FEB3D: An Efficient FPGA-Accelerated Compression Framework for Microscopy Images

Wanqi Liu, Yewen Li, Dawei Zang, Guangming Tan

FEB^3D: An efficient FPGA-accelerated compression framework for microscopy images

Wanqi Liu[1,2], Yewen Li[1,2], Dawei Zang[1], and Guangming Tan[1,2]

[1] Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100095, China
[2] University of Chinese Academy of Sciences, Beijing 100049, China
{liuwanqi,liyewen,zangdawei,tgm}@ncic.ac.cn

Abstract. With the rapid development of fluorescence microscope technologies, high-content screening and light-sheet microscopy are producing ever-larger datasets that pose great challenges in data storing and data sharing. As a popular compression tool, B^3D introduces a noise dependent compression algorithm for microscopy images to preserve the numerical intensities of all pixel within their uncertainties by exploiting the natural variability of each pixel value. Nevertheless, the high complexity of the processing flow restricts the deployment of the tool since the throughput and power consumption cannot satisfy the increasing demand. In this paper, we propose an efficient FPGA-accelerated data compression framework based on B^3D. Following the co-design methodology, the compression processing flows are partitioned into different blocks to deploy on CPU or FPGA according to their computation characteristics. Also, we design a custom accelerator core that consists of multiple full on-chip pipelines using the channel function of the Intel OpenCL toolkit to implement data-flow driven computation. Our experiments show that the proposed framework achieves up to 32× throughput for a single pipeline compared with Intel Xeon E3-1220 v5 operating at 3.00GHz, and 6× energy-efficiency compared with GPU implementation.

Keywords: Data Compression · FPGA · Throughput.

1 Introduction

Fluorescence microscope technologies such as the high-content screening, the single molecule localization microscopy, and the light-sheet microscopy provide new opportunities for biological research by increasing the speed of imaging, the number of specimens or the resolution of the observed structures [4]. Although these new technologies push biology to a new level, the production speed and volume of experimental image data at such a fast pace still pose a formidable challenge to data processing, storage, and transmission. For example, the rate of image data generated by single-plane illumination microscopy (SPIM) [5] is about 800 MB/s with 10 TB experiment size (Table 1). The real-time data handling becomes a bottleneck for new discoveries in many cases. A straightforward solution to this problem is to perform image compression.

Table 1: Data sizes in microscopy devices.

	Imaging device	Image size	Frame rate	Data rate	Exp size
SPIM	2x sCMOS camera	2048×2048	50/s	800 MB/s	10 TB
SMLM	2x EMCCD camera	512×512	56/s	56 MB/s	500 GB
screening	CCD camera	1344×1024	8.5/s	22 MB/s	5 TB
confocal	Zeiss LSM 880	512×512	5.0/s	12.5 MB/s	50 GB
SPIM×2	4x sCMOS camera	2048×2048	100/s	1600 MB/s	20 TB

Nonetheless, microscopy image compression generally means incompatibilities with previous tools, especially in the aspects of loss control, compression ratio, and compression speed. Unlike common photos that only need preserving those structures recognized by the human visual system, scientific images, however, are obtained through sets of quantitative measurements and therefore their compression should instead preserve the numerical values of all pixel intensities within their uncertainties [4]. Although the compression ratio (original size / compressed size) can be substantially increased with lossy compression algorithms such as JPEG2000 [13], their use is often discouraged as the loss of information depends heavily on the image content and cannot be explicitly controlled.

As a novel compression toolkit for microscopy image compression, B^3D [4] [5] can compress microscopy images both in lossless mode and lossy mode with a better compression effect. In particular, its lossy mode introduces a noise dependent compression algorithm for microscopy images to preserve the numerical intensities of all pixel within their uncertainties by exploiting the natural variability of each pixel value. As a result, the user can specify the maximally tolerated pixel error in proportion to the inherent noise to control the information loss. Although B^3D can solve loss control and compression ratio problems of microscopy images compression, the high complexity of the processing flow restricts the deployment of the tool since the throughput and power consumption cannot satisfy the increasing demand, such as SPIM×2 (Table 1). For example, the throughput of single thread CPU implementation of B^3D is nearly 10 M/s, which cannot meet the requirement of data production speed for microscopy images.

