2) Analyzing the performance of the accelerator in terms of machine utilization, power consumption, throughput and compression ratio;
(3) Comparing the achieved results to multiple software-based, i.e. non-offloaded, compression algorithms.

Note that this work analyzes only compression and not decompression. Big data applications, like utilized at CERN, often have I/O limitations and expectations, where a lowest throughput threshold must be passed e.g. to fulfill real-time requirements. Compression - compared to decompression - is the throughput limiting task and as a result studied in this work. Notably, for all selected algorithms the decompression speed is at least the compression speed, if not significantly higher.

As a result, this work will provide details for a better understanding of the offloading costs for compression tasks. The methodology allows in particular system designers to identify deployment opportunities given existing compute systems, data sources, and integration constraints.

2 RELATED WORK

To diminish the implications on (network) data movements and storage, massive utilization of compression can be found, among others, in communities working with databases, graph computations or computer graphics. For theses communities a variety of works exists which addresses compression optimizations by utilizing hardware-related features like SIMD and GPGPUs[4, 18, 25, 27].

Moreover, a multitude of research has been conducted to efficiently use Field Programmable Gate Array (FPGA) technology, as they excel at integer computations. Works here range from developing FPGA-specific compression algorithms [9], over integrating the FPGA compression hardware into existing systems [17, 24], to implementing well-known compression algorithms, like `gzip`, and comparing the achieved performance to other systems (CPU, GPGPU). For example, some work explored CPUs, FPGAs, and CPU-FPGA co-design for LZ77 accelerations [7], while other analyzed the hardware acceleration capabilities of the IBM PowerEN processor for zlib [16]. Regarding FPGAs, Abdelfattah et al. [1] and Qiao et al. [21] implemented their own deflate version on an FPGA.

However, the large majority of those works focuses on the compression performance in terms of throughput and compression ratio, but not the resulting integration costs for the entire system. Overall, little to no published work can be found covering the topic of system integration costs in the context of compression accelerators.

3 METHODOLOGY AND BENCHMARK LAYOUT

3.1 Server and Input Data

The server selected was an Intel Xeon E5-2630 v4. The server's technical specifications are listed in Table 1.

For the input data four standard compression corpora and three HEP data sets were selected. To be representative for big data, compression corpora with at least 200 MB were preferred:

- `enwik9`[20] consists of about 1 GB of English Wikipedia from 2006
- `proteins.200MB`[8] consists protein data of the Pizza & Chili corpus
- `silesia` corpus[2] consists of multiple different files, summing up to 203 MB, which were tarred into a single file
- `calgary` corpus[2] is one of the most famous compression corpora. Tarred into a single file, it only sums up to 3.1 MB. Therefore, it was copied multiple times to reach the 150 MB to conform with the benchmark requirements

The HEP data came from three different CERN experiments (ATLAS [4.1 GB], ALICE [1 GB] and LHCb [4.9 GB]).

At CERN, particles are accelerated in a vacuum very close to the speed of light. They are then collided and analyzed by the experiments. The data created consists of geographical locations where particles were registered. Due to the nature of quantum mechanics this process is close to a random number generator. A high compression ratio can therefore not be expected. All data sets contain uncompressed data[1], even though in their normal setup some type of compression might already be employed.

3.2 Compression Algorithms

The selection of general-purpose compression algorithms was based on the requirement to be lossless and their usage in related works and general popularity, including recent developments. Many of those algorithms allow tuning the performance for compression ratio[2] or throughput[3] by setting the *compression level*. Generally speaking, a higher compression level refers to a higher compression ratio, but also a longer computation time. For this study, compression algorithms run solely on CPU are called *software-based* (sw-based). For them, a pre-study was done to select compression

[1] Parts of the detectors perform already some compression, e.g. similar to the representation of sparse matrices in coordinate format. When talking about uncompressed data, it refers to any form of compression applied after retrieving the data from the detectors.
[2] compression ratio = (uncompressed data) / (compressed data)
[3] throughput = (uncompressed data) / (run time)

Table 1: Technical specifications of the server used for this study

	Xeon
Architecture	Intel
Platform	x86_64
CPU Type	E5-2630 v4 @ 2.20GHz
TDP (per socket)	105 W
Virtual Cores	40
Threads per Real Core	2
Real Cores per Socket	10
Sockets	2
NUMA Nodes	2
RAM (GB)	64
Memory Type	DDR4
RAM Frequency (GHz)	2133
Memory Channels	4
Max. Mem Bandwidth (GB/s)	68
Operating System	CentOS 7
Kernel	3.10.0-957.1.3.el7.x86_64

levels where significant changes in either compression ratio or throughput occurred (see Figure 1).

