

IMPERIAL COLLEGE LONDON

Enhancing FPGA Applications by Applying Codecs to Data Channels

by

Marlon F. Wijeyasinghe

A report submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering
Department of Electrical and Electronic Engineering

November 2019

Declaration of Authorship

I, Marlon Frank Wijeyasinghe, declare that this thesis titled, “Acceleration of FPGA Applications” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at Imperial College London.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Imperial College London or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Copyright Notice

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Abstract

FPGA streaming systems are well suited for high-performance computing (HPC) applications, where the parallelism of algorithms can be exploited using pipelines. The usage of FPGAs have become more common with increasing computational demand, since FPGAs have been very effective at meeting the demand.

Streaming systems are often implemented on heterogeneous platforms, where hardware accelerators on FPGAs are coupled with CPUs. In such cases, the programmer can achieve an improvement in performance by performing the computationally intensive parts of an algorithm on the FPGA. This way, the relative advantages of both software and hardware computation can be exploited. There are, however, challenges that heterogeneous systems face, which are limiting the performance that can be achieved.

This report explores a framework to enhance properties of data channels which transmit data to/from an FPGA kernel. Currently, the framework is applied to a PCI-express channel between an FPGA kernel and a CPU on a heterogeneous CPU-FPGA system by using of spare CPU and FPGA resources. Compute-intensive codecs such as compression are applied at line rate while maintaining the simplicity of a high-level abstraction. Encoding/decoding is done by splitting the data stream into segments and having multiple codec threads processing different segments. Multi-threading in software also allows the overlap of encoding, data transmission and decoding thereby functioning as a software pipeline. Hardware codecs are pipelined.

A number of compression codecs are applied using the framework and a model is developed to predict the run-time performance when codecs are applied, without the need to do a full implementation and benchmark.

Acknowledgements

First of all, I would like to thank my supervisor Dr David Thomas for sharing his ideas and providing guidance throughout the project.

I must also thank my colleagues in the Circuits and Systems Research Group, who have been a pleasure to work with. They provided their assistance and support on various technical and non-technical issues, as well as the board gaming nights, and other regular events, which provided the occasional escape from the world of research.

Some of my amazing friends here in London, and those who are half way across the world also provided an incredible amount of support throughout the past few years, without which the completion of this project would not have been possible.

I would like to thank Maxeler Technologies for providing equipment and software for this project, as part of the Maxeler University Program. Those on the Maxeler Developer Exchange forum also provided advice on various technical issues that were encountered during the project. Additionally, I would like to Nallatech for providing hardware and driver support for their FPGA platform.

This work was sponsored by the UK Engineering and Physical Sciences Research Council, and I would like to offer my gratitude to them for supporting this project.

Lastly, I also would like to thank Wiesia Hsissen for all her effort in managing the Circuits and Systems Research group.

Contents

Declaration of Authorship	2
Copyright Notice	3
Abstract	4
Acknowledgements	5
List of Figures	9
List of Tables	13
Abbreviations	14
1 Introduction	15
1.1 Motivation	15
1.2 Thesis Outline	18
1.3 Contributions	18
2 Background	20
2.1 Bandwidth	21
2.1.1 Rent's Rule	31
2.2 Adaptive Compression	32
2.3 Alternative Applications	35
2.4 Networking	35
2.4.1 Bitstream Compression	37
2.5 Design Time and Run Time	38
2.6 Power Consumption	39
2.7 Summary	39
3 A Multi-codec Platform to Enhance Data Channels in FPGA Streaming Systems	40
3.1 Introduction	40
3.2 Proposed Framework	42
3.3 Codecs	45
3.3.1 Delta Compression	47
3.3.2 Lossless Floating Point Compression	48

