Extending the PCIe Interface with Parallel Compression/Decompression Hardware for Energy and Performance Optimization

Mohd Amiruddin Zainol
Department of Electrical and Electronic Engineering,
University of Bristol,
Bristol, United Kingdom
Mb14650@bristol.ac.uk

Jose Luis Nunez-Yanez
Department of Electrical and Electronic Engineering,
University of Bristol,
Bristol, United Kingdom
j.l.nunez-yanez@bristol.ac.uk

Abstract—PCIe is a high-performing interface used to move data from a central host PC to an accelerator such as Field Programmable Gate Arrays (FPGA). This interface allows a system to perform fast data transfers in High-Performance Computing (HPC) and provide a performance boost. However, HPC systems normally require large datasets, and in these situations PCIe can become a bottleneck. To address this issue, we propose an open-source hardware compression/decompression system that can be used to adapt with continuously-streamed data with low latency and high throughput. We implement a compressor and decompressor engines on FPGA, scale up with multiple engines working in parallel, and evaluate the energy reduction and performance with different numbers of multiple engines. To alleviate the performance bottleneck in the processor acting as a controller, we propose a hardware scheduler to fairly distribute the datasets among the engines. Our design reduces the transmission time in PCIe, and the results show an energy reduction of up to 48% in the PCIe transfers, thanks to the decrease in the number of bits that have to be transmitted. The overhead in terms of latency is maintained to a minimum and user selectable depending on the tolerances of the intended application.

Keywords*- PCIe, FPGA, data compression, energy efficiency, parallel hardware*

*****

I. Introduction

Over the past few decades, traditional multi-core CPUs in High-Performance Computing (HPC) platforms have been adding accelerators using General Purpose Graphics Processing Units (GPU) [1][2] and/or Field Programmable Gate Arrays (FPGA) [3][4][5]. In a typical configuration, these accelerators are built as a coprocessor board to perform high-speed data processing and interfaced to the central host via a Peripheral Component Interconnect Express (PCIe). The amount of data that has to be moved and processed has continued to grow as new data center and HPC applications focus on data analytics, web searches, and virtual reality evolve. The PCIe interface is seen as a potential source of a bottleneck in the system, and current efforts are focused on integrating the host and accelerator tightly-coupled in the same device with a shared memory system [6]. Despite these developments, PCIe remains a popular choice, and the transmission of significant amounts of data reduces the performance and increases the energy requirements —as well as the cost of the utility bill.

Lossless data compression is one of the effective approaches to reduce not only the data size without affecting the original content but also the overall energy consumption. The reason behind this approach is that a compressed dataset can be transferred using less time, which reduces the energy per transfer. Furthermore, a high compression ratio yields fewer bytes of data requiring transmission, and this method can significantly improve the bandwidth in the PCIe. By reducing the transmission time, the energy efficiency can be improved as long as the compression overheads are low. This work proposes a novel open-source PCIe core based on a streaming data compression called CPCIe (Compression-enabled PCIe) and evaluates the extent to which the data compression/decompression implemented on the hardware can reduce the energy consumption and improve the overall performance. The contributions of this paper can be summarized as follows:

- We extend our open-source CPCIe core [7] by implementing a hardware scheduler based on first-in first-out scheduling scheme. Compared with our previous original work presented in [8] which had limitation to configure the route for AXI4-Stream channels, this paper presents a novelty that the hardware scheduler can distribute the block of datasets to multiple compressor and decompressor engines without any intervention from a soft processor,
- We create an efficient parallel system framework to communicate the scheduler with multiple compressor/decompressor engine cores. This framework enables the system to effectively hide the latencies from one engine to another engines and guarantees for fast compression or decompression output without requiring synchronization from a processor,
- We develop a software API for the host PC to facilitate the use of the CPCIe framework. Our open-source API can integrated by the developers to work with their own datasets and hardware accelerators,
- We demonstrate how a hardware-accelerated application can be integrated into our architecture using two realistic benchmarks. The first case study was based on matrix multiplication. This work has now been extended with a second case study called hotspot, which uses a 3x3 sliding window to form a convolution. The accelerators are created with high-level synthesis tool for rapid-prototyping development,
- Finally, we evaluate the benefits of the data compression implementation in a prototype with the hardware

accelerator, and perform accurate energy and performance measurements.

The whole system with integrated compression has been released open-source [7] to encourage third party testing and promote further work in the area. The remainder of this paper is structured as follows. Section 2 describes related work. Section 3 presents the proposed system design. Section 4 evaluates the main features of the demonstration system. Section 5 briefly describes the applications used with our CPCIe system. Section 6 evaluates the energy and performance with different configurations that include the original CPCIe, a CPCIe system with a single compressor/decompressor engine, and a CPCIe system with multiple compressor/decompressor engines. Finally, section 7 concludes the paper.

II. Related Work

Modern FPGAs are used as hardware accelerators because they can create custom circuits using millions of uncommitted logic elements, hardened PCIe interfaces, and can achieve peak performance over several GFLOPs, thanks to dedicated floating point resources [9]. This high throughput results in high requirements on the memory bandwidth and the corresponding interfaces to achieve maximum performance. Despite these challenges, there are many examples of the successful acceleration of compute-intensive applications using FPGAs interfaced via PCIe to a host PC [10], [11], [12].

Compression is a useful technique to reduce the storage requirement demands in the local workstation or cloud. There are two techniques that are used for compression: lossless or lossy compression. In most HPC applications, lossless compression is chosen because the original content in the datasets must be compressed without any bit losses in the data, and a slight change in this sensitive data after decompression is unacceptable. The lossy compression technique can be applied to other data types, such as video or image processing, since the losses bit in the decompressed output is tolerable. In this work, we are interested in using lossless compression for our evaluations.

An early study by Tremaine et. al. [30] used the IBM MXT algorithm to perform lossless data compression in the FPGA and their work has contributed to other research on hardware lossless data compression, as summarized in Table 1. Recently, the deployment of data compression in data centers and data warehouse systems has become a topic of interest in this research area. Putnam et. al [31] presented Catapult, which

Table 1: Comparison of existing lossless data compression on FPGA (sorted by year). A. = Altera, X. = Xilinx.

Work (Year)	Compression Algorithm	Freq. of FPGA	Throughput of Com. & Decom.	Technology (fabrication)	Power reported?	Stream to Acc.?	Open sources?
Jose et. al. (2003) [13, 14]	X-MatchPRO	50 MHz	C: 200 MB/s D: 200 MB/s	X. Virtex-E (180 nm)	No	No	Yes
Lin et. al. (2006) [15]	PDLZW+AHDB	100 MHz	C: 125 MB/s D: 83 MB/s	Simulated on 350 nm 2P4M	700 mW	No	No
Papadopoulos et. al (2008) [16]	LZ77	167 MHz	C: 1096 MB/s D: -	X. Virtex-5 (65 nm)	No	No	No
Zaretsky et. al. (2009) [17]	ZLIB	120 MHz	C: - D: 125 MB/s	X. Virtex-5 (65 nm)	No	No	No
Ouyang et. al. (2010) [18]	GZIP	132 MHz	C: 110 MB/s D: 306 MB/s	A. Cyclone-III (65 nm)	No	No	No
Villafranca et. al. (2010) [19]	FAPEC	48 MHz	C: 28 MB/s D: -	PROASIC3L (130 nm)	35 mW	No	No
Naqvi et. al. (2011) [20]	LZW	50 MHz	C: 88 MB/s D: 160 MB/s	X. Virtex-II (120 nm)	333 mW	No	No
Sukhwani et. al. (2011) [21]	842B	125 MHz	C: 1024 MB/s D: 1024 MB/s	A. Stratix IV (40 nm)	No	No	No
Shcherbakov et. al. (2012) [22] [23]	LZSS LZMA	100 MHz	C: 50 MB/s D: -	X. Virtex-5 (65 nm)	No	No	No
Hogawa et. al. (2013) [24]	GZIP	47 MHz	C: - D: 91 MB/s	X. Virtex-5 (65 nm)	No	SoR	No
Li et. al. (2014) [25]	LZMA	159 MHz	C: 16 MB/s D: -	X. Virtex-6 (40 nm)	No	No	No
Abdelfattah et. al. (2014) [26]	GZIP	193 MHz	C: 2867 MB/s D: -	A. Stratix-V (28 nm)	25 W (PC & FPGA)	No	No
Li et. al. (2015) [27]	DEFLATE	-	C: 432 MB/s D: -	X. Virtex-7 (28 nm)	No	No	No
Najmabadi et. al. (2015) [28]	htANS (based on ANS and tANS)	162 MHz	C: 146 MB/s D: -	X. Virtex-6 (40 nm)	No	No	Partially
Wijeyasinghe et. al. (2017) [29]	varies: RGB32/24, Delta, Float, D-S	200 MHz	from 579 MB/s to 5058 MB/s	X. Virtex-6 (40 nm)	No	FFT	No
This paper	X-MatchPRO	100 MHz	C: 659 MB/s D: 776 MB/s	X. Virtex-7 (28 nm)	19 mW to 131 mW	Yes	Yes

integrates reconfigurable hardware accelerators in data centres to improve the performance of the Microsoft's Bing search engine. They form a network of eight-node FPGA groups via high-speed serial links and each node performs a specific function. One of the nodes is the compression stage, and hence it is able save the bandwidth during transmission and increase the efficiency between the nodes. This project accelerated the scoring engines to rank the search results in Bing by a factor of 2X. This work is part of the research at Microsoft Research presented in [32] [33]. Delmerico et. al. [34] have developed several cluster computing systems for computing human genes. One of the systems used is the Netezza [35][36] data warehouse, which can process a huge amount of streaming datasets from storage disks. The Netezza comprises an FPGA close to the storage disk, along with a PowerPC processor and a memory system. However, the FPGA was only used to decompress the pre-compressed data from the disk during query processing, while the other computations were performed by the processor. Nevertheless, they reported that this approach improved the query processing time by 3 times and outperformed the parallel computation on the PC cluster.