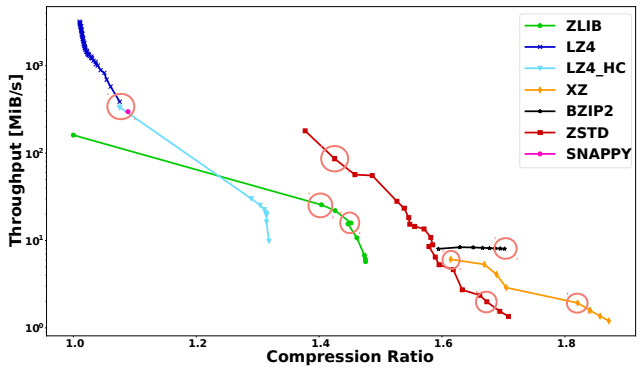
3.2.1 Bzip2. Bzip2[15] combines the Burrows-Wheeler algorithm with a Move-To-Front transform and Huffman coding. It offers nine different levels to change the block size between 100 kB and 900 kB. There is no option to change the compression function being utilized. For this study the largest block size (`level 9`) was chosen.

3.2.2 Lz4. Lz4[19] is an algorithm based on LZ77. The subtype Lz4_HC offers 13 levels to increase the compression ratio. For this study Lz4_HC `level 1` was selected. It has a low compression ratio, but a high throughput. For the rest of this study Lz4_HC is referred to as `lz4`.

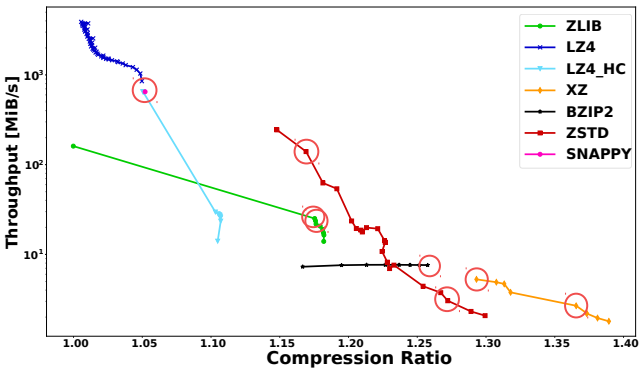
3.2.3 Snappy. Snappy[11] is an algorithm based on LZ77. It was developed at Google with the goal to have a short computation time. Snappy has no tuning parameters.

3.2.4 Xz. Xz[5] is a popular tool which offers multiple compression algorithms. The main algorithm used is LZMA2, which has 9 levels and can achieve a high compression ratio, but requires a long computation time. For this study `level 1` and `level 5` were selected.

3.2.5 Zlib. Zlib[3] utilizes the *deflate* algorithm[6]. Deflate is an algorithm based on LZ77, followed by Huffman coding. Zlib is a popular library used, among others, for ZIP and gzip compression. Zlib offers 9 compression levels, of which `levels 2` and 4 were used in this study. Additional



(a) ALICE



(b) LHCb

Figure 1: Pre-study to evaluate the compression algorithms. The throughput is of a single thread for the data sets ALICE and LHCb. The selected algorithms and there compression levels are circled in red.

to `levels 2` and 4 were selected to compare better against the accelerator zlib implementation.

3.2.6 Zstandard (zstd). Zstd[26] is an algorithm based on LZ77, in combination with fast Finite State Entropy and Huffman coding. It was developed by Facebook with the goal to do real-time compression. Zstd offers 23 levels. Above compression level 20 the configuration differs to achieve a higher compression ratio, but this increases significantly the memory usage. For this study `level 2` and `level 21` were selected.

3.3 Compression Accelerators

At the beginning of this study, two commercially available accelerators were considered: Intel QuickAssist 8970 (QAT), and AHA378. QAT is an ASIC which offers encryption and compression acceleration up to 100 Gb/s. AHA378 is an FPGA

accelerator with up to 80 Gb/s, solely designed for compression. Unfortunately, during this study the QAT exhibited instabilities in performance and thus did not deliver any presentable results. Therefore, this work will focus on the AHA378 accelerator. The PCIe Gen 3 x16 accelerator consists of 4 FPGAs, each providing 20 Gb/s, for a total of up to 80 Gb/s. It offers two compression algorithms: `lzs` and `deflate`. For deflate the additional data formats gzip and zlib are provided. The power consumption is specified to not surpass 25 W when the kernel module is not loaded, and not to exceed 75 W during heavy workload, as all the power is provided through the PCIe slot. For this study only deflate was analyzed, as a pre-study showed that lzs was inferior in performance compared to the others, and that the performance of deflate, gzip and zlib were equivalent.