3.3.3	Lossy Double Precision	48
3.3.4	RGB32 to RGB24 Conversion	49
3.4	Implementation	49
3.5	Mathematical Model	56
3.6	Evaluation	62
3.6.1	Varying Segment Size with Number of Threads	64
3.6.2	Varying Segment Size with Total Data Size	67
3.6.3	Varying Total Data Size	68
3.6.4	Comparison with Model	70
3.6.5	Miscellaneous	71
3.6.6	A Real-life Application	73
3.6.7	Summary of Performance Metrics	75
3.7	Conclusion	76
4	An Adaptive Compression Framework for FPGA Applications	78
4.1	Introduction	78
4.2	Proposed Framework	80
4.3	Implementation Details	82
4.4	Codecs	83
4.4.1	Delta Compression	83
4.4.2	Lossless Floating Point Compression	84
4.4.3	Identity Codec	84
4.5	Applications	84
4.6	Mathematical Model	85
4.7	Evaluation	87
4.7.1	Comparison with Model	88
4.7.2	Adaptive Compression	90
4.8	Applications	92
4.9	Conclusion	92
5	Application of the Framework to Non-streaming Platforms	94
5.1	Introduction	94
5.2	Conceptual Framework	96
5.3	Codecs	99
5.3.1	Lossy Double Precision	100
5.3.2	Lossless Floating Point	100
5.4	Implementation	101
5.5	Mathematical Model	102
5.6	Evaluation	103
5.6.1	Comparison with Model and Model Prediction	104
5.6.2	Varying the Data Sparsity in Lossless Floating Point Compression	106
5.6.3	Varying Segment Size	108
5.6.3.1	Lossless Floating Point	109
5.6.3.2	Lossy Double Precision	109
5.6.3.3	Summary of Performance for Varying Segment Size . . .	112
5.6.4	Memory Compression	113
5.6.4.1	Lossless Floating Point	113

5.6.4.2	Lossy Double Precision	115
5.6.5	Resource Usage	117
5.6.5.1	Memory Bandwidth vs Resource Usage: Lossless Floating Point	118
5.6.5.2	Memory Bandwidth vs Resource Usage: Lossy Double Precision	118
5.6.6	Summary of Parameters	119
5.6.7	Miscellaneous	122
5.7	Conclusion	124
6	Publications	126
7	Conclusion	127
7.1	Future Work	130
7.1.1	Power	130
7.1.2	Codecs and Applications	131
7.1.3	Automatic Optimisation of Parameters	132
7.1.4	Other Streaming and Non-Streaming Interfaces	133

List of Figures

1.1	Block diagram of the original system before a codec is applied.	
		17
1.2	Block diagram of the system when a codec is applied. The green blocks represent the extra resources used by the codec and the blue blocks represent existing in-use resources.	17
3.1	Block diagram of the original system before a codec is applied.	44
3.2	Block diagram of the system when a codec is applied. The green blocks represent the spare resources used by the codec and the blue blocks represent existing in-use resources.	45
3.3	Block diagram of the system when a codec is applied with multiple instances of the codec in software.	45
3.4	Diagram showing how queues are managed by the threads.	51
3.5	The design flow when codecs are applied using the framework.	54
3.6	Conceptual timing diagram showing performance of original implementation and the performance of sequential compression, without using the framework.	57
3.7	Conceptual timing diagram showing interleaved compression, kernel computation and decompression with a saturated kernel operating at maximum capacity. Number of encode threads, $n_e = 2$, $n_d = 2$, $r_e/n_e^a = 0.001s/MB$, $r_d/n_d^a = 0.001s/MB$	58
3.8	Conceptual timing diagram showing interleaved compression, kernel computation and decompression where the compression is too slow to saturate the kernel. $n_e = 2$, $n_d = 2$, $r_e/n_e^a = 0.002s/MB$, $r_d/n_d^a = 0.001s/MB$	58
3.9	Conceptual timing diagram showing interleaved compression, kernel computation and decompression where both the compression and decompression are too slow to saturate the kernel. $n_e = 1$, $n_d = 1$, $r_e = 0.002s/MB$, $r_d = 0.002s/MB$	59
3.10	Conceptual timing diagram showing interleaved compression, kernel computation and decompression where the decompression is too slow to saturate the kernel. $n_e = 2$, $n_d = 2$, $r_e/n_e^a = 0.001s/MB$, $r_d/n_d^a = 0.002s/MB$	59
3.11	Combined timing diagrams showing relative performance of transmitting raw data, sequential compression (1 thread), and multi-threaded compression. The image aims to illustrate that with a sufficient number of compression and decompression threads, a speedup can be achieved. If the number of threads are insufficient, the performance will slow down rather than speed up.	62