Compared with previous work, our solution offers lower latency limited to tens of clock cycles and is transparent to the user, as it is tightly coupled to an open-source PCIe core. The low latency and high throughput are obtained with hardware compression that matches the interface width with the compression word width and with a fully streaming pipeline. Furthermore, the previous work is proprietary, commercial and closed source, while in our research, we have made it open-source so that further research and optimizations are possible. CPCIe is agnostic to the acceleration function being implemented and has been tested with functions created in hardware using high-level synthesis tools. The first results of this work were presented in [8] and in this paper we extend that work with new benchmarks and a parallel version of the hardware and software system.

III. Design and Architecture of CPCIe

In this section, we show how we developed the overall system to perform data compression and decompression while providing streamed data for PCIe communication. In Fig. 1 (a), our proposed CPCIe is located between the host PC and the hardware accelerator. This is implemented on the FPGA development board, which supports the capability of the PCIestreamed interface. Fig. 1 (b) shows the detail of the CPCIe architecture. On the left hand side, the communication is performed between the host PC and the CPCIe via the PCIe interface, while on the right hand side the data transmission occurs between the CPCIe and the hardware accelerator via the AXI-4 Stream interface. The architecture consists of five major components and two minor components. The major components are the MicroBlaze processor, the Xillybus IP Core, a custom interface for Xillybus, the Compressor System and the Decompressor System. The two minor components are the UART and the Power Monitor, which are used to display the output and read the power readings, respectively. More details regarding the power readings are discussed in Section IV.

In our proposed design, the commands are issued by the host PC and acknowledged by the MicroBlaze processor in the CPCIe. The operational model of the host PC is shown in the following order:

1. Send the block size and threshold values that will be used for the compression and decompression. These two values are saved in the Dual-Port Shared Memory in the CPCIe.
2. Send the command to start the CPCIe with two options: either to use the compression/decompression mode or not to use it at all (send it as original file).
3. Wait for an acknowledgement signal from the CPCIe.
4. Send the datasets to the input buffer of the FPGA.
5. Retrieve the results from the output buffer of the FPGA.
6. Wait until the CPCIe indicates the execution has completed.

While on the FPGA side, the CPCIe will:

1. Wait for the command from the host PC.
2. The MicroBlaze processor configures the Compressor and Decompressor Systems based on the block size and threshold values given by the host PC. These two values are read from the Dual-Port Shared Memory.
3. The MicroBlaze sends an acknowledgement signal to the host PC.
4. The Decompressor System receives the input, then decompresses the compressed dataset, and sends it to the hardware accelerator.
5. The Compressor System then receives the output results from the hardware accelerator, compresses the output, and sends it back to the host PC.
6. Upon completion, the MicroBlaze sends done signal to the host PC.

In the next subsection, we first describe in detail our CPCIe architecture. Then, we show how our Compressor and Decompressor Systems can be used to distribute the datasets into multiple compressor or decompressor engines using our hardware scheduler.

A. CPCIe Architecture

In this section we discuss the components used in our CPCIe architecture. We begin with the MicroBlaze processor and then discuss the PCIe controller. Then we describe the Custom Interface used in the architecture, which is responsible for communicating between Xillybus and the CPCIe. Finally, we briefly present the Compressor and Decompressor Systems used in our system.

ISSN: 2454-4248
405 – 419

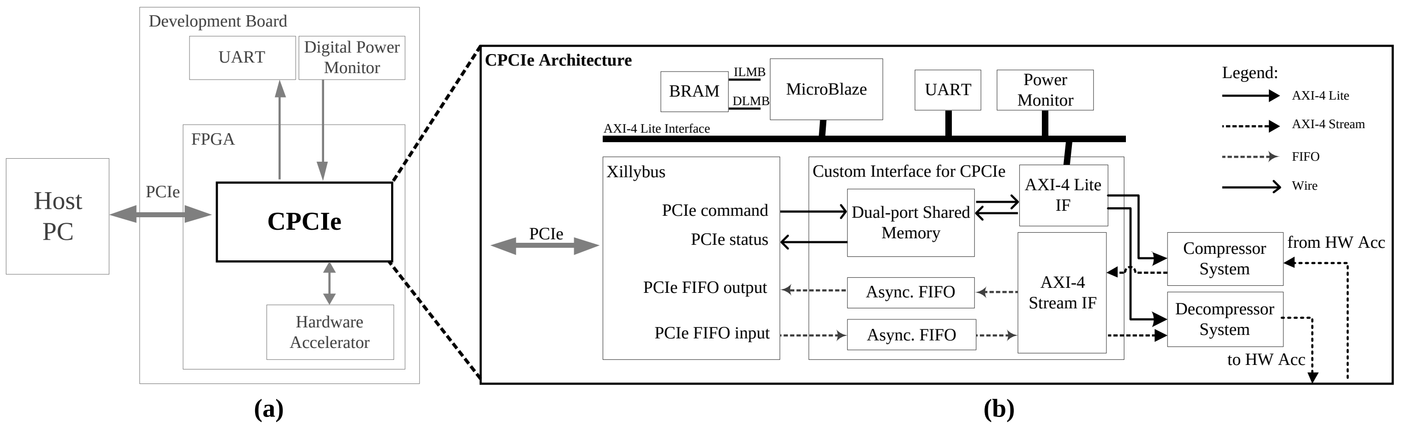

Figure 1: Overview of the full system. In (a), the deployment of CPCIe within the evaluation system. In (b), the architecture of CPCIe

MicroBlaze: This soft processor serves as a controller to display debugging output, to communicate with the host PC and to control the whole system of the CPCIe by, for example, configuring the peripherals via the bus interface. In addition, this processor is used to read the power dissipated from the on-board digital power monitor. The power reading is periodically displayed on the UART terminal. Xillybus: In order to overcome the complexity of developing the controller of the transaction layer of PCIe, several third-party IP cores were proposed and are available in opensource formats such as RIFFA [37] and Xillybus [38]. In our work, Xillybus was chosen because of its stability and the fact that no specific API is involved. Xillybus uses 32-bit data to transmit/receive data to/from the FIFO on the FPGA, and connects the FPGA application logic using a FIFO. Furthermore, the Xillybus periodically checks two signals from the FIFO: the FIFO's full signal, to initiate data transfer, and the FIFO's empty signal, to read data from the FPGA. On the software side of host PC, the Xillybus provides a device driver for the Windows or Linux operating system, where the file handler is opened by the host application before the user application performs the file I/O operation. The I/O files are written or read as binary files between the memory buffer and the FIFOs on the FPGA.

Custom Interface: The capabilities of the Xillybus core are extended by developing an interface to exchange its command, status and FIFO with the CPCIe. In this component, the AXI-4 Lite Interface is used by the MicroBlaze processor to store/read data to/from the Dual-Port Shared Memory (DPSM), and to communicate with the Compressor and Decompressor Systems. The DPSM is an on-chip dual-port block memory and can be accessed by the host PC to store the value of the command and status. The values in the DPSM are the start/stop/wait signals, the configuration/status of the Compressor and Decompressor Systems, and the configuration/status of the hardware accelerator. At the bottom of the DPSM, two asynchronous FIFOs are implemented between the Xillybus' FIFO and the AXI-4 Stream Interface for the input and output FIFOs. These FIFOs are essential to balance the different clocks used in the PCIe and the Compressor/Decompressor System and to ensure data integrity. The AXI-4 Stream interface is used to convert the FIFO signals into standard AXI-4 Stream protocols.

Compressor System: The Compressor System is responsible for compressing the result from the hardware accelerator and sending the compressed output back to the host PC. The MicroBlaze processor sends the command to start the compression, which consists of the value to configure the compressor engines inside this system.

Decompressor System: The compressed datasets is sent to the Decompressor System first before the uncompressed output is sent to the hardware accelerator. The flow in the Decompressor System is not very different from that in the Compressor System, except that this system will use a header data in the compressed file to schedule the compressed blocks during decompression.

In the following subsections, we describe in depth our implementation of the Compressor and Decompressor Systems. Then, we describe the functionality of the hardware scheduler used in both systems. Finally we discuss the compressor and decompressor engines using the X-MatchPRO compression/decompression algorithm used for our evaluation.

