Energy Efficient Multistandard Decompressor ASIP

Hoozemans, Joost; Tervo, Kati; Jaaskelainen, Pekka; Al-Ars, Zaid

DOI
10.1145/3456172.3456218

Publication date
2021

Document Version
Final published version

Published in
ICCDE 2021 - 2021 7th International Conference on Computing and Data Engineering

Energy Efficient Multistandard Decompressor ASIP

Joost Hoozemans
Delft University of Technology
Netherlands

Kati Tervo
Tampere University
Finland

Pekka Jääskeläinen
Tampere University
Finland

Zaid Al-Ars
Delft University of Technology
Netherlands

ABSTRACT

Many applications make extensive use of various forms of compression techniques for storing and communicating data. As decompression is highly regular and repetitive, it is a suitable candidate for acceleration. Examples are offloading (de)compression to a dedicated circuit on a heterogeneous System-on-Chip, or attaching FPGAs or ASICs directly to storage so they can perform these tasks on-the-fly and transparently to the application. ASIC or FPGA implementations will usually result in higher energy-efficiency compared to CPUs. Various ASIC and FPGA accelerators have been developed, but they typically target a single algorithm. However, supporting different compression algorithms could be desirable in many situations. For example, the Apache Parquet file format popular in Big Data analytics supports using different compression standards, even between blocks in a single file. This calls for a more flexible software based co-processor approach. To this end, we propose a compiler-supported Application-Specific Instruction-set Processor (ASIP) design that is able to decompress a range of lossless compression standard without FPGA reconfiguration. We perform a case study of searching a compressed database dump of the entire English Wikipedia.

KEYWORDS

FPGA, big data, ASIP, LZ77, Snappy, LZ4

ACM Reference Format:
Joost Hoozemans, Kati Tervo, Pekka Jääskeläinen, and Zaid Al-Ars. 2021. Energy Efficient Multistandard Decompressor ASIP. In *2021 7th International Conference on Computing and Data Engineering (ICCDE 2021), January 15–17, 2021, Phuket, Thailand.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3456172.3456218

1 INTRODUCTION

Data compression schemes have been in use since the dawn of the digital age, with the Lempel-Ziv papers [23, 24] serving as a basis for many variants of the popular sliding window compression algorithm. Data compression can considerably increase the efficiency of storage, hence it is a common step to perform before committing data to persistent storage. Therefore, some processors contain dedicated circuitry (ASIC) for certain algorithms, further increasing (energy) efficiency and decreasing associated performance penalties.

The challenge of these implementations is that they require considerable design effort, and are usually fixed to supporting a single standard (for example, gzip). However, new standards for data compression are actively being developed, and most large technology firms have their own variant to suit their particular needs. For example, recently Google developed Brotli and Snappy, Facebook created zstd, all of which are based on the LZ77 core algorithm. Additionally, Big Data formats such as parquet support multiple compression standards, which can even change between different sections of a single file. Clearly, these types of data sources are very difficult to offload to fixed function accelerators to perform the compression.

In this paper, we aim to address this issue by introducing an *application-specific instruction-set processor (ASIP)*, based on the *Transport-triggered architecture (TTA)* [8], that is optimized to efficiently decompress a wide range of current and predictably also future lossless compression standards. The presented prototype of the ASIP developed using the *TTA-based Co-design Environment (TCE)* [9] (also known as OpenASIP) supports the Snappy and LZ4 compression standards.

While in this paper only the FPGA-based implementation was evaluated, the goal of the proposed design is to provide energy-efficient on-the-fly decompression of data while providing control logic support for software based switching of the compression standards, thus being potentially useful also for ASIC based realization.

2 BACKGROUND AND RELATED WORK

This section briefly discusses the fields related to this work, along with the challenges we aim to address.