3.12	Speedup for 2GB data against segment size for lossless float compression. The peak performance of 1.33x was achieved with 7 threads. At 9 threads (not shown on graph), the performance was very slightly slower than with 7 threads, and it was not clearly distinguishable on the graph – hence the graph for 9 threads has not been included.	64
3.13	Speedup for 2GB data against segment size for delta compression. The peak performance of 1.58x was achieved with 9 threads. The performance for 9 threads was marginally higher than the performance for 7 threads. . .	64
3.14	Comparison of the performance of both codecs under threaded conditions.	66
3.15	Comparison of speedup against segment size for different total data sizes when using delta compression.	67
3.16	Comparison of speedup against segment size for different total data sizes when using float compression.	67
3.17	Speedup against total data size for lossless float compression.	68
3.18	Speedup against total data size for integer delta compression.	69
3.19	Modelled/actual time for lossless float compression with 3 and 7 threads. 7 threads are used here since it gave the best empirical performance. . . .	70
3.20	Modelled/actual time for delta compression with 3 and 9 threads. 9 threads are used here since it gave the best empirical performance.	70
3.21	Speedup for 2GB data against segment size for RGB32-RGB24 image conversion showing a peak of 1.23x	71
3.22	Speedup for 2GB data against segment size for double-single float conversion showing a peak of 1.85x	72
3.23	Ratio of time taken for lossy double precision to single precision conversion	72
3.24	Ratio of time taken for RGB32-RGB24 image conversion	73
3.25	Speedup for 2GB data against segment size for double-single float conversion applied to an FFT kernel	73
3.26	Speedup for 2GB data against segment size for double-single float conversion	74
3.27	Comparison of speedup against segment size for different total data sizes when using double-single float conversion applied to an FFT kernel. . . .	75
4.1	Block diagram showing dataflow in the system when codecs are applied. .	81
4.2	Modelled/actual time for delta compression with 2, 3, 4 threads.	89
4.3	Modelled/actual time for delta compression with 3 and 9 threads. 9 threads are used here since it gave the best empirical performance.	89
4.4	Graph showing performance of adaptive compression with two codecs. “Delta (normal)” shows delta compression applied to compressible data. “Delta (random)” shows the worst case scenario, where the delta codec was applied to incompressible random data. “Delta/identity random” shows the compression being overridden by the identity codec where the data is random and incompressible.	90
4.5	Graph showing performance for mixed data using a single codec and using three adaptive codecs.	92
5.1	Block diagram showing dataflow when the lossless floating point compression algorithm is applied to sparse data.	101
5.2	Block diagram showing dataflow in the decompression of floating point data.	101

5.3	The model is tested on line-rate compression where data is simultaneously compressed and transmitted across the PCIe channel.	104
5.4	Comparison of the time taken between measured values and predicted values for lossless floating point compression.	105
5.5	The performance of compression, transmission across PCIe and decompression are investigated separately in this section.	106
5.6	Graph showing how the sparsity of data affects the transmission time between the CPU and on-card memory, and the decompression time on FPGA hardware for a lossless floating point compression algorithm. . . .	106
5.7	Graph showing how the sparsity of data affects the compression time, transmission time between the CPU and on-card memory, and the decompression time on GPU hardware for a lossless floating point compression algorithm.	106
5.8	Compression being applied as data is transmitted from software to the onboard memory of the hardware (the same as in the model)	108
5.9	Compression being applied as data is transmitted from software to the onboard memory of the hardware and then decompression being applied before it is transmitted through an FPGA kernel.	109
5.10	Speed ratio against the segment size for lossless compression of sparse floating point data for different total data sizes, where software-pipelined data is transmitted into the on-board memory of the FPGA board. . . .	110
5.11	Speed ratio against segment size for lossless compression of sparse floating point data for different total data sizes, where software-pipelined data is transmitted into the on-board memory of the FPGA board and the FPGA kernel is executed (an unroll factor of 8 was used here).	110
5.12	Speed ratio against the segment size for lossless compression of sparse floating point data for different unroll factors in hardware. The FPGA kernel is executed in this instance.	110
5.13	Speed ratio against the segment size for lossy compression of floating point data for different total data sizes, where software-pipelined data is transmitted into the on-board memory of the FPGA board.	111
5.14	Speed ratio against the segment size for lossy compression of floating point data for different total data sizes, where software-pipelined data is transmitted into the on-board memory of the FPGA board and the FPGA kernel is executed (a SIMD factor of 8 was used here).	111
5.15	Speed ratio against the segment size for lossy compression of floating point data for different SIMD factors, where software-pipelined data is transmitted into the on-board memory of the FPGA board and the FPGA kernel is executed).	111
5.16	Decoding compressed data in the onboard memory of the FPGA card . .	113
5.17	Graph showing decompress FPGA kernel time for lossless floating point as the data sparsity increases (for different unroll factors and a work size of 1)	114
5.18	Graph showing decompress GPU time for lossless floating point as the data sparsity increases (for different unroll factors and a work size of 1) .	114
5.19	Graph showing decompress FPGA kernel time for lossless floating point (for low, medium and high sparsity) as unroll factor increases (Work size: 1)	114