3.4 Benchmark Layout

The benchmarks used for accelerator-based and sw-based compression were built upon the same principles to reduce any benchmark-related bias. The multi-threaded benchmarks are shown in Fig. 2: the `mainThread` coordinated the orchestration of all other threads, and multiple `compressThreads` executed the compression. For the accelerator-based benchmark additional `queueThreads` were introduced. The parameters being measured were: average compression ratio, total size of uncompressed data, total size of compressed data and the exact run time.

3.4.1 Time Measurements. For an accurate measurement of throughput, a particular focus was put on the time measurement. To improve the timing accuracy the `mainThread` not only set-up the `compressThreads`, but also used atomic variables to coordinate start and stop of the compression task, making sure that all `compressThreads` were finished setting up before starting the measurement. The `mainThread` then slept for the requested benchmark run time (2 min) and issued afterwards the stop signal. To measure the sustained throughput only the time within `compressThread` between start and stop was measured. Measuring the time in `mainThread` would falsify the results as it would include the release of the resources, while this study was only interested in the sustained throughput.

3.4.2 Software-based Benchmark. The sw-based benchmark called the compression libraries directly and used `hwloc` to bind threads equally split onto the NUMA nodes. Because of memory limitation reasons the input data set was loaded only once for each NUMA node and shared with the node's `compressThreads`. The compression function compressed 150 MB of the input data, moving via round-robin multiple times over the entire data set. The chunk size was limited to 150 MB to compensate for resource limitations. The chunk

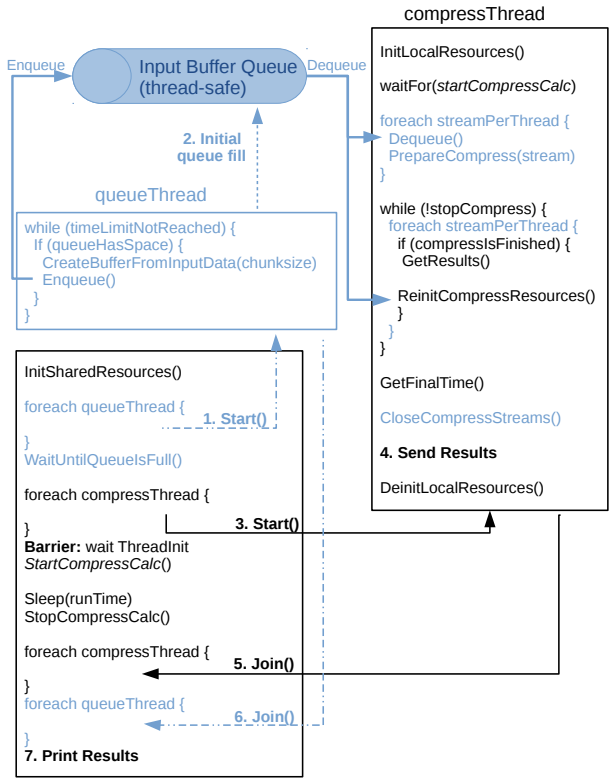


Figure 2: Benchmark layout: The `mainThread` coordinates the progress of the `compressThread`. The `compressThread` continuously compresses the input and records the results. For the accelerator benchmark the additional `queueThread` (in blue) provides the input data chunk-wise to the `compressThread`.

size was selected after pre-studies determined that 150 MB is the smallest chunk size which achieved both, stable performance and compression ratio close to the full data set. All other resources were uniquely assigned and private to each process (e.g. the output buffer).

3.4.3 Accelerator Benchmark. The accelerator benchmark consisted of additional `queueThreads` which filled a thread-safe queue with buffers containing chunks of the input data set. This was necessary as each compression stream needed access to its own allocated input data in order to be able to communicate with the accelerator. The thread-safe queue used locks for inter-thread coordination. `compressThread` continuously consumed those buffers and compressed them. To optimize the load for each thread, `compressThread` was able to run multiple compression streams in a single thread. The compression function used busy waiting to retrieve the results of the streams from the accelerator. Parameters that

could be modified were the number of each type of thread, the input buffer size, the size of the thread-safe queue, the number of compression streams within each `compressThread`, and the type of algorithm and its compression level.

Overall, the main difference to the sw-based benchmark was that each compression stream needed its own allocated memory for the input data, while the sw-based benchmark moved a pointer on the shared memory of the input data.

3.4.4 Performance Instrumentation and Measurement. Both, the acclerator-based and the sw-based benchmarks were run with the instrumentation tool `likwid`[23]. Likwid provides measurement of performance counters similar to `perf`, but is especially designed to support HPC environments. Likwid provides pre-defined groups to measure different aspects of performance. For this study two groups were selected. The first group measured the machine CPU and RAM power consumption, and the read, write and total memory bandwidth. The second group measured the cycle activity, which gave an impression how many cycles were additionally needed due to waiting for different resources. However, it turned out that the cycle activity measurements were not accurate enough on this server to be presentable.