2.1 Compression

Overall, compression standards fall into one of the two main categories: lossy and lossless. When using lossless compression, the decompressed output is identical to the original data. Lossy compression results in some form of information loss, usually because a lower fidelity was deemed adequate for the application (for example, MP3). The design proposed in this paper targets a particular family of lossless compression standards.

Data is (losslessly) compressible when there is redundancy, that is multiple occurrences of the same (sequence of) values are found. A mathematical discourse on this topic is beyond the scope of this paper, but can be found in information theory papers such as the well-known work of Shannon [16]. Many compression standards use similar strategies to find patterns in the data that have occurred before. This way, the pattern can be replaced by a reference to the earlier data (the reference being smaller than the pattern itself, thereby reducing the required storage space). Some standards, such as for example Brotli [4], use a pre-defined dictionary containing patterns that are often used in the targeted application domain. Often the compression algorithm keeps track of a sliding *window* of data, as proposed in the paper introducing the LZ77 algorithm [23].

The compression window provides a boundary on both the memory consumption and the runtime of the algorithm, and represents a trade-off between possibly better compression ratios and memory/time requirements. A larger window provides the algorithm with more historical data to find duplicates, but this means that more memory is needed for this buffer (both for the compressor and decompressor) and the compressor needs more time to compare all the patterns. While some standards allow the user to specify the window size (e.g., Brotli), many have a fixed size (e.g., gzip with 32 kiB).

The compressed data contains *symbols* that can either be a pattern of actual values (data that is not in the sliding window) often called a *literal*, or a *reference* to an earlier pattern. Symbols need to encode their length, so that the decompressor knows when a new symbol is starting. These 'headers' (encoding whether this symbol is a literal or a reference, and the length) are the reason why compressed files can sometimes be slightly larger than the original data.

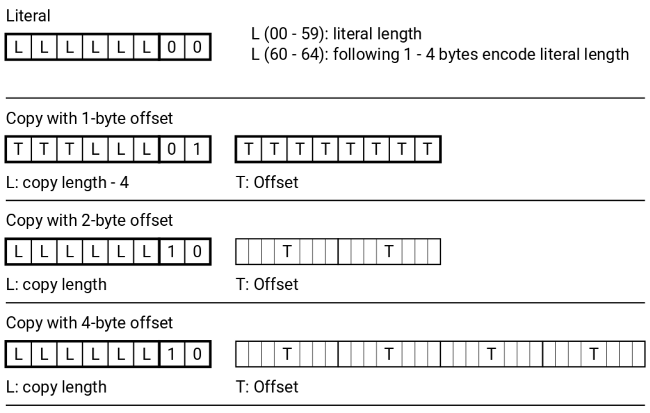

Figure 1: Encoding of different symbol types in the Snappy compression standard [11].

2.2 Compression ASICs

Several existing designs and patents exist that aim to accelerate (de)compression workloads on CPUs [1, 15], or that aim to perform compression on-the-fly when swapping pages to main memory [20]. Fixed function ASICs have the potential of providing the best possible energy-efficiency and performance, but are usually targeting a specific compression standard their lack of flexibility means the designs cannot be updated for new standards nor utilized in application cases where multiple compression formats are used in a dynamic manner.

2.3 Compression on FPGA

Various designs have been proposed that implement various compression standards in FPGA fabrics. Examples for *lossy* compression are FFT accelerators and IP cores for media codecs such as JPEG/MPEG or MP3. The compression standards targeted by our proposed design is lossless. In this domain, a multitude of FPGA-based designs are discussed in academic literature [2, 14, 22]. Additionally, several commercial IP cores accelerating GZip/ZLIB are available [7].

These designs are becoming increasingly relevant with upcoming general availability of hardware platforms that combine non-volatile memory and FPGAs on a single board. On these platforms, (de)compression can be performed on-the-fly while accessing storage. In addition to freeing compute cycles on the CPU, the FPGA is able to perform these tasks considerably more energy-efficiently and with low-latency. Combined with the complete transparency, these solutions can quickly become very cost-effective for many types of workloads.