5.20	Graph showing decompress GPU time for lossless floating point (for low, medium and high sparsity) as unroll factor increases (Work size: 1)	114
5.21	Graph showing the kernel time of a lossy double precision codec on an FPGA as the unroll and SIMD factors increase (Work size: 1024)	115
5.22	Graph showing the kernel time of a lossy double precision codec on a GPU as the unroll and SIMD factors increase (Work size: 1024)	115
5.23	Graph showing the kernel time of a lossy double precision codec on an FPGA as the work size increases (Work size: 1024)	116
5.24	Graph showing the kernel time of a lossy double precision codec on GPU as the work size increases (Work size: 1024)	116
5.25	Graph showing percentage of FPGA resources used as the unroll factor increases for lossless floating point (Work size: 1)	117
5.26	Graph showing the resource usage as the unroll factor increases for the lossy double precision codec (Work size: 1024)	117
5.27	Graph showing the resource usage of a lossy double precision codec as the SIMD factor increases (Work size: 1024)	117
5.28	Graph showing the resource usage of a lossy double precision codec as the work size increases with no unrolling and a SIMD factor of 16.	117
5.29	Graph showing the variation in bandwidth with ALM usage for lossless floating point (for different unroll factors and a work size of 1)	119
5.30	Graph showing the variation in bandwidth with register usage for lossless floating point (for different unroll factors and a work size of 1)	119
5.31	Graph showing the variation in bandwidth against the usage of block memory bits for lossless floating point (for different unroll factors and a work size of 1)	119
5.32	Graph showing the memory bandwidth against ALM usage for the lossy double precision codec. The work size is varied for a fixed SIMD factor of 16, and the unroll and SIMD factors are varied for a fixed work size of 1024	120
5.33	Graph showing the memory bandwidth against register usage for the lossy double precision codec. The work size is varied for a fixed SIMD factor of 16, and the unroll and SIMD factors are varied for a fixed work size of 1024	120
5.34	Graph showing the memory bandwidth against the usage of block memory bits for the lossy double precision codec. The work size is varied for a fixed SIMD factor of 16, and the unroll and SIMD factors are varied for a fixed work size of 1024	120

List of Tables

2.1	Comparison of some existing techniques against the framework discussed in the first chapter of this report.	29
2.2	Comparison of existing techniques against one of the frameworks discussed in this report	34
3.1	Variables for Equation 3.1	43
3.2	Parameters for model and their typical values	56
3.3	Comparing the original bandwidth, effective bandwidth, end-to-end latency, speedup and compression ratio for different compression techniques. The measurements were taken using a MAX3 card running MaxelerOS 2015.2.	76
4.1	Parameters for model and their typical values	85
5.1	Table showing a summary of the parameters for lossless floating point. . .	121
5.2	Table showing a summary of the parameters for lossy double precision. . .	121
5.3	Table showing how compute units and local memory affect the decompression kernel time in hardware. All times are in milliseconds.	123
5.4	Table showing how compute units and local memory affect the global memory write bandwidth in hardware. The write bandwidth is the speed at which decompressed data is transmitted out of the decompressor. . . .	123

Abbreviations

FPGA	F ield P rogrammable G ate A rray
HLS	H igh L evel S ynthesis
HPC	H igh P erformance C omputing
RTL	R egister T ransfer L evel
SoC	S ystem-on- C hip

Chapter 1

Introduction

1.1 Motivation

Modern day FPGA streaming systems are ideal for high-performance computing (HPC) applications since they use pipeline parallelism and replicated pipelines to achieve high speed computation, but they face a variety of challenges. The usage of FPGAs have grown as computational demands have increased, and FPGAs have been very successful in meeting this demand.

Streaming systems are often heterogeneous – where they consist of CPUs and hardware accelerators such as FPGAs. Such systems are particularly useful since the programmer can shift the computationally intensive parts of the software into the hardware accelerator. This way, the programmer can exploit the relative advantages of both the CPU and the hardware accelerator to maximise performance (as opposed to homogeneous systems). However, there are challenges that are now limiting the benefits that can be achieved.

A major challenge that FPGA streaming systems face is limited data bandwidth to/from the chip. The FPGA kernel is capable of high-speed computation, but in the case that the channel supplying data to the kernel is incapable of providing data at the required speed, sometimes it is not possible to make full use of the kernel’s computational capabilities. The amount of data that can be transferred to and from the FPGA is limited by the number of pins on the chip and the clock frequency of the channels supplying the chip.

In order to mitigate such problems in a streaming system, the HPC programmer would have to design complicated hardware. This adds complexity to the design process reduces the benefits of working with FPGAs in streaming systems.

An alternative solution would be to invest in newer hardware such as an FPGA with a larger number of pins and channels with higher clock frequencies. This, however is very expensive and so it is not practical in many cases.