In addition to CPU and GPU, Field Programmable Gate Array (FPGA) is another popular device used for algorithm acceleration because of its high parallelism, high energy efficiency, external connectivity [9], low latency, and so on. Its computation features are compatible with the characteristic of the critical blocks of B^3D. In this article, we propose an FPGA-accelerated data compression framework based on B^3D, called FEB^3D, to improve the throughput and energy efficiency of microscopy images compression tasks. Following the co-design methodology, the lossy and lossless processing flows are partitioned into different blocks to deploy on CPU or FPGA according to their computation characteristics. Also, we design a custom accelerator core that consists of multiple full on-chip pipelines using the channel function of the Intel OpenCL

toolkit [1] to implement data-flow driven computation. Our contributions can be summarized as follows:

1. Following the co-design methodology, we propose an FPGA-accelerated data compression framework based on B^3D, called FEB^3D, by partitioning the compression processing flows into different blocks to deploy on CPU or FPGA according to their computation characteristics.
2. We design a custom accelerator consisting of multiple full on-chip pipelines, which use the channel function of the Intel OpenCL toolkit to implement data-flow driven computation. Each pipeline includes six-stages which correspond to six algorithm blocks.
3. We implement a prototype according to the proposed framework and methods. The results show that the prototype achieves up to 32× throughput for a single pipeline compared with Intel Xeon E3-1220 v5 operating at 3.00GHz, and 6× energy-efficiency compared with GPU implementation.

The rest of this paper is organized as follows. In Section 2, we introduce various techniques used in image compression and background knowledge of CPU and FPGA co-design. In Section 3, we describe our hardware design, system collaboration, and further optimizations. In Section 4, we evaluate our design on Xeon CPU and Intel FPGA for microscopy images. In Section 5, we discuss the related works. In Section 6, we present our conclusion.

2 Background

2.1 B^3D Algorithm

Compression is a common and effective way to reduce the heavy burdens of massive data. Some commonly used compression algorithms such as deflate [7] only support the lossless mode and work well on sequence data. The image data compression algorithm like JPEG2000 supports lossy mode and works well on many kinds of images, but it comes at the cost of accuracy loss in the image transform domain (Fourier or Wavelet) so that the compression error of each pixel cannot be controlled and thus, is not suitable for scientific analysis. However, B^3D focuses on the microscopy image data compression, and adopts a strategy that compression errors can be controlled related to the standard deviation of the noise. It mainly involves five key steps: stabilizing transform, quantization [10], data prediction [15], run-length coding, and Huffman coding [11] (Fig. 1). Among them, the stabilizing transform and quantization are only used for the lossy mode. This paper mainly discusses the lossy mode. The detailed description of these steps is shown as follows.

– **Stabilizing transform.** This operation is the key to achieving B^3D error controlled lossy compression, including a nonlinear transformation, which transforms the input pixel intensity value to a new value, i.e., T (Eq (1)) [6]. I is the intensity value, while the rest parameters are determined by imaging sensors.

- **Quantization.** To make the transformed values more similar, each value is quantified by q. For example, if q equals 1, the quantified value will only be the multiple of 1.
- **Data prediction.** After data quantization, B^3D predicts quantified T of each pixel based on its neighboring T values (the top and the left). Then the difference between the predicted value and the original value is calculated, and this difference is often small because neighboring pixel values are similar. This operation can make the values of the whole image more concentrated.
- **Run-length coding.** Run-length coding can encode a series of repeated values (difference) into the value itself and the number of consecutive occurrences (i.e., value and count). Thanks to the good local correlation of the microscopy image data, this coding operation is able to significantly reduce the data volume.
- **Huffman coding.** Finally, Huffman coding is performed to further compress the run-length coding results by encoding less frequent symbols with more bits.

$$T = 2\sqrt{(I - offset)g + \sigma^2} \tag{1}$$

The main operations used in B^3D are listed above. There are only three steps in the lossless mode, i.e., data prediction, run-length coding, and Huffman coding. Moreover, the order of the prediction and the quantization can be changed [5]. When the prediction is before the quantization, there is a serious data dependency because the neighbor values used in the prediction should be uncompressed by their own neighbor values (last cycle). Conversely, when the prediction is after the quantization, this dependency disappears. Within a certain quantization error range, the two modes above have their own advantages and disadvantages. In some cases, the former has a better compression ratio than the latter, but it is not good for the pipeline and can lead to bad performance in FPGA. We choose the latter eventually, which achieves high performance.