As an example, Xilinx includes gzip IP cores in their Vitis library that can be easily included in user designs. Another use of these (de)compression cores in FPGA logic is to increase the effective bandwidth between host memory and accelerator card if the data is found to be highly compressible. This may alleviate the bottleneck that often results from the PCIe link bandwidths which have traditionally been (relatively) limited.

2.4 Compression in Big Data Analytics

Modern Big Data storage formats such as Apache Parquet [5] offer flexibility in employing various compression techniques. In addition to general algorithms, parquet can use encodings that are more effective for the type of data being stored; examples include dictionary encoding for datasets that often contains the same values, and delta encoding for storing datapoints that have relatively small differences between them. This flexibility makes supporting the standard more difficult when implementing an FPGA circuit. In addition, parquet supports using different compression algorithms for different blocks of data within a single file. This means that an FPGA circuit needs to either support several standards, resulting in logic under-utilization, or make excessive use of (partial) reconfigurations. Swapping in different accelerators based on the compression type used in the current block is difficult, because the granularity can be relatively small and in order to sustain the desired high throughput, multiple compression cores will likely be active concurrently.

As the supported compression standards are, algorithmically speaking, quite similar, it seems plausible that a middle ground between optimized circuitry and flexibility is the most optimal design space.

2.5 Application-Specific Instruction-set Processors

A middle ground between optimized circuitry targeting a specific application on one side, and providing the flexibility to target a certain range of applications in an application domain by still allowing programmability, can be created by an ASIP design. ASIPs can be extended with special instructions and have other customizable aspects that can be tuned for a certain application or an application domain. Special instructions with different granularities can be used to accelerate functions that are common within the applications in the targeted domain. Furthermore, ASIPs have the benefit of *excluding* features that are *not* required by the application domain, improving the area and energy efficiency of the design in comparison to *general purpose processors (GPP)*. When comparing to fixed function accelerators with streamlined pipeline or state machine based control logic, ASIPs include some additional overheads due to the instruction-set controlled logic circuitry required by the software-based flexibility. This includes the fetch and decode logic as well as the instruction memory needed to store the instructions of the executed program. However, an essential practical benefit of ASIPs is that they can be implemented both as ASICs on new chip designs and as soft cores in FPGAs with both platforms benefiting from the additional software-based flexibility.

On the other hand, a fully customized fixed function accelerator design requires large design and implementation effort whereas an ASIP only requires a relatively small addition of logic for the customized instructions, and the design process is toolset assisted. Examples of ASIP design and programming tools are Synopsys ASIP designer [18], Cadence Tensilica [6] and the TCE tools [9]) which were used for the design in this paper. The toolsets contain "base processors" which are design examples that can be extended and customized, and thus serve as quick starting points for new designs. The base processors contain functionality for e.g., in/output and debugging and are fully compiler programmable, helping making the design and debug cycles shorter compared to manual RTL design flows. In the recent years, high-level synthesis tools have appeared to help in the RTL design effort, however they still suffer from vendor-specific descriptions and thus steep learning curve and poor design effort reuse across different vendors.

In the end, as long as the majority of the workload can make use of the specialized instructions and other processor customizations, an ASIP can provide an excellent design point between software running on GPP and a fixed function accelerator. ASIPs have been traditionally popular in for example baseband communication, as for example mobile phones need to support several generations of wireless communication standards (2G, 3G, etc.), with little work published for the data compression application domain. A softcore ASIP design for data compression specifically for vision systems is presented in [17].

3 PROPOSED DESIGN

The design of the proposed ASIP was done using the open source TCE toolset also known as OpenASIP [19]. A leading insight for the design was to identify the common ground between the considered lossless decompression algorithms to be in the need to keep a sliding window of buffered data, and output either literal data from the input stream or data from the original buffer. This common ground between algorithms is a good target for hardware acceleration, and was done using specialized operations that can execute concurrently with the rest of the processor datapath. We implemented these operations as an RTL component that operates as a custom functional unit in the TTA processor. The input and output streams are connected directly to this unit.