This thesis describes a framework that makes use of spare or any available capacity on the hardware and software sides of the channel to improve its functionality to address the problem of bandwidth. The additional capacity could occur as a result of algorithms targeting smaller, older FPGAs being ported to newer and larger FPGAs; or it could involve achieving a balance between the resources consumed by the framework and the existing streaming application. Compression codecs can be applied on the software and hardware sides of the channel, to reduce the size of the data being transmitted across the channel. Software codecs must be multi-threaded and hardware codecs must be pipelined such that they operate at line rate (channel data rate) to avoid reducing bandwidth in the system. Using compression codecs operating at line rate will increase the effective bandwidth.

On the software side, the data stream is broken up into segments and each segment is processed by a sequence of threads: encoding, kernel transmission and decoding. Essentially, it functions as a software pipeline and the threads make use of unused CPU cycles. On the hardware side, the decoding module is inserted into the beginning of the pipeline before the computation kernel and the encoding module is inserted into the end of the pipeline, after the computation kernel. Data is automatically decoded prior to computation and encoded after computation. Different codecs such as delta compression and floating point compression are applied to increase the effective bandwidth of the channel.

Throughout the thesis, the compression ratio, c , is defined by the following formula, where S is the total size of the original data before compression, and S_c is the size of the data after compression:

$$c = \frac{S_c}{S} \tag{1.1}$$

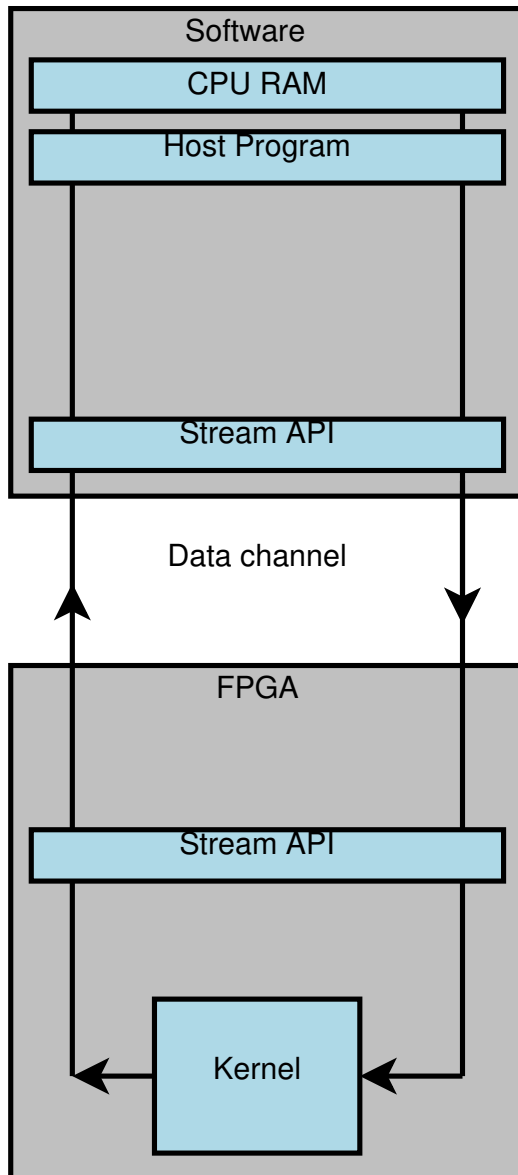


FIGURE 1.1: Block diagram of the original system before a codec is applied.