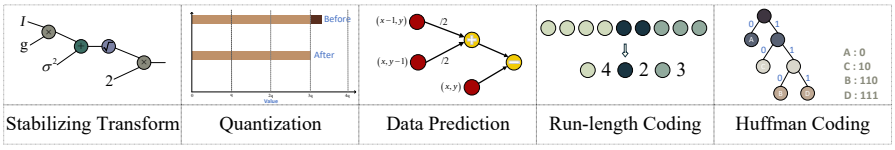

Fig. 1: Calculation steps in B^3D algorithm.

2.2 FPGA and OpenCL

Compared to CPU and GPU, FPGA provides a high level of parallelism, configurability and high power efficiency that enables custom to implement dedicated

high parallel pipelines suited to the particular needs of an application. For example, compression cores can be integrated with many specific pipelines that work parallel. In addition, various algorithmic choices can be altered depending on the characteristics of the data being processed. This flexibility may enable even higher levels of compression than a more general fixed approach, such as CPU and GPU. However, previous FPGA implementations were written in a hardware description language such as Verilog HDL or VHDL which are akin to assembly language for hardware. This makes FPGA design time-consuming and difficult to verify. Instead, OpenCL is a C-based language intended for application acceleration on heterogeneous systems. The use of OpenCL for FPGA implementation enables more productivity gains while maintaining high efficiency [3] as the generated system matches or exceeds the speed of prior work.

3 Design and Implementation

In this section, we describe the design and implementation of FEB^3D. Our main purpose is to improve the performance and energy efficiency of B^3D by mapping and optimizing the algorithm on the FPGA-accelerated platform. Therefore, we focus on the five main steps of B^3D, analyze the characteristics of the data path and each computing kernel, and then design a framework suitable for FPGA-accelerated implementation (Section 3.1). Moreover, we describe the optimized design of each kernel in detail (Section 3.2). In Section 3.3, we use further optimization methods to make our compressor work better. As already mentioned before, B^3D has both lossy and lossless modes. We mainly describe the design of lossy mode, and the lossless mode is the same as the lossy mode after removing two kernels (i.e., stabilizing transform and quantization).

3.1 The Co-design Framework

Modular and systematic thinking is extremely important for mapping software algorithms to FPGA-accelerated co-design frameworks. Therefore, we divide the tasks of the B^3D algorithm into several parts, and each part is processed by its own suitable computing resources (CPU or FPGA). Of course, most of the computing load is allocated to FPGA in our design because of its rich computing units, good parallelism, and flexible data path. In order to better divide and assign the computing tasks, we analyze the data path and the computing characteristics of the algorithm, respectively.

- **Data path.** According to the execution order of the B^3D algorithm, the microscopy image data requires to be loaded first. Then the original pixel values should be transformed into the floating-point data which remain the same number before and after. Next, the transformed values above are quantified by the quantization step q and the quantified values are then predicted by their neighbors, and we obtain the differences between their predicted values and themselves by simple subtraction. Furthermore, the differences are encoded to a series of data pairs, which include two values, i.e., the difference