4 ACCELERATOR RESULTS

The accelerator achieved a sustained throughput of 75 - 81 Gb/s, which is at least 94 % of the advertised throughput of 80 Gb/s. This advertised throughput was even surpassed for the calgary corpus. The configuration achieving this was using `numactl` to bind the entire benchmark on the same NUMA node to which the accelerator was connected to. The input parameters were set to a chunk size of 2 MB, 500 elements in the queue, 3 `queueThreads`, 4 `compressThreads` and 12 compression streams per `compressThread`. All results for throughput and compression ratio are listed in Table 2, for memory bandwidth in Table 3 and for power consumption in Table 4.

The lowest throughput and compression ratio was achieved by LHCb with 75 Gb/s and a compression ratio of 1.18. The highest compression ratio was achieved by silesia corpus with a ratio of 2.94 and a throughput of 79 Gb/s. Overall, the data-dependent change of the throughput was less then 4 %.

The memory bandwidth was between 22 - 30 GiB/s. ALICE, ATLAS and LHCb had the highest memory bandwidth, and calgary corpus the lowest memory bandwidth. Exactly the opposite correlation could be found when looking at the memory bandwidth percentage used for reads. Between 51 - 63 % of all memory accesses were reads. In general there was a correlation between having a high compression ratio and a high percentage of reads. A higher compression ratio means a lower amount of output to be written and as a result increases relatively the percentage of reads. Only calgary corpus defied this in the final measurement taken, with having the lowest read percentage (51 %), while at the same time the second highest compression ratio. However, multiple measurements showed that in general the read percentage of calgary corpus was with 59 % on average significantly higher.

The power consumption was stable and, similar to the throughput, was negligibly influenced by the data sets. The CPU power consumption was 58 W and the RAM power consumption was 7 W for the socket to which the accelerator was connected. While running the benchmark, the power consumption of the entire server was 266 W.

5 RESULT COMPARISON

In this section the results of the accelerator are compared to the results of the sw-based benchmark. The algorithms chosen and their respective compression levels are stated in section 3.2. The sw-based benchmark was run with 40 `compressThreads`, using all 40 virtual cores of the server.

The performance of throughput and compression ratio is listed in Table 2. Even though the achieved compression ratios of the accelerator were comparable to `zlib level 4`, `zlib level 2` will be taken as comparison. This decision was taken because, relative to `zlib level 2`, `zlib level 4` had only up to 7 % increase in compression ratio, while at the same time the throughput decreased by 13 - 37 %. Depending on the data set, the accelerator achieved a 8 - 17 times increased throughput compared to the sw-based `zlib level 2` when utilizing the entire server. A different algorithm, `zstd level 2` achieved a similar compression ratio as `zlib level 2`, while at the same time having 2 to 3 times higher throughput then `zlib level 2`.

`snappy` and `lz4` achieved the highest throughput: In the best case they achieved the same throughput as the accelerator on the data set PROTEINS. However, overall the throughput was very data-dependent and their compression ratio was only 65 - 82 % of `zlib level 2`. The lowest throughput achieved `zstd level 21` and in few cases `xz level 5`, but these were also the algorithms with the highest compression ratio. For the 150 MB version of the calgary corpus both algorithms, `zstd level 21` and `xz level 5`, were able to recognize the entire calgary corpus sequence, and achieved a compression ratio of over 170.

The memory bandwidth is listed in Table 3 and was for the sw-based algorithms between 1 - 40 GiB/s. `zlib level 2` had the lowest with 1 - 2 GiB/s and `xz level 5` the highest. The percentage of memory bandwidth being read access was between 49 - 77 %. `zlib level 2` had here a similar split as the accelerator.

The CPU power consumption of the sw-based algorithms was between 72 - 103 W for each socket, with `snappy` and `zstd level 21` having the lowest CPU power consumption

Table 2: Compression ratio and throughput for the accelerator and sw-based algorithms. For the latter only the maximum and minimum values are listed. Additionally, `zlib level 2` and `4` are listed to be able to compare to the accelerator and to see the difference between those two levels.