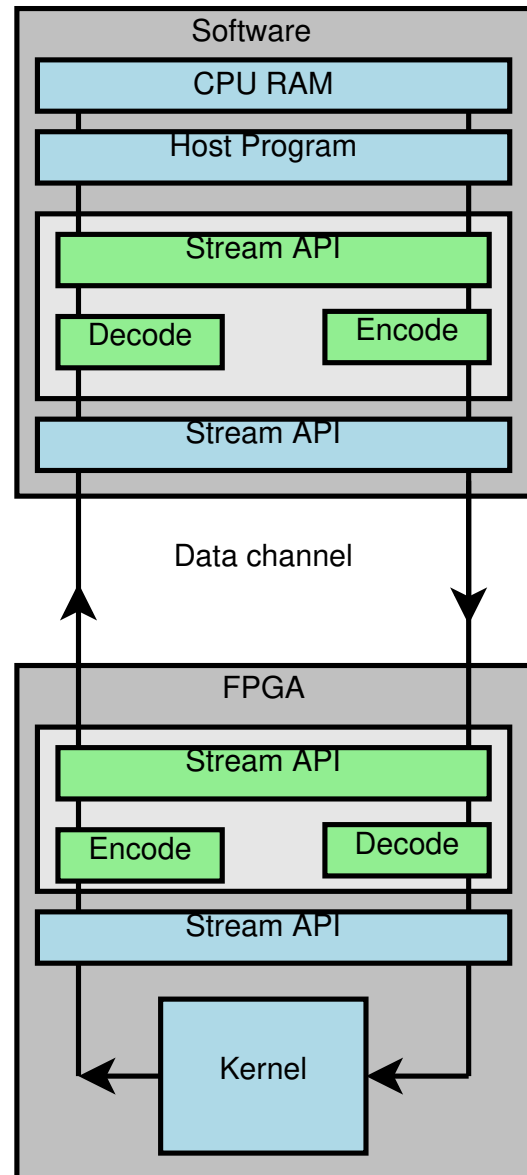


FIGURE 1.2: Block diagram of the system when a codec is applied. The green blocks represent the extra resources used by the codec and the blue blocks represent existing in-use resources.

Fig. 1.1 shows a block diagram of what a heterogeneous streaming system looks like without any codecs. Fig. 1.2 shows what the modified system would look like when compression and decompression are applied in both software and hardware.

1.2 Thesis Outline

Chapter 2 of the thesis contains the background and literature review, where existing attempts to solve the problems are discussed, along with work that is somewhat similar to the work discussed in this thesis. The existing work is analysed and compared against the work done in this project. Several implementations of compression algorithms are discussed, however, none of the existing implementations are particularly suitable for high bandwidth heterogeneous systems.

Chapter 3 proposes a framework to add compute-intensive line rate codecs by using spare CPU cycles on the software side and spare FPGA resources on the hardware side of the channel in heterogeneous streaming systems. A concrete implementation of the framework is developed on a commercial platform. A mathematical model to describe the performance of a streaming system operating under this framework is also developed.

Chapter 4 extends the framework to function adaptively, where the type of compression changes adaptively at run time depending on the nature of the data being transmitted.

Chapter 5 applies the framework to another platform to demonstrate its generality, and also extend the mathematical model to describe the performance of the framework operating on a different platform.

Chapter 6 lists the work that has been published so far during the project.

Chapter 7 summarises the achievements in this project and concludes the discussion.

1.3 Contributions

This thesis discusses a number of original contributions that are made towards the field of using FPGAs to accelerate HPC applications.

The contributions of the thesis are:

- A multi-threaded framework to apply compute-intensive codecs to a segmented data stream by using spare or extra resources in software and hardware for high-bandwidth applications.

- The trade-off between resource usage and bandwidth when extra software and hardware resources are used to improve performance.
- A concrete implementation of the framework on a commercial platform using C code on the CPU side and OpenSPL code on the FPGA side.
- A mathematical model for predicting the end-to-end transmission latency when the framework is applied to a streaming system and comparison of model predicted performance with empirical data to show the viability of the model.
- Evaluation of data transmission performance across a bandwidth limited PCI-express channel on a heterogeneous system with different compression codecs.
- An adaptive system which can apply simultaneous multiple compression codecs into the software-hardware data channels used to transmit data into an FPGA during run-time and the ability to switch between codecs at run-time.
- Evaluation of the run-time performance of a real life streaming application (FFT) with a codec.
- An implementation of the framework on an OpenCL platform to demonstrate the generality of the framework.
- An extension of the mathematical model and its application to the framework on a different platform to show its generality.
- Evaluation of memory compression performance, showing a decompression bandwidth of up to 7805MB/s for lossless floating point.

Chapter 2

Background

State-of-the-art FPGA streaming systems are ideal for high-performance computing (HPC) since they can accelerate certain computationally intensive algorithms by exploiting their parallelism and/or using replicated pipelines. This is done by transmitting data along a pipelined computation kernel at a high throughput. Extensive previous work on streaming applications have demonstrated that FPGAs can often outperform multi-core CPUs in terms of speed and energy efficiency. Such implementations are suitable where large amounts of data are transmitted such that the total transmission time is much larger than the pipeline latency; thereby making the implementation tolerant to pipeline latency (this is not the same as the end-to-end latency, which is discussed later).