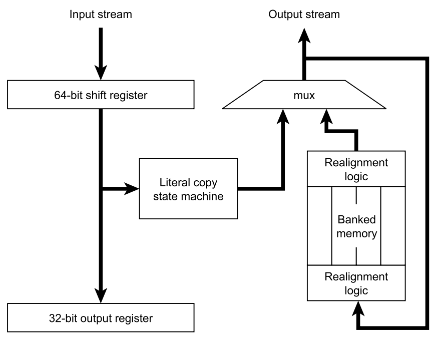

Figure 2: Internal logic of the custom function unit for stream operations.

3.1 Application-Specific Functional Unit

The ASIP contains a set of custom operations including reading symbols from the input stream into the processor, performing the transfer of literal data from the input stream to the output stream, or copying data to the output stream from an internal lookback buffer. These are all contained in one functional unit, the basic internal structure of which can be seen in Fig. 2.

The input stream is buffered with a simple byte-oriented shift register, keeping at least four bytes ready as long as the input stream does not stall. From this buffer, up to four bytes can be read into an internal register of the processor or into the output stream by custom operations. Any data written to the output stream is also written to a circular lookback buffer organized in 4 banks, configured to a total of 32 kilobytes. This lookback buffer is used to read data for references up to 32 kilobytes behind the head of the stream. The banked memory allows for reads and writes of up to four bytes to be aligned to any byte address. The custom function unit exposes these capabilities to the processor through the four operations listed in Table 1: reading 1-4 bytes of header data, transferring an N-byte literal from the input stream to the output stream, copying N bytes from an offset in the lookback memory, and signalling the last byte of the transfer, i.e., compression frame. This last operation will cause the functional unit to assert the `last` signal to the downstream consumer attached to its output.

3.2 Programmability

The decoding of the symbol headers is done using the base instruction set that supports common arithmetic and logical functions such as shifts as well as AND/OR. This instruction set it supported by the LLVM-based C-compiler included in the TCE toolchain, so that the decoding of the symbol headers is fully programmable. The

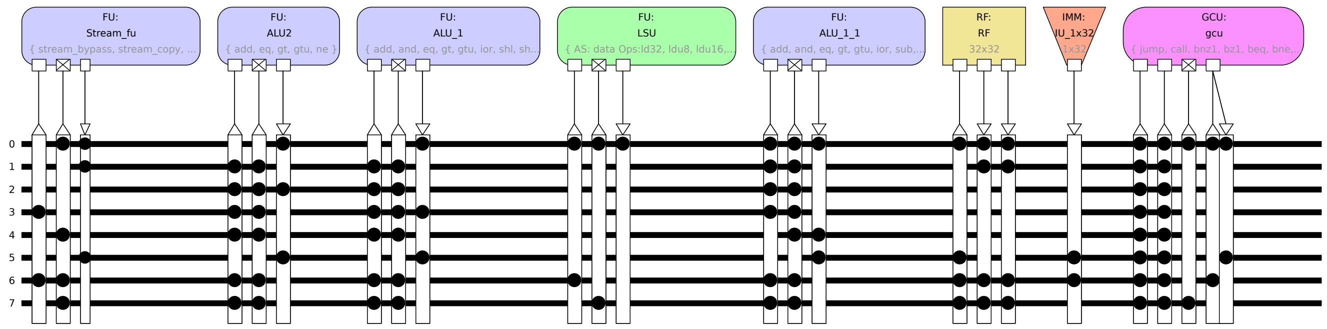
FU:
Stream_fu
{ stream_bypass, stream_copy, ...
FU:
ALU2
{ add, eq, gt, gtu, ne }
FU:
ALU_1
{ add, and, eq, gt, gtu, ior, shl, sh...
FU:
LSU
{ AS: data Ops:ld32, ldu8, ldu16,...
FU:
ALU_1_1
{ add, and, eq, gt, gtu, ior, sub,...
RF:
RF
32x32
IMM:
IU_1x32
1x32
GCU:
gcu
{ jump, call, bnz1, bz1, beq, bne,...
0
1
2
3
4
5
6
7