B. Compressor/Decompressor System

The Compressor and Decompressor Systems are shown in Fig. 2 and Fig. 3, respectively. The Compressor System comprises the scheduler, the header packer and multiple compressor engines. Note that the number of compressor engines is pre-configured before synthesis. In order to start the compression, the scheduler requires the value of the original file size, and the other two configurations from the host PC, which are the block size and threshold value. The value of the original file size is stored in the header file, while the other two configuration values are required by the compressor engines. Each time the compressor engines have finished compressing, they generate the compressed size and CRC values, which are stored in the FIFO-based header packer. The compressed size is the total number of compressed words (1 word is equivalent to 4 Bytes) and the CRC value is the 32-bit encoding from the original uncompressed block. After all of the compressed data have been sent from the output of the scheduler, the Compressor System will send the header data from the header packer to be stored as part of the compressed binary file in the host PC. Finally, the Compressor System will notify the MicroBlaze processor that it has completed the job via the AXI-4 Lite interface.

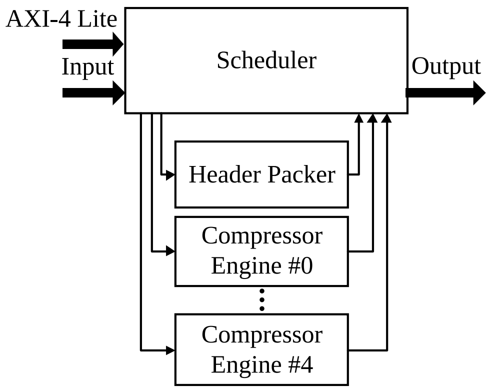

Figure 2: Compressor System in the CPCIe Architecture.

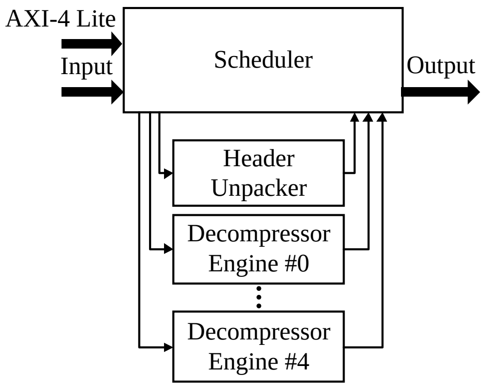

Figure 3: Decompressor System in the CPCIe Architecture

The process flow and the components in the Decompressor System are very similar to the Compressor System, except that the header unpacker is used to replace the header packer. The FIFO-based header unpacker is responsible for temporarily store the header of the compressed binary file, which is unpacked by the scheduler. After all of the data in the header have been stored, the scheduler is ready to start distributing the compressed data to the decompressor engines. The scheduler reads the first compressed size from the header unpacker and stores it in its own counter to count the number of data words. This counter is used to count the maximum number of data elements for the decompressor engine. Simultaneously, the decompressed data from the first decompressor engine is sent to the hardware accelerator via the output FIFO in the scheduler. Once the counter has reached zero, the scheduler updates the counter with the second compressed size and starts to send the second block of compressed data. Upon completion, the scheduler sends a signal to the MicroBlaze processor to close the connection.

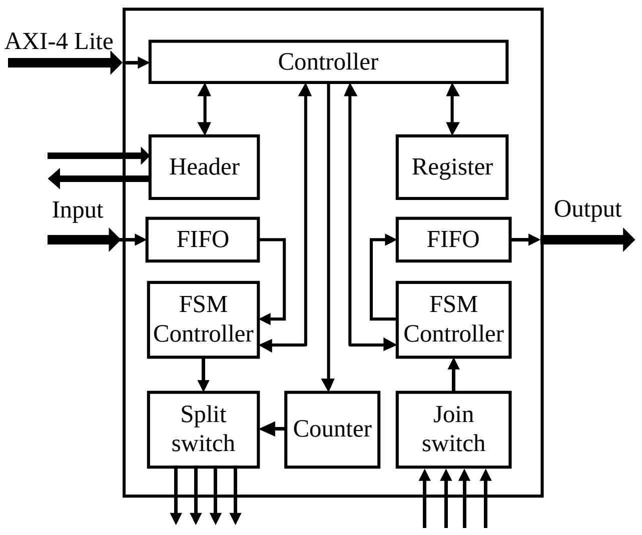

Figure 4: The architecture of scheduler in both Compressor and Decompressor Systems.

C. *Hardware Scheduler*

The architecture of scheduler for both Compressor and Decompressor System is shown in Fig. 4. The compressor and decompressor parallel engines have each their own schedulers to handle multiple blocks of specified block sizes based on the uncompressed (for compression) or compressed (for decompression) datasets. The scheduler is implemented in hardware using a Finite State Machine (FSM) and manages two inputs from the user which are the file size and block size (ranging between 1 KB to 32 KB). Another capabilities of the scheduler is that the FSM controller can be configured to disable the compression mechanism and transmit the raw datasets.

Prior to sending a block to compress or decompress, the scheduler continuously checks if an engine is ready by inspecting the status associated with the engine. By using this scheduling scheme, the scheduler ensures all engines remain busy while there is input data waiting in the buffer and ready to be assigned to the next available engine. If an engine is available and ready to start processing, the scheduler performs the following tasks: locks the engine, configures the split switch, configures the block size and transfers the data from input buffer to the engine. While the data is being transferred to the engine, the scheduler will inspect the next available engine. If the engine is busy, the scheduler keeps monitoring the engine until it is ready to be processed before sending the next block to it. In our design we used loop-checking as the cycles are relatively low in the range between 30 to 50 cycles.

Table 2: Resource utilization of each component used in the CPCIe system.

	Number of Engines	LUTs	Registers	BRAMs
Xillybus PCIe	-	6500 (2%)	6668 (1%)	9 (1%)
Compressor System	1-Engine	7085 (2%)	3532 (1%)	26 (1%)
	2-Engines	12800 (4%)	6105 (1%)	38 (2%)
	3-Engines	18515 (6%)	8678 (1%)	50 (2%)
	4-Engines	23230 (8%)	11251 (2%)	62 (3%)
Decompressor System	1-Engine	3790 (1%)	1882 (1%)	21 (1%)
	2-Engines	6328 (2%)	2827 (1%)	28 (1%)
	3-Engines	8866 (3%)	3772 (1%)	35 (2%)
	4-Engines	11404 (4%)	4717 (1%)	42 (2%)

In compression mode after an engine has finished compressing its input data, the scheduler executes the following tasks: configures the join switch, copies or ”pushes” the compressed size and cyclic redundancy check (CRC) value to the FIFO of the header packer, transfers the compressed data from compressor to the output buffer of the scheduler, and unlocks the engine. If the decompression mode is active and an engine has finished decompressing, the scheduler performs the following tasks: reads or ”pulls” the compressed size and CRC value from the FIFO of the header unpacker, writes these values to the decompressor engine, and transfer the decompressed data from the decompressor to the output buffer of the scheduler. The decompressor engine requires the compressed size value to be written in a register as a counter to count the number of compressed data read during decompression. While the copied CRC value is used after all data has been decompressed and will be compared with the CRC value generated by the decompressor to identify any corrupted data. If the CRC flag outputs an error, the decompressor will repeat the process again.

D. X-MatchPRO Compressor/Decompressor Engine

The X-MatchPRO compressor/decompressor is used in our evaluation platform since it is also available as an open-source core and it offers useful features regarding performance and latency. It belongs to the category of dictionary-based compressors and consists of a compressor and decompressor channel. The compression mode consists of the compressor to read uncompressed input data from the input buffer, compressing it based on the block size used, and generates 32-bit data words to the output buffer. In the decompression mode, the decompressor reads compressed input data from the input buffer, decompress the data based on the dictionary references and reconstructs the original data. The compressor and decompressor use a parallel dictionary of previously seen data to match or partially match the current data element with an entry in the dictionary and it is coded in VHDL. More details are available at [13][14].

E. Software API

We developed four API calls to be used by the CPCIe: compress, decompress, write addr, read addr.

```
compress (input, bsize, trshold, output)
decompress (input, output)
```

The compress and decompress calls represent the compression and decompression function calls from the host PC, respectively. In the case of the compress function call, the input of the uncompressed dataset is specified by the input argument. The block size (bsize) and threshold (trshold) of the function argument are the window size for compression and the threshold for the compressor engine, respectively, and the return value of the compressed dataset is written to an address output and saved as a new compressed file. Meanwhile in the case of the decompress function call, the block size and the threshold are not required in the argument, as these values have already been written into the header of the compressed file during compression.

```
write_addr (u32 addr, u32 value)
read_addr (u32 addr)
```

The write addr call performs the writing of a hex value to an address specified by the argument addr. A developer can modify the address maps to execute any command on either the host PC or soft processor (e.g. MicroBlaze processor). These address maps are used to control and communicate with the host PC, while the soft processor will perform an operation based on the command values stored in the specific address. The addresses which are used to store these values are the start and stop operation for the hardware accelerator and compressor/decompressor systems, the status of the hardware accelerator, the status of the compressor/decompressor engines, and the value of metadata (the original file size, the block size and the number of blocks) for the main header in the compressed file. While the read addr call is used to read a value from an address, especially to read the status of the soft processor. This status is useful not only for debugging purposes but also for monitoring the activities in the FPGA.