While FPGAs have massive computational capabilities, a major challenge that FPGA systems face is limited data bandwidth to/from the chip. If the channel supplying data to the FPGA kernel is incapable of providing data at the required speed, it is not possible to make full use of the kernel's computational capabilities. Investing in new hardware with higher bandwidth is often impractical due to the cost involved. An alternative way to mitigate the bandwidth issue is by using compression. Compression has, for a long time, been used in computing applications due to the discrepancy between computational capability and input/output data rates. While compression was in the past used to conserve storage space on disks, the growing discrepancy between processing speed and data bandwidth has resulted in compression being used in high bandwidth applications [1]. The massive computational capabilities of FPGA systems increases the feasibility and the necessity of using compression. To mitigate the bandwidth issue in an

existing high-level streaming system, the HPC programmer has to design compression at RTL or include pseudo-RTL code within the high-level code if they are using an HLS tool. This unfortunately adds complexity to the design process in both software and hardware.

Adding compression to heterogeneous CPU-FPGA systems poses several major challenges. If compression were to be applied, it must be performed at line rate on the data as it is transmitted. This requires the compression to have a high throughput – the compression must be performed at a throughput that is higher than the bandwidth of the channel, in order to overcome the bandwidth bottleneck. Getting a specific compression algorithm to run at a high throughput on both software and hardware is a major challenge. Most fast compression algorithms will often work efficiently on either software or hardware, but not both. Achieving a good compression ratio is important too, however, a good compression ratio is not very useful if the compression algorithm itself is very slow. There is also a challenge of resource usage, since the compression must share resources with whatever existing application it is being applied to. Therefore, this is a trade-off between throughput, compression ratio, and resource usage. It may also be the case that when an algorithm that was previously implemented on an older/smaller FPGA is ported to a newer/larger FPGA, there would be spare resources available.

If compression is to be applied transparently without affecting the function of the original system that is being modified, lossless compression should ideally be used. Lossy compression may be used in applications where a loss of precision would not affect the result of calculations. Applying lossy compression adds another factor – loss of precision. This becomes a trade-off not just between throughput, ratio and resources, but also precision.

2.1 Bandwidth

The bandwidth problem arises due to limitations in the capability of data channels. The I/O bandwidth of the chip is restricted by the number of I/O pins and the clock frequency of their corresponding registers. The physical characteristics of the off-chip data channel(s) which are used to access the FPGA, including its clock frequency can also limit the bandwidth with which the FPGA can be accessed externally.

The bandwidth problem arises due to limitations in the capability of data channels in high-bandwidth applications. The I/O bandwidth of the chip is restricted by the number of I/O pins and the clock frequency of their corresponding registers. The physical characteristics of the off-chip data channel(s) which are used to access the FPGA can also limit the bandwidth with which the FPGA can be accessed externally. For example, this work primarily uses the Maxeler [2][3] heterogeneous CPU-FPGA platform.

The Maxeler MPC-C500 machine, used for most of the experiments in this thesis, transmits data between an Intel Xeon CPU and a Xilinx Virtex 6 FPGA via PCI-express 2.0 x8. The PCI-express channel has a theoretical bandwidth of 4GB/s in either direction (total 8GB/s), but the Maxeler card itself only has a theoretical bandwidth of up to 2GB/s in either direction (total 4GB/s). When the duplex channel bandwidth was measured under the low-latency interface (discussed later), it was observed to be 2690MB/s.

If the theoretical bandwidth of 2GB/s is achieved, assuming simultaneous transmission in both directions and assuming that one token of data is transmitted on each clock cycle: If a pipelined computation kernel on the FPGA has a 128-bit wide data bus, the channel bandwidth is sufficient to support the kernel being clocked at up to 134MHz. According to the measured bandwidth however, a 128-bit wide kernel clocked at up to 88.1MHz will be supported (or 64-bit wide kernel clocked at up to 176MHz). Again, this assumes that one token of data is transmitted per cycle. Even if a computation kernel is not constantly sending and receiving data on each cycle, these bandwidth limitations would apply to any bursts of data that are transmitted. A kernel with a higher clock frequency will not run at full capacity since the channel supplying the kernel is a bottleneck.

The OpenCL platform is also used and this is done mostly on a Nallatech 385 card [4]. The Nallatech card contains an Altera Stratix V FPGA and the host system has two Intel Core i3 CPU cores. The driver for the Nallatech card supports a bandwidth of approximately 3.1GB/s (according to the specification), which is slower than the bandwidth of the PCI-express channel that it uses, so a heterogeneous HPC application based on this platform would also experience a bandwidth bottleneck. The bandwidth on the Nallatech system was measured to be 2308MB/s. The OpenCL platform is used on a heterogeneous CPU-GPU system too, with an Nvidia Quadro K4000 GPU and two

Intel Core i3 CPU cores. The PCIe channel of the CPU-GPU system has a measured bandwidth of 2990MB/s.