Figure 3: Design of the ASIP decompression processor. The design has a special function unit Stream_fu, three parallel arithmetic-logic units, and a load-store unit. All of them are supported with a simple 2r+1w register file, thanks to the TTA programming model.

Table 1: Operations supported by the custom Stream_fu functional unit.

operation	returns	parameter 1	parameter 2
READ_HEADER	1 - 4 bytes from input stream	nr. bytes to read	–
BYPASS	–	nr. of bytes to bypass	–
COPY	–	nr. of bytes to copy	offset in bytes

architecture has three parallel *arithmetic logic units (ALUs)* that support basic C operators. The ALUs are kept quite basic – each takes up under 100 LUTs in the final design.

TTA is an "exposed datapath architecture" [12] which allows software-based register file bypassing, where values can be denoted by the compiler to not be written to or read from the register file, and instead transferred directly from function unit to function unit. This model allows for a simple register file, with only two read ports and one write port, to serve three ALUs and the custom function unit whereas in a standard VLIW design it would have required a large number of register file ports, typically ending up in the critical path of the design [13].

We have implemented both the Snappy and LZ4 algorithms on the processor, and the performance of the processor for the common cases of these standards are listed in Table 2. An LZ4 symbol always has a literal and copy component, so it is more difficult to determine the exact latency (some cycles contain operations used for both).

Table 2: Decoding latency for the Snappy and LZ4 compression standards. These numbers are for the case where the arguments do not need additional bytes (Snappy; literal length < 60 or copy with 1-byte offset). The other cases require more cycles to decode, but can also result in longer sequences to output (during which the processor can continue decoding following headers).

Snappy literal	Snappy copy	LZ4 sequence (literal and copy)
14 cycles	17 cycles	28 cycles

3.3 Design Considerations

The processor is fully compiler-programmable, which means it is in principle able to decode any current and future compression standard based on the LZ77 sliding window principle. Additionally, the custom functional unit could be implemented using any interface width, and with any buffer size. However, the following practical limitations should be considered:

3.3.1 Buffer size. The design must have a buffer size that can accommodate the target compression standards. Any standard that requires support for a buffer that is larger than what is present will not be supported. conversely, buffer capacity will go to waste when decompressing standards that only require a smaller buffer. In a multi-core design it could be possible to share buffers between adjacent instances, combining multiple buffers into one, but this will render the donor instances unable to operate, thereby reducing the throughput. In addition, such an arrangement could negatively affect the cycle time because of the required additional connections and multiplexers.

3.3.2 Interface width. In this paper, we have chosen an interface of four bytes per cycle for the custom functional unit. This is the upper limit of the design in terms of throughput. For example, in an FPGA prototype with a target operating frequency of 200 MHz, this design has a theoretical maximum throughput of 4 × 200 MHz = 800 MBytes/s. However, the actual throughput is highly dependent on the properties of the compressed data stream. When the input consists of many small sequences of literals and copies, close to the interface width, the actual throughput will quickly be limited by the speed in which the processor is able to decode the headers.

Designs with small interface widths will not deliver high throughput per stream, but can be useful in cases where multiple sources of data can be processed in parallel. Wider designs can deliver high performance for a single stream, but will need to decode multiple headers per cycle.