IV. DEMONSTRATION SYSTEM

The demonstration board used in our work is the Xilinx Virtex-7 FPGA VC707 evaluation board [39] and synthesized with Xilinx Vivado 2014.4. This board consists of PCIe interface, and on-board digital power monitor to obtain power readings from the FPGA board. For host PC, we used a workstation with Intel Xeon E5520 processor, running on 64-bit Windows 7 operating system.

A. Resource Utilisation

In this subsection we present the resource utilization of the different modules in our CPCIe core: the Xillybus PCIe core, the compressor and the decompressor. In this project, we wanted to make sure that all clocks are synchronous and we could only achieve timing closure at fixed frequency of 100 MHz due to the complexity of using several compression and decompression engines in parallel. Table 2 shows the resource utilization of the compressor and decompressor engine with various number of engines. Each compressor and decompressor is divided into four different rows to represent the utilization using different number of engines.

At the moment once the number of engines has been defined and the design implemented it is not possible to change the number without a full implementation cycle. It will be possible to create configurations off-line with different number of engines and then load the correct configuration at run-time using a partial reconfiguration technique but this has not been explored in the current work.

One point to note is that the compressor engine utilizes almost twice as many logic resources as the decompressor. The reason is that in the original configuration of the compressor, there is a compressor engine and also a decompressor engine implemented in the component for verification purposes. The decompressor engine in the compressor is used to produce CRC value, after all compressed data has completely decompressed. This CRC value will be compared with original CRC value in the compressor engine to check for identical result. Both the compressor and decompressor engines have the same resources of CRC which consists of 32 flip-flop registers and 160 LUTs.

Our proposed CPCIe can be extended to other platform because none of these resources are restricted to a specific technology. Furthermore, the Xillybus also supports other FPGA vendors such as the Altera’s PCIe board. For example, if one requires using Altera board, the source code has to be modified by changing the FIFOs and the BlockRAMs, as these resources are based on primitives on the FPGA.

It is correct to that as more FPGA resources are used by the CPCIe accelerator in the device the place&route (PAR) phase can have more problems finding a good implementation and the timing constraints could not be met. In our case the CPCIe core is not the bottleneck for performance and the timing constraints set for the accelerator can be met for the CPCIe for the accelerator. If this was a problem options available in Vivado could be used to change the PAR strategy. For all experiments, we used the strategy of Performance ExplorePostRoutePhsyOpt, which is provided by the Vivado. This process is repeated until all the timings are met on the FPGA design.

B. *Evaluation of Compression Efficiency*

In this subsection we use the demonstration system to evaluate the effectiveness of the compressor with typical data obtained from high-performance applications. The focus of the paper is to use datasets with content that can be processed by the hardware accelerator and that achieve good compression. In our project, we use a lot of floating point numbers to process in the accelerator before returning the result back to the host PC. Traditional datasets such as Canterbury datasets were not suitable for our purposes because these datasets tend to be based on text or binary files which are not suitable for the hardware accelerators.

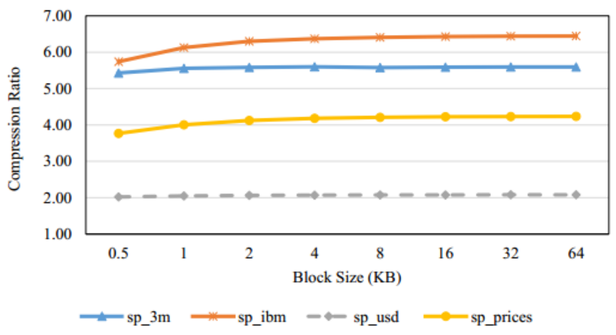

Figure 5: The compression ratio of test datasets. The higher compression ratio is the better.

There are four financial time series datasets acquired from public domain; sp 3m and sp ibm, which represent the stock price of 3M and IBM market, respectively, and the other two are sp usd and sp prices, which represent the exchange rates between US Dollar and Japanese Yen, respectively. All datasets are coded in IEEE 754 single-precision floating-point number representation and saved as a binary file. To measure the compression efficiency, the compression ratio is used as a relative size scale. The compression ratio is defined as:

$$\textit{Compression Ratio} = \frac{\textit{Original Data Size}}{\textit{Compressed Data Size}}$$

where a higher ratio indicates that a higher rate of compression can be achieved in the data. In our evaluation we compressed each dataset with different block sizes using the XMatchPRO compressor. Upon completion of compressing, the compressed output is decompressed to verify correctness.

Fig. 5 analyzes the compression ratio for the different test datasets and compares them with block sizes ranging between 512 Bytes and 64 KB. Compression varies between factors 2X to 6X depending on the contents of the datasets. It is possible to appreciate that 4 KB is a suitable block size to compress and this size is consistent across all evaluated datasets. Compressing the data in small independent blocks of 4 KB has the advantage that bit errors will only corrupt the data of the block in which they are located, thus reduces the overall latency and opens the possibility of working in independent blocks in parallel. These bit errors can happen during transmission between computers. For example, sending raw data from one machine (without an FPGA-based compressor engine) to another machine (with an FPGA-based compressor engine) over the Ethernet might incur the possibility of a bit error during transmission. The CPCIe has CRC checking capabilities to check any bit errors during transmission, and it will request that the specified block to resend again if a bit error is been detected. The nature of dictionary-based adaptive compression/decompression algorithms means that in unblocked mode only a single engine could be used since the dictionary is built as new data is seen and its state must be synchronized during the compression and decompression processes. This 4 KB block size will be used for the energy and performance experiments conducted in section VI and also in the following sections.

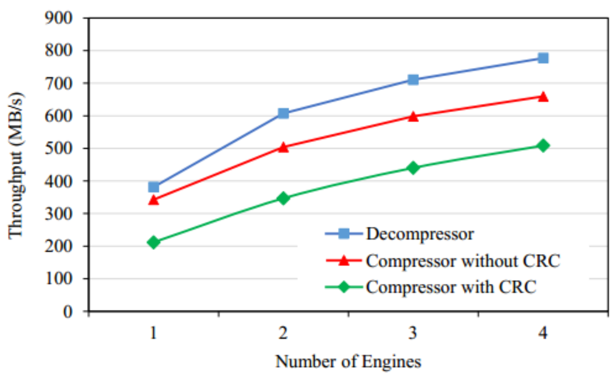

Figure 6: Throughput of compressor/decompressor in various number of engines. The filesize used is 4 MB, and compressed/decompressed with 4 KB of blocksize.

C. *Latency and Throughput*

To measure the latency in CPCIe, we used a counter to count the total number of clock cycles. The counting of the clock is measured from the endpoint of PCIe to the endpoint of output FIFO in the scheduler (just before the accelerator). The first measurement was taken to obtain two measurements of the latency which occurred in the X-MatchPRO engine and in the scheduler during distributing the datasets. The compressor/decompressor engine itself has a latency of only nine clock cycles before it is ready to produce output data compressed and decompressed. The other measurement is the latency between the host PC and the CPCIe, which is between 11 and 18 µs. This latency consists of several paths from the host PC, Xillybus, and PCIe, before reaching the engines. It is important to note that this latency is bounded by the limitations of PCIe communication which in this work consists of the Xillybus, the PCIe connections and the PCIe socket on the motherboard.

To measure the throughput, we implemented a wall-clock timer in the host PC and used the following calculation:

$$Throughput = \frac{filesize}{time}$$

This calculation is based on a user-selectable mode in the host PC: if the compression mode is used, the filesize in the calculation will be the uncompressed file size (or the compressed file size if decompression mode is selected), while the time is the total period to compress the whole file (or the total period to decompress the whole file if decompression mode is selected).

It is important to note that the frequency of the compressor and decompressor engine is fixed at 100 MHz, while the frequency in PCIe remains the same at 250 MHz. The PCIe theoretical throughput is 800 MB/s and the FIFO in CPCIe should not become the bottleneck although the CPCIe runs at 100 MHz. This is achieved using an asynchronous FIFO in each compressor/decompressor engine. Fig. 6 shows the throughput for compression and decompression, with various number of engines implemented in the CPCIe. In

Table 3: Power measurements of three experiments: when transmitting a data, during compression and during decompression.

Power Experiments	Component	Idle Power	Active Power
Exp. 1: Data Transmission (PC+FPGA)	Host PC	101.52 W	132.41 W
	Xillybus PCIe	2.473 W	2.625 W
Exp. 2: Compression (FPGA)	1 Compr.	0.027 W	0.032 W
	2 Compr.	0.054 W	0.065 W
	3 Compr.	0.081 W	0.097 W
	4 Compr.	0.108 W	0.131 W
Exp. 3: Decompression (FPGA)	1 Decompr.	0.019 W	0.023 W
	2 Decompr.	0.038 W	0.046 W
	3 Decompr.	0.057 W	0.071 W
	4 Decompr.	0.072 W	0.093 W

decompression mode, the throughput is around 776 MB/s with four engines, and the throughput decreases as the engines reduced. In compression mode, the throughput of four engines without CRC is around 659 MB/s. However, the compressor engine with CRC could not achieve the throughput close to 400 MB/s (in one engine for example) as it has to wait for its own decompressor to finish to generate the identical CRC result.