	Compression Ratio					Throughput (Gb/s)				
	lowest	highest	zlib level 2	zlib level 4	AHA	lowest	highest	zlib level 2	zlib level 4	AHA
ALICE	lz4	xz level 5		+1%		zstd level 21	lz4		-29%	
	1.07	1.83	1.42	1.44	1.4	0.2	45.2	4.5	3.2	76
ATLAS	snappy	xz level 5		+1%		zstd level 21	snappy		-21%	
	1.36	1.92	1.65	1.67	1.67	0.3	63.4	6.2	4.9	77
CALGARY	snappy	xz level 5		+7%		xz level 5	snappy		-29%	
	1.81	185	2.73	2.93	2.86	0.6	53.5	9.3	6.6	81
ENWIK9	snappy	zstd level 21		+7%		zstd level 21	snappy		-28%	
	1.79	3.95	2.72	2.92	2.87	0.2	50.4	9.6	6.9	79
LHCb	lz4	xz level 5		<+1%		zstd level 21	lz4		-13%	
	1.05	1.37	1.17	1.18	1.18	0.3	46.2	4.7	4.1	75
PROTEINS	snappy	zstd level 21		+3%		zstd level 21	lz4		-37%	
	1.23	3.35	2.04	2.11	2.12	0.1	78.2	8.1	5.1	78
SILESIA	snappy	xz level 5		+5%		zstd level 21	snappy		-26%	
	1.93	4.11	2.74	2.87	2.94	0.3	66.4	9.8	7.3	79

Table 3: Memory bandwidth and percentage of reads for the accelerator and sw-based algorithms. For latter only the maximum and minimum values are listed. Additionally, `zlib level 2` is listed to be able to compare to the accelerator.

	Memory Bandwidth (GiB/s)				Percentage of Reads			
	lowest	highest	zlib level 2	AHA	lowest	highest	zlib level 2	AHA
ALICE	zlib level 2	xz level 5			lz4 snappy	xz level 1		
	1	33	1	30	49%	77%	53%	58%
ATLAS	zlib level 2	xz level 5			zlib level 2	xz level 1		
	2	25	2	30	50%	68%	50%	59%
CALGARY	zlib level 2	xz level 5			zstd level 21	xz level 1		
	2	38	2	22	50%	69%	64%	59%
ENWIK9	zlib level 2	zstd level 21			xz level 5	xz level 1		
	2	28	2		56%	69%	61%	57%
LHCb	zlib level 2	xz level 5 zstd level 21		29	zlib level 2	xz level 1		
	2	30	2	30	45%	72%	45%	56%
PROTEINS	zlib level 2	xz level 5			lz4 snappy	xz level 1		
	2	40	2	29	51%	83%	65%	61%
SILESIA	zlib level 2	xz level 5			zstd level 21	xz level 1		
	2	22	2	28	55%	72%	65%	63%

Table 4: CPU and RAM power consumption for the accelerator and sw-based algorithms. For the latter only the maximum and minimum values are listed. Additionally, `zlib level 2` is listed to be able to compare to the accelerator.

	CPU Power Consumption (W)				RAM Power Consumption (W)			
	lowest	highest	zlib level 2	AHA	lowest	highest	zlib level 2	AHA
ALICE	zstd level 21 82	zstd level 2 101	99	58	zlib level 2 4	xz level 5 10	4	7
ATLAS	zstd level 21 85	zstd level 2 101	98	59	zlib level 2 5	zstd level 21 xz level 1,5 8	5	7
CALGARY	xz level 5 88	zstd level 2 103	97	56	zlib level 2 4	xz level 5 11	4	6
ENWIK9	zstd level 21 84	zstd level 2 101	98		zlib level 2 5	bzip2 level 9 xz level 5 zstd level 21 9	5	7
LHCb	snappy 72	zlib level 2 99	99	59	zlib level 2 4	xz level 1,5 zstd level 21 9	4	8
PROTEINS	zstd level 21 84	zlib level 2 zstd level 2 101	101	58	zlib level 2 4	xz level 5 11	4	7
SILESIA	snappy zstd level 21 94	zstd level 2 102	98	557	zlib level 2 5	bzip level 9 xz level 1,5 zstd level 21 8	5	7

and `zstd level 2` and `zlib level 2` having the highest CPU power consumption. `zlib level 2` used between 98 - 101 W, which is 30 W more CPU power consumption than the accelerator. The RAM power consumption of sw-based algorithms was between 4 - 11 W, with `zlib level 2` having the lowest and `xz level 5` having the highest RAM power consumption. All results for CPU and RAM power consumption are listed in Table 4. There was a direct correlation between RAM power consumption and the memory bandwidth measured for the sw-based algorithms. However, for the same RAM power consumption of 7 W, the accelerator benchmark achieved twice the memory bandwidth - the value reached for 9 W by the sw-based algorithms. The server's power consumption was with 266 W for the accelerator within the range also achieved by the different sw-based algorithms (259 - 301 W).

6 DISCUSSION

This study analyzed the integration cost of accelerators used for compression in terms of throughput, power consumption and effort to integrate into the software stack using as example the AHA378 compression card. During the study it was shown that the accelerator achieved up to 17 times the throughput compared to the sw-based equivalent of the compression algorithm (`zlib level 2`) while utilizing over 80 % less resources. The throughput achieved by the accelerator was at least 94 % of the advertised throughput.