3.3.3 Speculative header decoding. As can be seen in Figure 1, the length of a symbol header is variable. This means that the location of a new symbol depends on the length of its predecessor. Therefore, if multiple headers need to be decoded per cycle, speculation is required [10]. This means that for every new busword of input data, the processor should decode each byte as if it were the start of a symbol. As soon as the length of the previous symbol is known, the proper new symbol can be selected. Although these techniques can in principle be programmed in software (for example leveraging SIMD instructions supported by TCE), it must be noted that hardware circuits are more suitable for this because of their pipelined nature. Studying the practical implications of this is an interesting topic for future work, but outside the scope of this paper.

4 EVALUATION

In order to evaluate the performance of our proposed design in practice, we implemented a multi-softcore FPGA realization of the ASIP design. The evaluation platform was a dual-socket server with two Xeon Silver 4114 CPUs clocked at a base frequency of 2.2 GHz and a turbo frequency of max 3.0 GHz. The TDP of this processor setup is 85 watts. The machine has 8 DDR4 DIMMs of 16 GB each, running at 2666 MHz. Power measurements of the CPUs was performed using Intel Running Average Power Limit (RAPL) through the Linux powercap interface [21]. These include the power consumption of both full CPU sockets, but exclude DRAM power consumption. The FPGA platform used is a Xilinx Alveo U200 accelerator card, connected via PCIe gen3 x16. Power measurement of the FPGA is performed by monitoring the vccint voltage and current as reported by the Xilinx `xbutil` utility. These also include the power consumption of the full FPGA chip, but exclude the power consumption of the FPGA board's DRAM and other peripherals.

4.1 Case study: Wikipedia Search

As a case study to evaluate a real-world big data-like application of the design, we created software and FPGA implementations of a search program that scans through the entire English Wikipedia database dump (without index) to find and count occurrences of a given expression. Articles smaller than 64 bytes were filtered out of the dataset, as these articles are often uncompressible (they are mostly stubs and links to other articles). The raw dataset is 29 GB of text, stored in records that each contains the article title and the contents compressed using either LZ4 or Snappy (randomly selected). These records are evenly distributed across 15 chunks of 1.1 GB each for a total of 16 GB of input data. When starting the application, these chunks are distributed over 3 different banks of on-board DRAM memory. Each DRAM bank is connected to a *super logic region (SLR)* through individual memory controllers. Five kernels are assigned to each SLR, to distribute the logic utilization evenly over the FPGA. A kernel contains 3 instances of the processor for a total of 45 softcores, each connected to a simple string matching engine.

The throughput of the design in comparison with the CPU implementation is depicted in Figure 4. The 45-core FPGA prototype

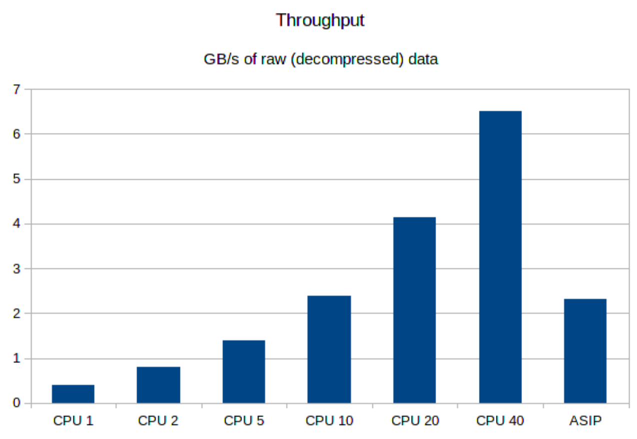

Figure 4: Throughput of the proposed design versus a dual-socket Xeon Silver 4114 system with varying number of threads.