D. *Power Consumption*

In general, there are two types of power we need to consider in a circuit: dynamic power and static power. Dynamic power is the power when the clock is running and gates are switching in the circuit. While static power is the leakage power in the circuit with clocks switch off. When the system is not performing any operations but the clocks are running, the system is said to be in idle state. The idle state has its own power called idle power, that contains both dynamic and static power components. The static power tends to be significant, while the dynamic power is still present in the idle state since the clock network is active and some circuits such as communication circuits are maintained alive. The system is said to be in active state if application operations and events occur. The active state has its own power called active power due to the processing functions that take place. During active state, both dynamic and static power are dissipated and are included in the active power.

Before taking any measurements, this work assumes that the datasets have been compressed off-line and stored in the PC host side in the form of archives or databases that can be processed when needed. In essence this will be done only once, while the same datasets are used with different accelerators multiple times. Notice that if the datasets are uncompressed in the PC side and must be compressed by software when needed, the whole approach is impractical since the software will be much slower than the FPGA board. This idea is similar to how images and videos are maintained in a compressed state in the database and transmitted in compressed format, and only decompressed before processing is needed. In our case our datasets are not images or videos but files consisting of floatingpoint numbers.

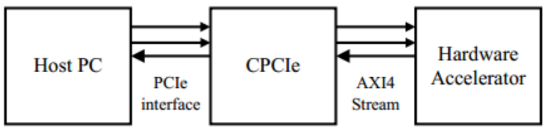

Figure 7: The implementation of CPCIe between Host PC and accelerator. There are two inputs and one output from the accelerator, connected using an AXI4-Stream interface.

In order to calculate the power consumed by the whole system, both the host PC and the FPGA board are considered in the measurements. The on-board FPGA power consumption alone is not sufficient since we are trying to evaluate the effects on the whole system. The measurement for the whole system is required which includes the reading and writing dataset from the host PC to the FPGA (and vice versa). Since the dataset is transmitted from the memory buffer in the host PC to the FPGA, the affected energies are the main memory in the host PC, the PCIe socket and the FPGA itself. Note that small size data will achieve less energy compared to huge data during this transmission. This energy must be taken into account for the total energy consumption.

To measure and isolate the power on the FPGA, we used the on-board digital power monitor available on the Xilinx Virtex- 7 VC707 development board. The power monitor has a Texas Instruments UCD9248 chip controller [40] and has the capabilities to obtain voltage, current and power readings from 12 voltage rails on the FPGA board. The power monitor chip wires the connection between voltage rails and the PMBus (powermanagement bus) [41] to obtain the power reading. We used the MicroBlaze processor to obtain the output power from the PMBus using our custom software. Then the output power is written to the UART to display the power from the selected rails. To evaluate the power usage, only a few selected rails on the board are measured which are referred as VCCINT, MGTAVCC, and MGTAVTT; other unnecessary power rails are not measured as they are wired to the unrelated peripherals which were not involved in the experiments (such as HDMI ports, USB, Ethernet and others). The VCCINT rail provides internal power supply to FPGA resources such as registers, LUTs, and DSPs. While the MGTAVCC and MGTAVTT voltage rails are the analog supply to GTX transceivers in the PCIe endpoints, which are used for internal analog circuits and termination circuits, respectively. These are the power in the PCIe which are wired with eight GTX transceivers located in transceiver bank of the Multi Gigabit Transceiver MGT BANK 114 and MGT BANK 115 on the FPGA chip.

To obtain the power measurement on the PC side, we used a power meter [42] located between the power wall socket and the host PC. Our measurements try to evaluate the real power and energy required by the system and the possible costs savings of using compression and decompression. The power wall measurements are real power since they represent energy that the user needs to pay for the bills, even if some of that energy does not make into the system and it is wasted. For that reason we do not perform adjustments based on efficiency and use the raw measurements from the power sensors in our calculations. It is worth noting that the power consumption to transmit a datasets requires several components to be involved, which include the power from the motherboard, the CPU, memories, and the FPGA board.

We conducted three power experiments to get the measurement of idle and active power that correspond to the: (1) transmission of a dataset between the host PC and FPGA, (2) compression, and (3) decompression. The objective of these experiments is to obtain the idle and active power in three components which are the host PC, the Xillybus PCIe and the FPGA. In each experiment, the measurement was taken during two states: (1) while the components were in idle state, and (2) while the components were in active state when the transmission takes place between the host PC and FPGA. By measuring these power from the host PC and FPGA side, we can obtain an estimation of the total energy of the overall system using both the uncompressed and compressed datasets. We can then estimate the energy savings obtained by using the proposed compression/decompression system, compared with not doing so.

Table 3 shows the power experiments during idle and active states. In experiment (1), the idle power and active power in the host PC are 101.52 W and 132.41 W, respectively. These measurements were obtained from the power meter only. The idle and active power on the PC side alone are the same during compression and decompression, regardless of the number of engines. The only difference is the time taken for the PCIe to transmit the data, which contributes to the total energy consumed for the different configurations. While the idle power and active power in the Xillybus PCIe are 2.473 W and 2.625 W, respectively. These measurements were obtained from the VCCINT, MGTAVCC and MGTAVTT rails of the on-board power monitor. In experiment (2) and (3), the idle and active power in the compressor/decompressor engines were obtained from the VCCINT rail only. Based on the number of engines used, one can observe that the power in the compression were between 27 mW to 131 mW. While the power in the decompression were slightly lower than compression which results between 19 mW to 93 mW.

V. Application Selection and Analysis

We have selected two applications to represent benchmarks typically used in high performance computing: the buffer applications and streaming applications. The matrix multiplication is an example of buffer application, and the hotspot convolution is an example of streaming application. In matrix multiplication, two matrices are written into two input buffers, multiplication is performed, and the final results are obtained from the output buffer. In our specific case, we consider that we need to multiply 4 tiles of 256x256 matrices which give us a total of 1024x1024 matrices. Typically the buffers are processed locally in the block memory. There are a lot of data reused as rows and matrix are multiplied and added together. It is important to note that the input and output buffers are synthesized as a Block RAMs by Vivado HLS. The use of Block RAMs in the FPGA dramatically reduces the amount of available Block RAMs for the user application and limits the size of matrices that can be multiplied. While the hotspot application is used to observe how temperature propagates on the chip surface and to avoid extreme thermal effects. Hotspot typically uses 2D convolution and a 3x3 sliding window size (or kernel) that is applied to the input frames that represent the chip surface.

Table 4: The utilization of each resource on FPGA used by the accelerators.

	LUTs	FFs	BRAMs	DSP48E
Matrix Mult.	108881	110941	640	1280
Hotspot	3088	3443	10	22
Available Resources	303600	607200	2080	2800

Table 5: Power consumption of each application benchmarks.

	Idle Power	Active Power
Matrix Multiplication	0.103 W	0.145 W
Hotspot	0.081 W	0.103 W

The input datasets are chosen to be representative of real problems. In our work, we consider floating-point numbers instead of integer numbers, as floating-numbers are widely used in scientific computing. The datasets used are two input matrices which comprise 1024x1024 floating-point numbers. These datasets have two versions for our evaluation: an uncompressed and pre-compressed datasets. The pre-compressed datasets are compressed beforehand in the host PC. All the source code for these applications is written using Vivado High Level Synthesis 2014.4 with an AXI4-Stream interface and used IEEE 754 single-precision. For the purposes of this experiment these accelerators are constrained to 100 MHz to achieve synchronous clock in the whole FPGA design.

Table 4 the Xilinx Virtex-7 FPGA. Clearly matrix multiplication is the most complex compared to hotspot. Table 5 shows the power consumption of test applications for the benchmark. The idle and active power in matrix multiplication is around 0.103 W and 0.145 W, respectively. While in hotspot, the idle and active power is around 0.081 W and 0.103 W, respectively. Fig. 7 shows an overview of the implementation of CPCIe to be used with these applications.

VI. PERFORMANCE AND ENERGY ANALYSIS

In this section, we compare three systems with and without deploying the compression technology to understand the impact of the compressor and decompressor in terms of transmission speed and energy. The designed system sends two input datasets with matrix size of 1024x1024 each and obtains and output with the same matrix size.

A. *Test case without compression*

In this test system, the task is to send an *uncompressed* dataset from host PC to the input of accelerator, and the output of the accelerator are then sent back to the host PC as an *uncompressed* result after processing. Fig. 8 illustrates the execution phase in the PCIe and the accelerator. As soon the accelerator receives the input, it starts to process the dataset. It is clear that the active time is the most time consuming in the PCIe transmission, in both sending, T_{pas} (the active period during host PC sending the data over PCIe), and receiving dataset,T_{par} (the active period during host PC receiving the data from PCIe). We measure the performance and energy of this test case to be used as a comparison reference with the test cases that use compression.

B. *Test case with compression*

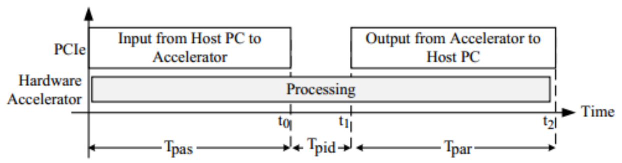
PCIe
Input from Host PC to Accelerator
Output from Accelerator to Host PC
Hardware Accelerator
Processing
Time
t0
t1
t2
Tpas
Tpid
Tpar

Figure 8: The execution phase of sending an uncompressed input and reading uncompressed output.