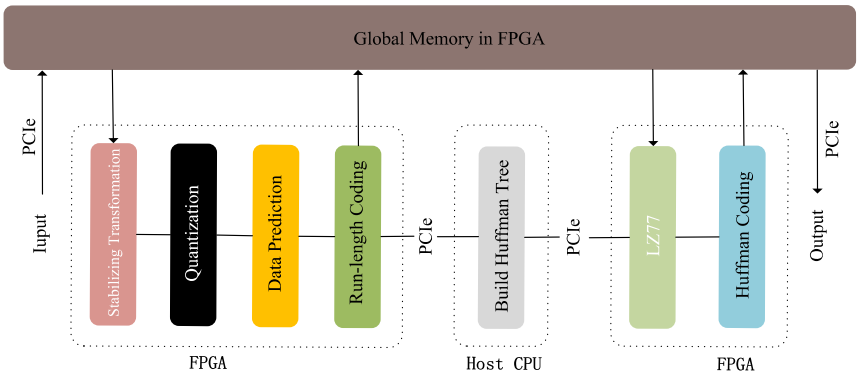

Fig. 2: Overall co-design framework of FEB^3D.

and its count. After the run-length coding, generally, the total number of data is significantly reduced. In addition, the differences and the counts are used to build a Huffman table, and then they are encoded by this Huffman table. The output data of the Huffman coding is a bitstream. At the end of the compression process, this bitstream is written to a file in a particular format. The data path from the original image to the compressed file is as described above. Obviously, the entire calculation process is streaming, which is suitable for pipeline design.

– **Computing characteristics.** One of the most critical features of B^3D is the error control, which is implemented by stabilizing transform. In order to avoid the reduction of compression ratio and compressed image quality, the transformed value should be the floating-point type. Therefore, there are a large number of floating-point calculations in this algorithm, which is a heavy computing burden for the CPU because of its relative lack of computing resources. Moreover, there are some complex operations like square root (eq (1)) in the step of the stabilizing transform and division operations in the quantization, which are also not the CPU friendly computing characteristics. This algorithm has a high degree of data parallelism, hence, FPGA is able to take advantage of its computing resources and get better parallel design. 90% of the computing time is spent on data loading, stabilizing transform, quantization, data prediction, run-length coding and generation of Huffman codes implemented on FPGA, and the rest 10% of time consumption is for constructing Huffman tree implemented on CPU. LAWRENCE L. LARMORE et al. [12] proposed a method to construct Huffman tree in parallel, but for us it is hard to be implemented efficiently on FPGA. This will be considered as the future work.

Based on the analysis above, we can draw the following ideas about the co-design framework of FEB^3D as Fig. 2.

1. According to the analysis of the computing characteristics, we can come up with the allocation plan for the computing tasks: The construction of the Huffman tree is assigned to the CPU, and the rest is assigned to the FPGA.
2. The pipeline design is good for implementing this algorithm. We build a six-stage pipeline, including data loading, stabilizing transform, quantization, prediction, run-length coding, and Huffman coding. The first five kernels are communicated via FIFO, and the last kernel is connected with CPU, communicated via PCIe.

3.2 Hardware Optimized Design

After the framework of the algorithm is determined, each computing kernel requires to be implemented on the hardware, i.e., FPGA. Furthermore, in order to achieve higher performance, we optimize the hardware level design based on the characteristics of each kernel, and also fully consider the connection between kernels (Fig. 3).

We notice that for the pipeline design, both the execution speed of each single computing kernel and the balance between each kernel speed are equally significant. For instance, if most of the kernels are running slow (without effective optimizations), the overall performance will undoubtedly be very tremendously low. On the other hand, if there is only one bad-performance kernel (the other kernels perform well), this bad-performance kernel will become the performance bottleneck of the entire system, which seriously reduces the overall performance. For the reasons above, we follow the two principles below when optimizing the hardware design: The first is to accelerate each computing kernel as much as we can and the second is to reduce the performance difference between the six kernels as much as possible.

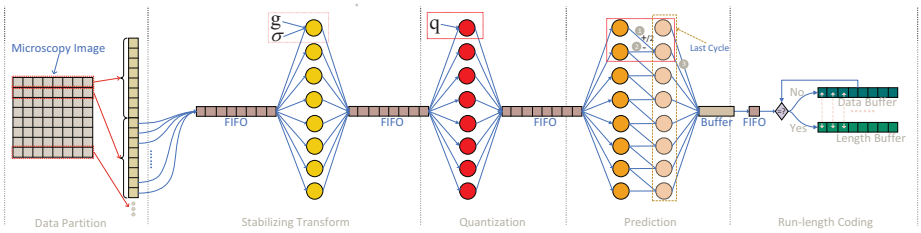

Fig. 3: Data flow and Hardware optimized design.