6.1 Interpretations

6.1.1 Throughput. The accelerator's throughput was stable independent of the input data (<4 % variation), while the software-based `zlib level 2` had a throughput reduction of up to 55 % depending on the data set. This suggests that the logic on the accelerator could support a higher throughput but some (bandwidth) bottleneck prevents this. During the study it was shown that NUMA binding is important to maintain a stable and good performance. Without NUMA binding the same data input could have a decreased throughput of up to 10 Gb/s for the entire benchmark. Likwid did not deteriorate the measurements. Utilizing hyper-threading for the sw-based algorithms did not increase the throughput, but instead prevented a degradation of the performance

compared to the single core performance multiplied by the number of real cores.

6.1.2 Memory Bandwidth. The measured memory bandwidth, when using the accelerator, was between 2.8 to 3.3 times higher than the throughput of the uncompressed stream to the accelerator. For normal operations one would expect this to be a factor of 1.3 to 1.8 (1x sending uncompressed data, 1x receiving uncompressed data). This can be explained by the procedure how the data was sent to the card. In order to not be limited by slow disk I/O bandwidth, data was supplied from a circular, static memory buffer. To simulate the additional write that would come from a high-speed network card or storage device, from which the data would originate, data was then copied out of the circular buffer in properly sized chunks for the compression accelerator, which accounted for one read and write. Data is then directly read by the accelerator via DMA and sent back to host memory after compression, also via DMA. This results in 2 read/write cycles for the uncompressed input data, plus the compressed output data, plus additional overheads caused by the benchmark and the driver itself. Overall, this results in a memory bandwidth 2.8 to 3.1 times higher than the measured throughput.

6.1.3 Power Consumption. When installing the accelerator, the default power consumption is set to 25 W. If the PCIe slot supports 75 W, the kernel module can be loaded. The difference between listed power consumption (25 W) and measured power consumption (35 W), which was 10 W, can be resolved as follows: the efficiency of the power supply is about 80 %. This number was acquired by measuring the total power consumption of the server in idle mode and while running the compression benchmark full blast and expecting that - as stated by the manufacturer - the accelerator consumed 75 W. Removing the 20 % inefficiency of the measured 35 W results in a power consumption of 28 W. The slight difference to the manufacturer-stated 25 W was likely due to the power consumption of the motherboard itself and an increased inefficiency of the power supply at a lower total power consumption.

6.1.4 Comparison with Related Works. For a comparison with the works of Abdelfattah et al. [1] and Qiao et al. [21], the calgary corpus was run like described in their works. Each file part of the corpus was compressed separately and afterwards the geometric mean was calculated over all those files to get the compression ratio for the entire corpus. This was run with a subset of the sw-based algorithms, choosing algorithms with the maximum and minimum compression ratio and `zlib`. The sw-based algorithms achieved the following values: `snappy` achieved 1.85, `zlib level 2` achieved 2.65, `zstd level 2` achieved 2.76, `zstd level 21` achieved 3.26 and `xz level 5` achieved 3.38. The accelerator achieved a geometric mean compression ratio of 3.02 significantly surpassing the cited works which achieved a compression ratio of 2.17 with 24 Gb/s[1] and 2.03 with 80 Gb/s[21].

6.1.5 Applicability for Real-World Applications. The here presented pipeline-setup is a realistic model of the planned data acquisition pipeline for the LHCb experiment at CERN. For each captured particle collision the data will be filtered for interesting events, compressed in-flight and sent to storage. The compression will be done by directly streaming the data to the accelerator after the data was filtered. Thus, the data will at all times reside in the server's main memory. Only after the compression the data will be sent to permanent storage, e.g. HDDs.

6.1.6 Limitations. The authors hoped to be able to analyze the cycle activity for both benchmarks. The cycle activity would have included e.g. an analysis about the percentage of stalls, and stalls due to pending memory load for each algorithm. With this it would be possible to argue about the efficiency of each implementation, and if there is the possibility to overlap dead time due to stalls with other computational tasks. However, the server did not provide reliable results for the cycle activity, often returning no results or results which changed from run to run significantly. This was not an issue of likwid, but an issue of the server itself, as running with the Intel Performance Counter Monitor showed the same problems. Such analysis is left for future work.

6.2 Implications

In general, there is no surprise that the accelerator is faster than the software-based algorithms while using significantly less resources. However, for the system integration it is important to understand the underlying requirements so that the accelerator's performance is not degraded. To achieve the maximum throughput, the accelerator needs 4 `compressThreads` NUMA-bound to the same node to which the accelerator is attached. That means, for this particular server, only 10 % of the CPU compute resources are needed (excluding the resources for `queueThreads`, which are considered part of I/O). While compute requirements are rather low, the memory bandwidth requirements of the accelerator are notable. With up to 30 GiB/s of memory bandwidth it utilizes over 80 % of the available memory bandwidth on a single socket[4]. This memory bandwidth includes the allocation in `queueThreads` for the data to be compressed.