In this second task, the objective is to send a compressed dataset from host PC to the accelerator, and the result from accelerator is sent back to the host PC as a compressed result. We deployed our CPCIe core in between the host PC and the accelerator, which connected with an AXI4-Stream interface. The CPCIe will decompress the compressed dataset first, before supplying the decompressed data into the input of the accelerator. The result of the accelerator is connected to the compressor, and will be compressed by the CPCIe before returning the result to the host PC.

Fig. 9 shows the execution phases of CPCIe using a single compressor/decompressor engine, while Fig. 10 shows the execution phases using four engines. The execution phases with two and three engines are similar to the four engines case and not shown in this work. The time taken to read the header is denoted as Txdh. While the host PC is transmitting the compressed dataset, the scheduler stores its header file into a temporary buffer before decompressing the blocks. We refer this as a timing overhead and it is measured between the endpoint of the PCIe interface and the FIFO in CPCIe. The overhead consists of time to control the signals in the FSM controller and transfer delays in PCIe. The header must be fully loaded into the temporary buffer in the scheduler during PCIe transmission, before the compressed data can be used in the decompressor. The total time of the header to be fully loaded is determined by the number of blocks. The time to write the header, denoted as Txch, in the configuration with four engines took almost the same amount of time with one compressor/decompressor engine.

In a single-engine configuration, the scheduler depends on the availability of the compressor/decompressor engine before transferring the input data for compression/decompression. That is, if one block of data is still processing in an engine (either in compressor or decompressor engine), this will induce a waiting time in the scheduler until the engine has finished. While in the multiple-engine configuration, the compressor and decompressor engines work in parallel. This can be seen after Txdh in Fig. 10 where all decompressors are running independently at the same speed, and the scheduler does not have to wait for a decompressor to finish and can keep the performance by continuously distributing the data among the decompressor engines.

We also repeated the same experiments with other two datasets which have a significantly higher compression ratio of 4X and 6X and the results in terms of performance and energy are shown in the next section.

C. *Performance and Energy Evaluation*

For each transmitted Byte, we are interested in exploring the energy requirements. We did this by transmitting datasets for different file sizes (ranging from 1 KB to 4 MB). By

Table 6: Energy consumption and energy savings in Matrix Multiplication using CPCIe

		PCIe Energy	Energy Savings
Uncompressed		453.11 J	-
Compression Ratio 1.5X	1-E	430.91 J	4.90 %
	2-E	428.75 J	5.38 %
	3-E	424.98 J	6.21 %
	4-E	423.10 J	6.62 %
Compression Ratio 4X	1-E	399.79 J	11.77 %
	2-E	398.08 J	12.14 %
	3-E	396.34 J	12.53 %
	4-E	393.04 J	13.26 %
Compression Ratio 6X	1-E	385.14 J	15.00 %
	2-E	382.57 J	15.57 %
	3-E	378.19 J	16.53 %
	4-E	375.94 J	13.26 %

Table 7: Energy consumption and energy savings in Hotspot using CPCIe

		PCIe Energy	Energy Savings
Uncompressed		81.07 J	
Compression Ratio 1.5X	1-E	65.46 J	19.25 %
	2-E	65.02 J	19.80 %
	3-E	64.88 J	19.97 %
	4-E	63.82 J	21.28 %
Compression Ratio 4X	1-E	50.27 J	38.00 %
	2-E	48.80 J	39.80 %
	3-E	47.38 J	41.56 %
	4-E	45.86 J	43.43 %
Compression Ratio 6X	1-E	45.89 J	43.39 %
	2-E	44.56 J	45.04 %
	3-E	43.26 J	46.64 %
	4-E	41.87 J	48.35 %

dividing the energy required by the file size, we could obtain the tradeoff nJ/Byte:

$$\textit{Trade-off, nJ/Byte} = \frac{\textit{Energy (nJ)}}{\textit{Filesize (Byte)}}$$

Fig. 11 investigates the tradeoff between energy and block size in unit of nanoJoules/Bytes. The block size of 512 Bytes has the highest Energy/Bytes (14.2 nJ/B), followed by block size of 1 KB (13.8 nJ/B), and 2 KB (13.6 nJ/B). However, for more than 4 KB of block size, this trade-off is around 13.5 nJ/B. From this graph, we can conclude that for optimal energy per byte over PCIe, we need to use a basic block transfer size of at least 4 KB. The reason behind this situation is that the block size of less than 4 KB has a high amount of overhead (header, protocol, status, etc.) over the payload size. Furthermore, 4 KB is also a good block size from a pure compression ratio point of view as seen in Fig. 5. For these reasons, in this evaluation, all tests used the block size of 4 KB during compression and decompression.

Table 6 and Table 7 show the result of the energy consumption of the matrix multiplication and hotspot, respectively. Both tables show the energy consumed during the processing of the datasets in the hardware accelerator, which includes transmitting, processing, and sending back the results of the datasets. The row called ”Uncompressed” is the result for the energy consumed while processing the raw datasets in the hardware accelerator. The following rows are the compression ratio of a data, with other configurations varying the number of compressor and decompressor engines working in parallel (in addition to the single engine). In order to obtain the total energy involved performing the process, the energy of both the host PC (such as CPU, memory, etc.) and the FPGA should be considered in the calculation. In these tables, the total energy consumed while processing the uncompressed datasets for the matrix multiplication and hotspot application was calculated around 453 J and 81 J, respectively.

In the next row, it can be seen that the total energy has reduced due to the presence of the compressor/decompressor engine. Note that the energy reductions are dependent on the compressed data size and also the number of engines used. For example, considering a compression ratio of 1.5X using a single engine (which consists of one compressor and one decompressor engine) in the matrix multiplication, the energy reports around 430 J. Thus, the energy in the PCIe of this configuration decreases by 4.90% compared with the energy using the uncompressed datasets. The next row presents the configuration using two engines, and this reports the energy consumed to be around 428 J. Although the number of engines was doubled, the energy used is only slightly lower compared

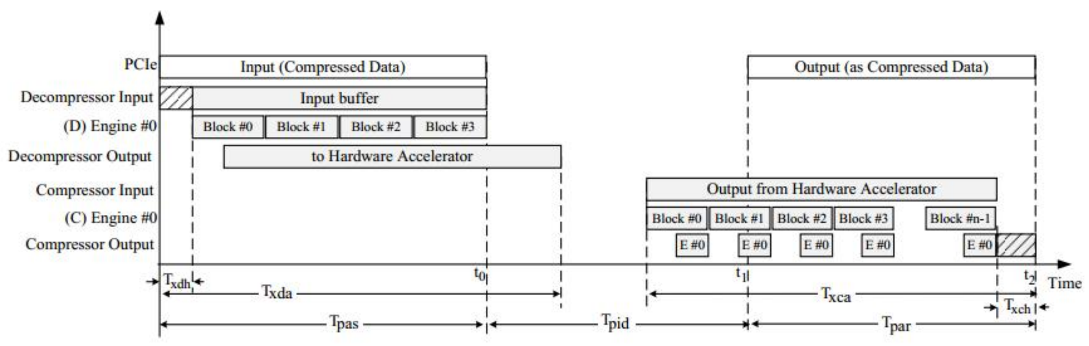

Figure 9: The execution phase of sending a compressed input and decompress the dataset with a single decompressor engine before sending the decompressed dataset to the hardware accelerator. The result is then compressed with a single compressor engine before it is sent over PCIe. In Compressor Output, the E #0 is the compressed output from a single engine.

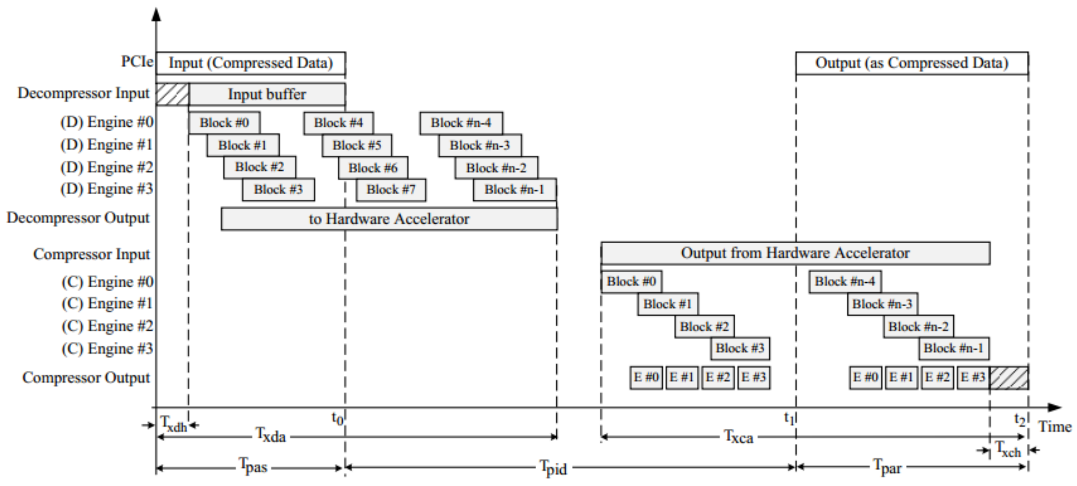

Figure 10: The execution phase of sending a compressed input and decompress the dataset with multiple decompressor engines before sending the decompressed dataset to the hardware accelerator. The result is then compressed with multiple compressor engines before it is sent over PCIe. In Compressor Output, the E #*n* is the engine's number to generate the compressed output from the particular engine.

to a single engine. The reason is that although the additional compressor and decompressor engines on the FPGA increase the power, the energy reduces due to the reduction in time. In Table 7, we found that the hotspot application obtains the highest energy savings around 48%. This was acquired with datasets that achieve a compression ratio of 6X and deploying on four compressor/decompressor engines in parallel. In conclusion, significant energy savings can be achieved with two conditions: the CPCIe uses a high number of compressor/decompressor engines working in parallel and the compression ratio of a dataset is high (more than 6X of compression ratio).