– **Data loading.** This is the first kernel performed on FPGA. In our design, the microscopy image data (TIFF, such as Zebrafish) is read from the disk to the memory by CPU, and then we use the OpenCL function interface to write the data in the CPU memory to the global memory of the FPGA via PCIe. When the above process is completed, FPGA reads data from its

global memory. Considering the advantages of the FPGA, we read a block of data in one cycle, i.e., eight unsigned short integers (Type $ushort8$ in the OpenCL), instead of a single unsigned short integer. This operation is effective for FPGA because of the increasing of the reading rate, which also reflects the flexibility of the data type and the data path on FPGA. During the same cycle, this kernel writes an $unshort8$ variable (containing eight unsigned short integers, i.e., sixteen bytes) to the FIFO, and then waits for the next cycle. This $ushort8$ variable is stored in the latch (consists of on-chip logic units, not the memory), which is the fastest way. The main operations of this kernel are the data reading (global memory) and the data writing (FIFO) rather than the complex calculation. Therefore, the speed of the data loading kernel is high.

- **Stabilizing transform.** This is the second kernel performed on FPGA, which is aimed to transform the input data to another data with eq (1), based on the camera parameters (such as the read noise). The input data of this kernel in a cycle is an $ushort8$ variable, received from the data loading kernel via FIFO. The data parallelism of this kernel is tremendously high, because there is nearly no dependency on the input $ushort8$ variable, i.e., eight unsigned short variables. In other words, these eight variables are calculated by themselves, using eq (1), and mapped to the hardware, which means that there are eight computing units performed in parallel during the same cycle. Although the computing complexity of eq (1) is not low, this concurrent design helps the stabilizing transform kernel achieve good performance. The output of this kernel is an $float8$ variable (i.e., eight floating-point variables), which is then written to the FIFO at the end of this cycle.
- **Data quantization.** In our design, the data quantization is the third kernel. As mentioned before (Section 2.1), there will be severe data dependency if the prediction is performed before the quantization. Therefore we choose the scheme that the prediction is performed after the quantization. As for a single variable, the quantization step includes multiplication and a division. When the transformed $float8$ variable is received by the quantization kernel, these eight floating-point variables are calculated in parallel with the same operations. The output of the quantization kernel is also an $float8$ variable, which is then sent to the FIFO.
- **Data prediction and Run-length coding.** The data prediction kernel is the fourth. After the quantization of eight floating-point variables, the predicted values ($float$) are calculated by their neighbors (i.e, the left and the top), and the differences ($float$) between the predicted values and the original values are obtained in parallel. In Fig. 3, the top comes from the last cycle. Unlike the previous kernels, the output of the prediction kernel is not $float8$, but $float$. The reason is that the run-length kernel will be obviously slower than others if eight floating-point variables are processed in the run-length kernel together. The run-length step compares the neighbors, and records the counts one by one, limited by data dependence, thereby leading to bad performance. We adopt a pipeline design on FPGA, so the performance of the kernels requires to be balanced. In order to solve this

problem, we adopt a data buffering strategy, which means that during one cycle, the prediction kernel only sends a single $float$ variable to the run-length kernel via FIFO, and the rest seven $float$ variables are stored in the latch, waiting for the next cycle. As for the run-length kernel, the input is a $float$ variable. At the end of this kernel, three kinds of variables are written to the global memory, i.e., values, values counts, and the length of data, and after that they are sent to the CPU via PCIe for constructing the Huffman tree and obtaining the Huffman table.

– **LZ77 & Huffman coding.** In our design, in order to get a better compression ratio, the Huffman coding kernel is updated to the LZ77 [18] and Huffman kernel, based on the proposal of Mohamed S. Abdelfattah et al. [3]. This is the last kernel on FPGA. The inputs are the outputs of the run-length kernel and the Huffman table computed by CPU. The outputs are a bitstream and CPU reads it via PCIe. It is worth noting that this kernel is not the bottleneck (nearly 3GB/s), and the throughput of the whole design is no more than 2GB/s.