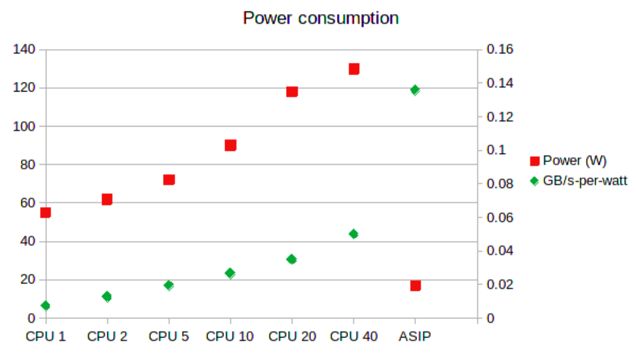

Figure 5: Power consumption and power efficiency.

reaches a throughput of 2.3 GB/s which is not enough to keep up with the full dual-socket CPU system achieving 6.5 GB/s. However, in terms of power efficiency, depicted in Figure 5, it already provides a clear advantage even through the Xeon CPUs used in the reference system are already rather energy-efficient at 86 Watt TDP. The CPU cores, when fully loaded with our test application, measure a power draw of approximately 65 Watts each. In contrast, the FPGA draws approximately 18 Watts of power when the design is active.

The resource utilization of the FPGA prototype is depicted in Table 3. A single instance of the processor consumes a very small amount of resources, so that in principle more than 100 cores could be instantiated on our FPGA platform. However, because the Alveo shell already consumes a significant amount of resources, the number of processors that could fit on the chip is more limited in our use case prototype.

5 CONCLUSION

In this work, we proposed an ASIP design that is fully compiler programmable. We demonstrate the programmability by showing

Table 3: FPGA resource utilization of a single instance of the ASIP and the full wiki search design on the Alveo U200. The Alveo shell (listed as platform resources) already consumes a significant portion of the available resources.

	single ASIP instance	45-core design (user resources)	45-core design (platform resources)
Clock frequency (MHz)	220	177	177
LUTs	4237 (0.36%)	468,092 (40%)	321,965 (27%)
Registers	1178 (0.05%)	525,100 (22%)	429,358 (18%)
BRAMs	17 (0.79%)	767 (35%)	766 (35%)
URAMs	0	0	0

that the design is able to decode both LZ4 and Snappy. To evaluate the performance, we implemented an FPGA realization of the design and integrated it into a multi-core Big Data application. This prototype using a single FPGA can not keep up with the throughput of a dual-socket CPU system, but already provides considerably improved energy efficiency. An ASIC implementation will increase this advantage even more and should provide more throughput, but we must note that the performance will stay limited by the symbol decoding latency. Mitigating this latency by using parallel decoding and introducing speculation is an interesting future research direction.

ACKNOWLEDGMENTS

This project has received funding from the ECSEL Joint Undertaking (JU) under grant agreement No 783162 (FitOptiVis [3]). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Netherlands, Czech Republic, Finland, Spain, Italy.

REFERENCES

[1] B. Abali, B. Blaner, J. Reilly, M. Klein, A. Mishra, C. B. Agricola, B. Sendir, A. Buyuktosunoglu, C. Jacobi, W. J. Starke, H. Myneni, and C. Wang. 2020. Data Compression Accelerator on IBM POWER9 and z15 Processors : Industrial Product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA45697.2020.00012

[2] 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

[3] Zaid Al-Ars, Twan Basten, Ad de Beer, Marc Geilen, Dip Goswami, Pekka Jääskeläinen, Jiří Kadlec, Marcos Martinez de Alejandro, Francesca Palumbo, Geran Peeren, and et al. 2019. The FitOptiVis ECSEL Project: Highly Efficient Distributed Embedded Image/Video Processing in Cyber-Physical Systems.

[4] Jyrki Alakuijala and Zoltan Szabadka. 2016. Brotli compressed data format. *Internet Engineering Task Force* (2016).

[5] Apache Software Foundation. [n.d.]. *Apache Parquet*. https://parquet.apache.org/

[6] Cadence. [n.d.]. *Tensilica Customizable Processor and DSP IP*. https://ip.cadence.com/ipportfolio/tensilica-ip

[7] CAST IP cores. [n.d.]. *ZipAccel-D GUNZIP/ZLIB/Inflate Data Decompression*. https://www.cast-inc.com/compression/gzip-lossless-data-compression/zipaccel-d/

[8] Henk Corporaal. 1997. *Microprocessor Architectures: from VLIW to TTA*. John Wiley & Sons, Inc.