Fig. 12 summarizes the result for the matrix multiplication, while Fig. 13 for the hotspot application. The horizontal axis on each graph is the number of engines (the orange bar represents the implementation without CPCIe), the vertical axis on the left is the total energy for completing a task, and the vertical axis on the right is the performance of execution time. In each number of engine, we repeated the experiments with three datasets of different compression ratio of 1.5X, 4X and 6X.

As the graphs illustrates, the energy obtained by implementing

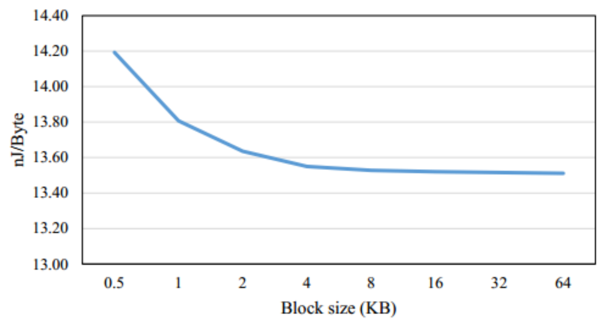

Figure 11: The energy-blocksize tradeoff for transmitting a compressed dataset and measured in nJ/Byte, sorted by block size.

a single engine is significant lower than those obtained without compression. The hotspot achieved slightly higher reduction of energy since this accelerator does not use a lot of logic resources, while the matrix multiplication has to fill up all the input matrices in the buffers before it can process the result. It can be seen that in both graphs, the execution time to complete the acceleration task using a single engine increases around 3%. This performance degradation is due to the overhead during compression and decompression. However the graphs also show that these overhead reduces as the number of engines increases. Nevertheless, one can observe that the energy consumed for the task to complete has decreased since the amount of time during active state in PCIe becomes shorter.

In the configuration with four engines we found that the performance has reached its peak value. At this point, the performance of the whole system entirely depends on the accelerator and its frequency to achieve higher performance. For both applications, the implementation of four compressor/decompressor engines is the number needed to be configured to maintain the same performance as the case without compression.

For the other compression ratios (by taken the configuration with one engine for example), the compression ratio of 4X results in a 11% energy reduction for the matrix multiplication, and a 38% energy reduction for the hotspot application. The compression ratio of 6X on the other hand, has only a slightly energydecreasecomparedwith4Xratio, whichresultsina15% and 43% in both application, respectively. However, the execution time of each compression ratio is not significantly affected as the size and number of blocks needed to be compressed/decompressed remains the same. We also tested up to eight compression/decompression engines, but there were unique cases for each implementation.

In decompression mode, more than four decompressor engines will result in close to 800 MB/s. The throughput cannot exceed

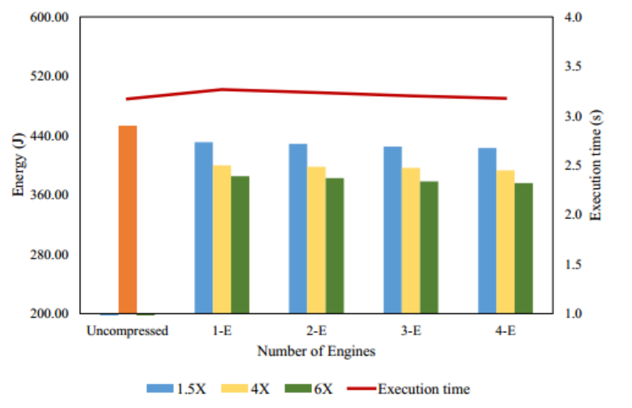

Figure 12: Energy and performance for matrix multiplication: without CPIe (in orange) and with various number of compressor/decompressor engines. Each column has three different compression ratio.

more than 800 MB/s due to the PCIe limitation (the board used was the PCIe Gen2, which is limited to up to eight lanes). Although CPCIe can be configured with between five and eight decompressor engines, some of the decompressor engines will idle most of the time since the process in the

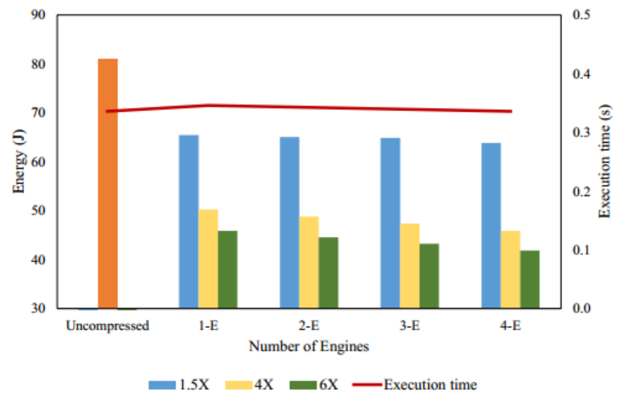

Figure 13: Energy and performance for hotspot application: without CPIe (in orange) and with various number of compressor/decompressor engines. Each column has three different compression ratio.

previous engine(s) finishes earlier before the current engine can take any data. This situation will create another drawback on energy consumption where the additional power on the idle decompressor engines will contribute to the total energy. If these are plotted on Figures 12 and 13, the execution time will still be the same, but the total energy will increase based on the increment of the number of engines. In compression mode, adding more engines increases the critical path in the CPCIe. In our experiments, we found that the timing of four compressor engines is to meet the timing constraint of the accelerator, but the critical path of more than four compressor engines increases with each additional engine becoming a bottleneck.

VII. Discussion

Alternative compression scheme: The compression algorithm X-matchPRO is designed for fast hardware implementation so it will not be very efficient in a software implementation and for that reason a CPU will not be able to obtain the required performance. The system is designed for networking applications in which data is compressed in hardware in one node and it arrives to another node in which is decompressed and processed also in hardware. It will be possible to replace the compression scheme with another compression algorithm and obtain new hardware. For example, the most popular compression scheme is the GZIP compression, which is widely used in the computing field, especially in the following two areas: networking (on websites such as Google and Yahoo for sending/receiving data over the Ethernet), and data storage (in cloud storage to store the compressed format rather than the original format). Since GZIP is open source, the user can easily use the software code, which can be extracted from the public repositories. However, to deploy such software into the hardware (FPGA) will require a new hardware development cycle. It will be also difficult to achieve the real-time performance that X-matchPRO achieves because GZIP is inherently a serial algorithm. To acquire rapid development in this research, an open-source compression/decompression hardware on FPGA was used. We selected XMatchPRO, as it is licenced under the LGPL licence and can be ported on any FPGA vendor. Although the X-MatchPRO algorithm is not similar to other algorithms such as GZIP or other LZ-family, the X-MatchPRO algorithm is able to make use of the FPGA resources and thanks to its parallelism is very fast.

Other compression method (lossy compression): While it is true that this work can be expanded into lossy compression, we have limited the scope of this work to using lossless compression. The reason is that, in our research, we are working with sensitive data that should be preserved. Slight changes in the original data floating-point data could lead to different results in these applications. Several floating-point datasets, such as the stock price market, have been obtained from public repositories. Although these data contain millions of floating-point values, the compression ratio of each dataset is different and is between a ratio of 2.0 and 6.5. However, if the datasets are compressed into the lossy format (and used the quantization) then the achieved compressed ratios will much larger. Based on our results in Figures 12 and 13, we could predict that the compression ratios of more than 7.0 will use less energy. Although the lossy compression can benefit from our proposed system, the drawback of the lossy method is that some data will be lost after quantisation. This could lead to incorrect decisions in a stock market application.

Alternative accelerators: Our focus on this work is to show that by using hardware resources, the overall performance and energy characteristics of PCIe can be improved by using hardware resources to create a hardware core that performs compression/decompression. Thus, this work was not really to demonstrate the advantage of FPGAs against competing solutions. Future work will consider the overall performance of the acceleration solution with compression enabled against other HPC platforms (e.g. Xeon Phi, GPU) and measure the possible advantages of FPGAs in this case.

VIII. Conclusion

In this paper, we have proposed the CPCIe (Compressionenabled PCIe) framework that employs the X-MatchPRO compressor/decompressor engines. We have demonstrated that data compression implemented between the

PCIe core and the accelerator can result in energy savings while maintaining the performance for custom hardware accelerator designs. Our evaluation shows that our CPCIe design can reduce the PCIe energy from 5% to 48%, depending on the number of compressor/decompressor engines working in parallel and the achieved compression ratio. Furthermore, the latency is kept to be minimum in the hardware scheduler needed to complete the scheduling task. The configuration with four engines provides the highest performance and has the lowest overheads, while the compression ratio of 6X provides the highest energy reduction during the PCIe transmission. Based on our experiments, it can be concluded that energy efficiency during PCIe transmission can be improved thanks to the reduction in the transmission period obtain with parallel compression/decompression. This improvement is proportional to the obtain compression ratios and the number of engines working in parallel.