These results suggest that compute-intensive tasks are most compatible for a collocation on such a server, in order

[4]Note that for a dual socket the memory bandwidth will not increase by the factor of two if a single program uses the entire server. This is due to penalties in the cross-socket communication.

to utilize the unused 90 % of the CPU time. Still, NUMA binding is important to prevent unnecessary data transfers and collisions. Furthermore it allows to utilize the entire memory bandwidth efficiently on the sockets which are not connected directly to the accelerator. Possibly, the best choice would be a task which utilizes the compression as post-processing step, making sure that the data is already available in memory. Alternatively, two accelerators could be put into one server, which would then saturate the memory bandwidth.

As the benchmark used busy-waiting, it is strongly believed that the CPU overhead can still be reduced further. Future work will focus on implementing a more complex communication model based on polling to reduce the CPU overhead, while also integrating other tasks on the same server.

The results furthermore suggest that CPUs are not the perfect solution for compression algorithms. Due to their flexibility, they can either offer a higher throughput or compression ratio than FPGAs for lower implementation costs, but not both. However at the same time, CPUs are not as optimized as FPGAs for heavy integer and bit operations, which might even perform best in exotic bit lengths. It would therefore be interesting to analyze the cycle activity, especially the stalls, for CPUs and FPGAs.

In terms of integration of the AHA accelerator, the authors think the effort needed was - considering the complexity of accelerators - on the easy to moderate side. Most of the time the documentation clearly written and contained enough information to achieve good performance. However, for some API calls it was significant to read the documentation thoroughly to understand all implications. Example code was available and allowed a better understanding of the code structure needed to interact with the accelerator. Compared to other accelerators with similar functionality, the API provided did not seem to enforce over-complicated or overly redundant structures to be maintained. Furthermore, the possibility to encapsulate the process in a single thread allows to implement an easy-to-use access wrapper for high-level developers. In contrast to this, the Intel QuickAssist came with many different versions of accelerators or on-chip hardware to choose from. In the documentation it was not clear which feature is available on which device and the implementations needed more manual intervention for a working device configuration. Additionally, the results achieved were in the end not presentable. [5]

For choosing an appropriate sw-based compression algorithm 1 acan be used as reference. Independent of the data set the relative ranking of throughput and compression ratio of the algorithms should stay in the majority of cases the same.

[5]The authors only used the compression option (API and QATzip provided by Intel). The encryption functionality was not tested.

7 CONCLUSION

This study provided insights of the integration cost of accelerators used for compression from system perspective, considering power consumption and total machine utilization. The results were compared to multiple software-based algorithms, including `zlib` which was also provided by the accelerator. Compared to the software-based `zlib` the accelerator achieved equal compression ratio, but with up to 17 times increased throughput and utilizing over 80 % less CPU resources. Overall, it seems that such a commercially available compression accelerator is a good option to harvest the advantages of FPGAs of providing high throughput for well-known, reliable compression algorithms without intensive FPGA development effort. Future works to be considered are: Further improvements to reduce the workload on the CPU while maintaining the compression performance and characterizing the accelerator performance while different workloads are run in parallel on the CPU.

ACKNOWLEDGMENTS

The authors want to thank the collaborations of CERN experiments ATLAS, ALICE, CMS and LHCb for providing the data sets. This work has been sponsored by the Wolfgang Gentner Programme of the German Federal Ministry of Education and Research (grant no. 05E15CHA)

REFERENCES

[1] Mohamed S. Abdelfattah, Andrei Hagiescu, and Deshanand Singh. 2014. Gzip on a Chip: High Performance Lossless Data Compression on FPGAs Using OpenCL. In *Proceedings of the International Workshop on OpenCL 2013 & 2014* (Bristol, United Kingdom) *(IWOCL '14)*. Association for Computing Machinery, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/2664666.2664670

[2] Dr.-Ing. Jürgen Abel. [n.d.]. *Corpora.* Retrieved March 10, 2020 from http://www.data-compression.info/Corpora/index.html

[3] Mark Adler and Greg Roelofs. [n.d.]. *zlib Home Site.* Retrieved August 2, 2019 from https://www.zlib.net/

[4] Naiyong Ao, Fan Zhang, Di Wu, Douglas S. Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. 2011. Efficient Parallel Lists Intersection and Index Compression Algorithms Using Graphics Processing Units. *Proc. VLDB Endow.* 4, 8 (May 2011), 470–481. https://doi.org/10.14778/2002974.2002975