3.3 Further Optimizations

In addition to the framework design and specific hardware optimized design, we also propose further optimizations for higher performance as follows.

1. **Further Parallelism and Data Partition.** The previous proposed six-stage flow design is with a single pipeline. Today, there are abundant resources on FPGA, making it possible to duplicate multiple pipelines (parallel execution). Moreover, the data loading, the stabilizing transform, the data quantization, the data prediction, and the run-length coding kernels are tremendously lightweight. Therefore, we duplicate five copies of the above kernels. This further parallelism makes use of the bandwidth of PCIe and the resources of FPGA, which further improves performance (nearly 6×). Correspondingly, in order to cope with the pipeline duplication, the data partition is necessary. As is shown in Fig. 3, We divide the image data by row, and each pipeline processes multiple rows. Moreover, it is worth noting that the number of rows is required to be at least two because the predicted value is calculated by its left value and its top value, as shown in Section 2.1. The data partition can influence the compress ratio, and the number of rows is able to be adjusted.
2. **The Strategy of Building Huffman Tree.** Building the Huffman tree is processed on CPU, which nearly takes up 10% of the whole time. In order to further improve performance, we need to reduce the time consumption of this step. We put together one hundred images into a single image, and build the Huffman tree for this single image instead of building a hundred trees. Furthermore, inspired by Bulent Abali et al. [2], w e notice that it takes a long time to count the frequency of each symbol for building the Huffman tree because of the passing of massive input data. Therefore, We only count the frequencies of a part of the input data (10%), selected randomly.

4 Experiment and Evaluation

4.1 Experimental Setup

- **Platform.** There are three different experimental platforms, including CPU, GPU and FPGA. The CPU is an Intel Xeon E3-1220 v5 (80W) running at 3.00 GHZ. The GPU is an NVIDIA GeForce GTX 970 (165W) graphics processing unit, which is used by Bálint Balázs et al. [4]. The last platform is a Intel Arria 10 GX FPGA (42W) device. The above platforms are of similar magnitude (None of them are the latest products). Our design is implemented with Intel OpenCL SDK. Software designs are compiled using Microsoft Visual Studio 2012. Moreover, We use the Intel OpenCL IP function *write_channel_intel*() for the FIFO design.
- **Datasets.** We use two kinds of fluorescence microscopy image data as our datasets. One is the images of Zebrafish, and the another is the images of the brain of Caenorhabditis elegans. There are 100 images (400 M in size) in both two datasets.
- **Baselines.** We have two baselines for comparison: our implementation of CPU-based B^3D using C++ and the B^3D library based on GPU, proposed by Bálint Balázs et al. [4] [5].

4.2 Results and Analysis

In our design, a key point is to reduce the bottleneck of the pipeline, so that the data flow is executed quickly and smoothly. As is described in Section 3.2, the execution time of the run-length coding kernel is significantly longer than other kernels, making it a main bottleneck of the pipeline. Therefore, we design a buffer in the prediction kernel to reduce the burden of the run-length coding kernel (Fig. 3). We adjust the buffer size and measure the execution time and resource consumption of the first five pipeline stages. As is shown in Fig. 4, the larger the buffer size, the shorter the running time and the lower the resource usage. The reason for shortened running time is that the larger buffer can better balance the execution time of the first five kernels (8 is enough). The resource is saved is because the larger buffer reduces the hardware complexity of the run-length coding kernel, which saves far more resources than that brought about by a larger buffer.

We compare our FPGA-accelerated design with the above baselines. We refer the single pipeline design as FPGA, and the further optimized FPGA as the design with six pipelines (excepts LZ77 and Huffman coding) and the use of the strategy of building the Huffman tree in Section 3.3.

Table 2: Throughput of CPU and FPGA implementation (MB/s).

	CPU	**CPU**×8	**FPGA**	**FPGA(FO)**
Zebrafish	9	46	285	1782
Elegans	6	31	205	1282

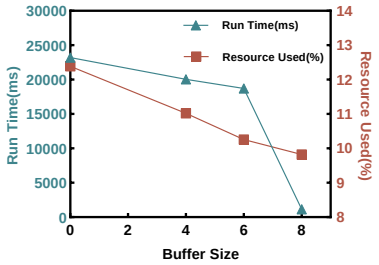

Fig. 4: Buffer Sizes.