ACKNOWLEDGMENT

This work is supported by the UK EPSRC under grants ENPOWER (EP/L00321X/1) and ENEAC (EP/N002539/1) grants.

REFERENCES

[1] Z. Fan, F. Qiu, A. Kaufman, S. Yoakum-Stover, Gpu cluster for high performance computing, in: Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference, IEEE, 2004, pp. 47–47.

[2] O. Kayıran, A. Jog, M. T. Kandemir, C. R. Das, Neither more nor less: optimizing thread-level parallelism for gpgpus, in: Proceedings of the 22nd international conference on Parallel architectures and compilation techniques, IEEE Press, 2013, pp. 157–166.

[3] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, J. Cong, Optimizing fpgabased accelerator design for deep convolutional neural networks, in: Proceedings of the 2015 ACM/SIGDA International Symposium on FieldProgrammable Gate Arrays, ACM, 2015, pp. 161–170.

[4] A. Iordache, G. Pierre, P. Sanders, J. G. d. F. Coutinho, M. Stillwell, High performance in the cloud with fpga groups, in: Utility and Cloud Computing (UCC), 2016 IEEE/ACM 9th International Conference on, IEEE, 2016, pp. 1–10.

[5] K. Nagasu, K. Sano, F. Kono, N. Nakasato, Fpga-based tsunami simulation: Performance comparison with gpus, and roofline model for scalability analysis, Journal of Parallel and Distributed Computing 106 (2017) 153–169.

[6] H. Giefers, R. Polig, C. Hagleitner, Accelerating arithmetic kernels with coherent attached fpga coprocessors, in: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, EDA Consortium, 2015, pp. 1072–1077.

[7] Cpcie, https://github.com/mohdazainol/cpcie/, [maintained by Mohd A. Zainol].

[8] M. A. Zainol, J. L. Nunez-Yanez, Cpcie: A compression-enabled pcie core for energy and performance optimization, in: Nordic Circuits and Systems Conference (NORCAS), 2016 IEEE, IEEE, 2016, pp. 1–6.

[9] G. Inggs, S. Fleming, D. Thomas, W. Luk, Is high level synthesis ready for business? a computational finance case study, in: Field-Programmable Technology (FPT), 2014 International Conference on, IEEE, 2014, pp. 12–19.

[10] O. Lindtjorn, R. Clapp, O. Pell, H. Fu, M. Flynn, O. Mencer, Beyond traditional microprocessors for geoscience high-performance computing applications, Ieee Micro 31 (2) (2011) 41–49.

[11] C. de Schryver, P. Torruella, N. Wehn, A multi-level monte carlo fpga accelerator for option pricing in the heston model, in: Proceedings of the ConferenceonDesign, AutomationandTestinEurope, EDAConsortium, 2013, pp. 248–253.

[12] M. Araya-Polo, J. Cabezas, M. Hanzich, M. Pericas, F. Rubio, I. Gelado, M. Shafiq, E. Morancho, N. Navarro, E. Ayguade, et al., Assessing accelerator-based hpc reverse time migration, IEEE Transactions on Parallel and Distributed Systems 22 (1) (2011) 147–162.

[13] J. L. Nunez-Yanez, S. Jones, Gbit/s lossless data compression hardware, IEEE Transactions on very large scale integration (VLSI) systems 11 (3) (2003) 499–510.

[14] J. Nunez-Yanez, V. Chouliaras, Gigabyte per second streaming lossless data compression hardware based on a configurable variable-geometry cam dictionary, IEE Proceedings-Computers and Digital Techniques 153 (1) (2006) 47–58.

[15] M.-B. Lin, J.-F. Lee, G. E. Jan, A lossless data compression and decompression algorithm and its hardware architecture, IEEE TRANSACTIONS on very large scale integration (vlsi) systems 14 (9) (2006) 925 – 936.

[16] K. Papadopoulos, I. Papaefstathiou, Titan-r: A reconfigurable hardware implementation of a high-speed compressor, in: Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on, IEEE, 2008, pp. 216–225.

[17] D. C. Zaretsky, G. Mittal, P. Banerjee, Streaming implementation of the zlib decoder algorithm on an fpga, in: Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on, IEEE, 2009, pp. 2329–2332.

[18] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, K. Sheng, Fpga implementation of gzip compression and decompression for idc services, in: Field-Programmable Technology (FPT), 2010 International Conference on, IEEE, 2010, pp. 265–268.

[19] A. G. Villafranca, S. Mignot, J. Portell, E. Garc´ıa-Berro, Hardware implementation of the fapec lossless data compressor for space, in: Adaptive Hardware and Systems (AHS), 2010 NASA/ESA Conference on, IEEE, 2010, pp. 164–170.

[20] R. Naqvi, R. Riaz, F. Siddiqui, Optimized rtl design and implementation of lzw algorithm for high bandwidth applications, Electrical Review 4 (2011) 279–285.

[21] B. Sukhwani, B. Abali, B. Brezzo, S. Asaad, High-throughput, lossless data compresion on fpgas, in: Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on, IEEE, 2011, pp. 113–116.

[22] I. Shcherbakov, C. Weis, N. Wehn, A high-performance fpga-based implementation of the lzss compression algorithm, in: Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International, IEEE, 2012, pp. 449–453.

[23] I. Shcherbakov, N. Wehn, A parallel adaptive range coding compressor: Algorithm, fpga prototype, evaluation, in: Data Compression Conference (DCC), 2012, IEEE, 2012, pp. 119–128.

[24] D. Hogawa, S.-i. Ishida, H. Nishi, Hardware parallel decoder of compressed http traffic on service-oriented router, in: Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2013, p. 1.

[25] B. Li, L. Zhang, Z. Shang, Q. Dong, Implementation of lzma compression algorithm on fpga, Electronics Letters 50 (21) (2014) 1522–1524.

[26] M. S. Abdelfattah, A. Hagiescu, D. Singh, Gzip on a chip: High performance lossless data compression on fpgas using opencl, in: Proceedings of the International Workshop on OpenCL 2013 & 2014, ACM, 2014, p. 4.

[27] Y. Li, Y. Sun, G. Dai, Y. Wang, J. Ni, Y. Wang, G. Li, H. Yang, A self-aware data compression system on fpga in hadoop, in: Field Programmable Technology (FPT), 2015 International Conference on, IEEE, 2015, pp. 196–199.

[28] S. M. Najmabadi, Z. Wang, Y. Baroud, S. Simon, High throughput hardware architectures for asymmetric numeral systems entropy coding, in: Image and Signal Processing and

Analysis (ISPA), 2015 9th International Symposium on, IEEE, 2015, pp. 256–259.

[29] M. Wijeyasinghe, D. Thomas, Combining hardware and software codecs to enhance data channels in fpga streaming systems, Microprocessors and Microsystems 51 (2017) 275–288.

[30] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, P. M. Bland, Ibm memory expansion technology (mxt), IBM Journal of Research and Development 45 (2) (2001) 271–285.

[31] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al., A reconfigurable fabric for accelerating large-scale datacenter services, in: Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on, IEEE, 2014, pp. 13–24.

[32] J. Y. Kim, S. Hauck, D. Burger, A scalable multi-engine xpress9 compressor with asynchronous data transfer, in: Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on, IEEE, 2014, pp. 161–164.

[33] J. Fowers, J.-Y. Kim, D. Burger, S. Hauck, A scalable high-bandwidth architecture for lossless compression on fpgas, in: Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on, IEEE, 2015, pp. 52–59.

[34] J. A. Delmerico, N. A. Byrnes, A. E. Bruno, M. D. Jones, S. M. Gallo, V. Chaudhary, Comparing the performance of clusters, hadoop, and active disks on microarray correlation computations, in: High Performance Computing (HiPC), 2009 International Conference on, IEEE, 2009, pp. 378–387.

[35] G. S. Davidson, J. R. Cowie, S. C. Helmreich, R. A. Zacharski, K. W. Boyack, Data-centric computing with the netezza architecture., Tech. rep., Sandia National Laboratories (2006).

[36] P. Francisco, The netezza data appliance architecture: A platform for high performance data warehousing and analytics, IBM Redguide.

[37] M. Jacobsen, D. Richmond, M. Hogains, R. Kastner, Riffa 2.1: A reusable integration framework for fpga accelerators, ACM Transactions on Reconfigurable Technology and Systems (TRETS) 8 (4) (2015) 22.

[38] Xillybus, www.xillybus.com/.

[39] Xilinx, https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html, xilinx Virtex-7 FPGA VC707 Evaluation Kit (UG885 v1.7).

[40] Texas instruments inc, http://www.ti.com/lit/ug/sluu490/sluu490.pdf

[41] Power management bus (pmbus), http://www.pmbus.org/Home/.

[42] Brennenstuhl primera-line wattage and current meter pm 231-e, https://www.brennenstuhl.co.uk/en-EN/primera-line-wattage-and-current-meter-pm-231-e-gb/ .