[5] Lasse Collin. [n.d.]. *XZ Utils.* Retrieved August 2, 2019 from https://tukaani.org/xz/

[6] L. Peter Deutsch. [n.d.]. *RFC 1951 - DEFLATE Compressed Data Format Specification version 1.3.* Retrieved March 9, 2020 from https://tools.ietf.org/html/rfc1951

[7] M.D. Edwards, J. Forrest, and A.E. Whelan. 1997. Acceleration of software algorithms using hardware/software co-design techniques. *Journal of Systems Architecture* 42, 9 (1997), 697 – 707. https://doi.org/10.1016/S1383-7621(96)00071-9

[8] P. Ferragina and G. Navarro. [n.d.]. *Pizza&Chili Corpus – Compressed Indexes and their Testbeds*. Retrieved March 10, 2020 from http://pizzachili.dcc.uchile.cl/texts/protein/

[9] J. Fowers, J. Kim, D. Burger, and S. 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*. 52–59. https://doi.org/10.1109/FCCM.2015.46

[10] Sameh Galal and Mark Horowitz. 2011. Energy-Efficient Floating-Point Unit Design. *IEEE Trans. Comput.* 60, 7 (July 2011), 913–922. https://doi.org/10.1109/TC.2010.121

[11] google/snappy. [n.d.]. *GitHub - google/snappy: A fast compressor/decompressor*. Retrieved August 2, 2019 from https://github.com/google/snappy

[12] AHA Products Group. [n.d.]. *AHA Products Group*. Retrieved October 25, 2019 from http://www.aha.com/

[13] M. Horowitz. 2014. 1.1 Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 10–14. https://doi.org/10.1109/ISSCC.2014.6757323

[14] Intel. [n.d.]. *Intel QuickAssist Technology (Intel QAT) Improves Data Center...* Retrieved October 25, 2019 from https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html

[15] Jseward. [n.d.]. *bzip2 : Home*. Retrieved March 9, 2020 from https://www.sourceware.org/bzip2/

[16] Anil Krishna, Timothy Heil, Nicholas Lindberg, Farnaz Toussi, and Steven VanderWiel. 2012. Hardware Acceleration in the IBM PowerEN Processor: Architecture and Performance. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (Minneapolis, Minnesota, USA) *(PACT '12)*. ACM, New York, NY, USA, 389–400. https://doi.org/10.1145/2370816.2370872

[17] S. Lee, J. Park, K. Fleming, Arvind, and J. Kim. 2011. Improving performance and lifetime of solid-state drives using hardware-accelerated compression. *IEEE Transactions on Consumer Electronics* 57, 4 (November 2011), 1732–1739. https://doi.org/10.1109/TCE.2011.6131148

[18] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. 2016. SIMD Compression and the Intersection of Sorted Integers. *Softw. Pract. Exper.* 46, 6 (June 2016), 723–749. https://doi.org/10.1002/spe.2326

[19] Lz4. [n.d.]. *LZ4 - Extremely fast compression*. Retrieved August 2, 2019 from http://www.lz4.org

[20] Matt Mahoney. [n.d.]. *About the Test Data*. Retrieved March 10, 2020 from http://mattmahoney.net/dc/textdata.html

[21] W. Qiao, J. Du, Z. Fang, M. Lo, M. F. Chang, and J. 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)*. 37–44. https://doi.org/10.1109/FCCM.2018.00015

[22] Cliff Robinson. [n.d.]. *Microsoft Project Corsica ASIC Delivers 100Gbps Zipline Performance*. Retrieved October 25, 2019 from https://www.servethehome.com/microsoft-project-corsica-asic-delivers-100gbps-zipline-performance/

[23] RRZE-HPC. [n.d.]. *GitHub - RRZE-HPC/likwid: Performance monitoring and benchmarking suite*. Retrieved Accessed on 29th October, 2019 from https://github.com/RRZE-HPC/likwid

[24] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. 2012. Database Analytics Acceleration Using FPGAs. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (Minneapolis, Minnesota, USA) *(PACT '12)*. ACM, New York, NY, USA, 411–420. https://doi.org/10.1145/2370816.2370874

[25] Jiangong Zhang, Xiaohui Long, and Torsten Suel. 2008. Performance of compressed inverted list caching in search engines. In *Proceeding of the 17th International Conference on World Wide Web 2008, WWW'08*. 387–396. https://doi.org/10.1145/1367497.1367550

[26] Zstandard. [n.d.]. *Zstandard - Real-time data compression algorithm*. Retrieved August 2, 2019 from https://facebook.github.io/zstd/

[27] M. Zukowski, S. Heman, N. Nes, and P. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *22nd International Conference on Data Engineering (ICDE'06)*. 59–59. https://doi.org/10.1109/ICDE.2006.150