[9] Otto Esko, Pekka Jääskeläinen, Pablo Huerta, Carlos S. de La Lama, Jarmo Takala, and Jose Ignacio Martinez. 2010. Customized Exposed Datapath Soft-Core Design Flow with Compiler Support. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications (FPL '10)*. IEEE Computer Society, Washington, DC, USA, 217–222. https://doi.org/10.1109/FPL.2010.51

[10] J. Fang, J. Chen, J. Lee, Z. Al-Ars, and H. P. Hofstee. 2019. Refine and Recycle: A Method to Increase Decompression Parallelism. In *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Vol. 2160-052X. 272–280. https://doi.org/10.1109/ASAP.2019.00017

[11] Google. [n.d.]. *Snappy; a fast compressor/decompressor*. https://github.com/google/snappy/

[12] Pekka Jääskeläinen, Heikki Kultala, Timo Viitanen, and Jarmo Takala. 2015. Code Density and Energy Efficiency of Exposed Datapath Architectures. *Journal of Signal Processing Systems* 80, 1 (2015), 49–64. https://doi.org/10.1007/s11265-014-0924-x Published Online 25 July 2014<br/>Contribution: organisation=tie,FACT1=1<br/>Portfolio EDEND: 2014-12-16<br/>Publisher name: Springer US.

[13] Pekka Jääskeläinen, Aleksi Tervo, Guillermo Paya-Vaya, Timo Viitanen, Nicolai Behmann, Jarmo Takala, and Holger Blume. 2018. Transport-Triggered Soft Cores. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. https://doi.org/10.1109/IPDPSW.2018.00022

[14] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, and K. Sheng. 2010. FPGA implementation of GZIP compression and decompression for IDC services. In *2010 International Conference on Field-Programmable Technology*. 265–268.

[15] Larry Seiler, Daqi Lin, and Cem Yuksel. 2020. Compacted CPU/GPU Data Compression via Modified Virtual Address Translation. *Proc. ACM Comput. Graph. Interact. Tech.* 3, 2, Article 19 (Aug. 2020), 18 pages. https://doi.org/10.1145/3406177

[16] Claude Elwood Shannon. 2001. A mathematical theory of communication. *ACM SIGMOBILE mobile computing and communications review* 5, 1 (2001), 3–55.

[17] Tomoki Sugiura, Jaehoon Yu, Yoshinori Takeuchi, and Masaharu Imai. 2015. A low-energy asip with flexible exponential golomb codec for lossless data compression toward artificial vision systems. In *2015 IEEE Biomedical Circuits and Systems Conference (BioCAS)*. IEEE, 1–4.

[18] Synopsys. [n.d.]. *ASIP Designer*. https://www.synopsys.com/dw/ipdir.php?ds=asip-designer

[19] Tampere University. [n.d.]. *OpenASIP*. www.openasip.org

[20] US9287893B1. [n.d.]. *ASIC block for high bandwidth LZ77 decompression*. https://patents.google.com/patent/US9287893B1/en

[21] Vince Weaver. [n.d.]. *Reading RAPL energy measurements from Linux*. http://web.eece.maine.edu/~vweaver/projects/rapl/

[22] Nils Voss, Tobias Becker, Oskar Mencer, and Georgi Gaydadjiev. 2017. Rapid Development of Gzip with MaxJ. In *Applied Reconfigurable Computing*, Stephan Wong, Antonio Carlos Beck, Koen Bertels, and Luigi Carro (Eds.). Springer International Publishing, Cham, 60–71.

[23] J. Ziv and A. Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343.

[24] J. Ziv and A. Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory* 24, 5 (1978), 530–536.