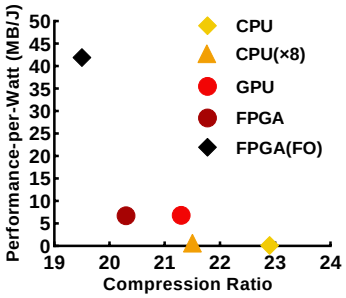

Fig. 5: Efficiency and Ratios.

1. **FPGA versus CPU.** We evaluate our FPGA-accelerated design and the CPU implementation on the above two datasets. As is shown in Table 2, the throughput of FPGA design far exceeds that of single-thread CPU (32×). Even for the multi-thread CPU implementation (×8), the performance is worse than the single pipeline FPGA design. Moreover, the performance-per-watt of FPGA is also better than that of CPU (Fig. 5). The compression ratio of FPGA is lower because of the parallelism design and reducing symbols of the frequency count (Section 3.3).
2. **FPGA versus GPU.** GPU implementation enjoy high performance (beyond 1 GB/s). From Fig. 5 we can know that the performance-per-watt of the single pipeline FPGA-based design is close to that of GPU because of the high power consumption of GPU and the further optimized one is much better than GPU (6×).
3. **Resource utilization.** Table 3 demonstrates that the first five kernels consume far fewer resources than the last kernel (LZ77 and Huffman) so that we can duplicate multiple copies for the fist five kernels.

Table 3: Resource utilization from synthesis for single pipeline.

Kernel Name	ALUTs	FFs	RAMs	DSPs
load data	2291	2188	13	0
photon transform	9477	6407	24	29
prediction	2677	2964	0	8
quantization	14801	10133	37	36
rle coding	5080	12660	82	0
lz77 huffman	315228	149105	1430	0
Total	**430317 (55%)**	**341658 (22%)**	**1873 (75%)**	**140 (10%)**

5 Related Work

There have been many studies on data compression techniques. JPEG2000 and LZW [14] are both widely used in conventional image compression, but they are not suitable for scientific image data because of the defects of pixel value

error control and compression ratio. SZ [8] is a popular error-bounded lossy compression framework for scientific data, but it suffers from low compression throughput limited by CPU implementation. The reason is that the CPU has hardware disadvantages in terms of parallelism and computing resources. B^3D introduce a noise dependent compression algorithm for microscopy images to preserve the numerical intensities of all pixel within their uncertainties by exploiting the natural variability of each pixel values. However, CPU-based B^3D has a low throughput, which cannot meet the requirement of the microscopy image data production rate obviously. GPU-based lossless B^3D gets a performance improvement compared to the CPU implementation, but the power consumption is also really high. Moreover, GPU generally improves performance through SIMT (single instruction, multiple threads) [16], accompanied by a certain synchronization overhead. Field programmable gate array (FPGA) is popular in the algorithm acceleration field because of its configurability, high energy efficiency, low latency, external connection, and so on. Xiong et al. proposed GhostSZ [17], and Tian et al. proposed WaveSZ [16], both of them implemented SZ algorithm based on FPGA and got performance improvement. B^3D is more suitable for microscopy image data because the stabilizing transform step is based on camera parameters, and the run-length coding step is also suitable for this kind of image data. As we know, FEB^3D is the first FPGA-accelerated framework based on B^3D, and we improve both throughput and energy efficiency through customized hardware co-design.

6 Conclusion

In this work, we propose an efficient FPGA-accelerated data compression framework called FEB^3D, which can improve the performance and the energy efficiency of B^3D, an effective compression algorithm for fluorescence microscopy images both in lossy and lossless modes. We properly assign the computing tasks to FPGA and CPU, and design each hardware kernel according to their characteristics. Moreover, we also adopt further optimizations both in hardware and software in order to further improve performance. The throughput of this framework far exceeds the CPU implementation (32× for the single pipeline), and the performance-per-watt of the further optimized FPGA design is about 6× higher than that of the GPU implementation.

7 Acknowledgments

This work is supported by the National Natural Science Foundation of China under grant no. 61902373, is supported by Strategic Priority Research Program B of the Chinese Academy of Sciences under grant no. XDB24050300, grant no. XDB44030300.

References

1. Intel FPGAs and Programmable Devices - Intel FPGA, `https://www.intel.com/content/www/us/en/products/programmable.html`, library Catalog: www.intel.com
2. Abali, B., Blaner, B., Reilly, J., Klein, M., Mishra, A., Agricola, C.B.: Data Compression Accelerator on IBM POWER9 and z15 Processors. In: ISCA. IEEE, Los Angeles, CA (2020)
3. Abdelfattah, M.S., Hagiescu, A., Singh, D.: Gzip on a chip: high performance lossless data compression on FPGAs using OpenCL. In: IWOCL. pp. 1–9. ACM Press, Bristol, United Kingdom (2014)
4. Balázs, B., Deschamps, J., Albert, M., Ries, J., Hufnagel, L.: A real-time compression library for microscopy images. bioRxiv (Jul 2017)
5. Balázs, B., Deschamps, J., Albert, M., Ries, J., Hufnagel, L.: A real-time compression library for microscopy images Supplementary Notes and Figures. bioRxiv p. 15 (2017)
6. Bernstein, G.M., Bebek, C., Rhodes, J., Stoughton, C., Vanderveld, R.A., Yeh, P.: Noise and Bias In Square-Root Compression Schemes. PUBL ASTRON SOC PAC **122**(889), 336–346 (Mar 2010)
7. Deutsch, P.: DEFLATE Compressed Data Format Specification version 1.3. Tech. Rep. RFC1951, RFC Editor (May 1996)
8. Di, S., Cappello, F.: Fast Error-Bounded Lossy HPC Data Compression with SZ. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 730–739. IEEE, Chicago, IL, USA (May 2016)
9. Geng, T., Wang, T., Wu, C., Yang, C., Wu, W., Li, A., Herbordt, M.C.: O3BNN: an out-of-order architecture for high-performance binarized neural network inference with fine-grained pruning. In: Proceedings of the ACM International Conference on Supercomputing. pp. 461–472. ACM, Phoenix Arizona (Jun 2019)
10. Gray, R., Neuhoff, D.: Quantization. IEEE Transactions on Information Theory **44**(6), 2325–2383 (Oct 1998), conference Name: IEEE Transactions on Information Theory
11. Huffman, D.: A Method for the Construction of Minimum-Redundancy Codes. Proc. IRE **40**(9), 1098–1101 (Sep 1952)
12. Larmore, L.L., Przytycka, T.M.: Constructing Huffman Trees in Parallel. SIAM J. Comput. **24**(6), 1163–1169 (Dec 1995)
13. Rabbani, M., Joshi, R.: An overview of the JPEG 2000 still image compression standard. Signal Processing: Image Communication **17**(1), 3–48 (Jan 2002)
14. Savari, S.A.: Redundancy of the Lempel-Ziv-Welch code. In: Data Compression Conference, 1997. DCC '97. Proceedings (1997)
15. Tao, D., Di, S., Chen, Z., Cappello, F.: Significantly Improving Lossy Compression for Scientific Data Sets Based on Multidimensional Prediction and Error-Controlled Quantization. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 1129–1139. IEEE, Orlando, FL, USA (May 2017)
16. Tian, J., Di, S., Zhang, C., Liang, X., Jin, S., Cheng, D., Tao, D., Cappello, F.: waveSZ: a hardware-algorithm co-design of efficient lossy compression for scientific data. In: PPoPP. pp. 74–88. ACM, San Diego California (Feb 2020)
17. Xiong, Q., Patel, R., Yang, C., Geng, T., Skjellum, A., Herbordt, M.C.: GhostSZ: A Transparent FPGA-Accelerated Lossy Compression Framework. In: FCCM. pp. 258–266. IEEE, San Diego, CA, USA (Apr 2019)
18. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inform. Theory **23**(3), 337–343 (May 1977)