To tackle the bandwidth bottleneck on streaming systems, several different compression methods have previously been implemented on FPGAs. Specific compression methods are often required in different HPC applications that have been implemented on FPGAs, due to the nature of the data that they handle and the data throughput involved. For example, high throughput floating point hardware compression and decompression have been design to be placed at either end of a Lattice Boltzmann Method (LBM) streaming kernel to improve effective memory bandwidth. The bandwidth was the bottleneck since the nature of the algorithm requires the kernel to transmit redundant data to/from memory to parallelise the algorithm. One such example is a segment-parallel hardware based implementation for streaming floating data by Sano[5], which achieved a compression ratio of around 3.8. Another such implementation of hardware compression to improve memory bandwidth for LBM accelerators was designed by the authors of [6] and it achieved a compression ratio of around 3.5, hence a 3.5x increase in memory bandwidth since the algorithm operates at line rate.

Similarly, sparse matrix vector multiplication is a memory-bound algorithm that could benefit from compression. An algorithm [7] for compressing redundant non-zero data using spare FPGA resources achieved average compression ratios of 1.14-1.40 and a maximum ratio of 2.65. Image compression is also a common in FPGAs. An implementation of JPEG-LS compression for satellite images achieved a speed of 75 megapixels per second [8]. It achieves a lower compression ratio than conventional JPEG compression, but the fast nature of the algorithm is beneficial for transmitting data. An HLS based video compression codec was implemented on an embedded system using a combination of software and hardware [9]. This is a case of the software and hardware working together to compress data, rather than data being transmitted between a software entity and hardware entity. These compression methods are specific to the data type being compressed and these are optimised for the specific algorithm for which they were designed.

The benefit of implementations such as [5] and [7] mentioned above is that an improvement in performance can be achieved without modifying the computation kernel. The frameworks described in this report also function under the principle of not modifying

the kernel but focusing the enhancements on the data channels to/from the kernel. However the emphasis of the work in this thesis is broader than purely accelerating memory bandwidth-bottlenecked HPC applications by implementing state-of-the-art compression algorithms in hardware – but rather to use different compression algorithms to demonstrate the capability of the proposed framework. The framework aims to enhance communication between software-hardware boundaries – with the potential for non-compression based enhancements.

While the existing compression methods described above are specific for a certain application and a certain type of data, some general purpose compression algorithms have also been implemented on FPGAs previously. These include ZLIB, Gzip and several LZ-based algorithms.

For example, the authors of [10] have built a streaming ZLIB decoder on a Xilinx Virtex 5 FPGA using RTL (VHDL). By converting a sequential software implementation of the algorithm into a parallelised hardware implementation, they achieved a throughput of 125MB/s and a 2.83x speedup in the FPGA compared to a software implementation.

The authors of [11] made an optimised compression core, which performs on-the-fly LZ-based compression and decompression with a higher throughput than existing implementations (at the time) and also a lower resource usage. This implementation has a throughput of 16MB/s.

The work in [12] is about a high performance and power efficient RTL design of the LZW algorithm. The design achieved a compression bandwidth of 88MB/s and a decompression bandwidth of 160MB/s on a Xilinx Virtex 2 FPGA, and had a power consumption of 333mW. When it was designed, its throughput was relatively high compared to the commercial compression and decompression hardware accelerators of that time. Another implementation, this time of LZW decompression [13], achieved a throughput of 280MB/s on a Xilinx Virtex 7 FPGA. The authors of [13] designed their decompression core to use limited resources, such that multiple instances of it can operate concurrently.

The authors of [14] and [15] describe a hardware implementation of the LZMA compression on a Xilinx Virtex 6 FPGA. This implementation has a throughput of up to 16MB/s. Their aim was to improve performance and reduce power usage by using hardware-based compression in comparison to software-based compression.

Another popular compression algorithm is Gzip which is based on the Deflate [16] algorithm. The work in [17] discusses a hardware-based implementation of Gzip for Internet data compression services. This implementation achieved a compression bandwidth of 110MB/s and a decompression bandwidth of 306MB/s. The authors observed that this improved the effective bandwidth of disk I/O (for downloads and uploads), accelerated software programs and freed up CPU resources.

The authors of another paper have implemented a compression core [18] on a Xilinx Virtex 6 FPGA to offload Gzip compression onto a hardware accelerator from the CPU. They achieved an average compression throughput of 6.5MB/s and a peak throughput of 166MB/s. The authors observed their compression core to operate at more than twice the speed compared to an Intel Core i7. The data is transmitted to the FPGA via PCIe using the Xillybus PCIe core [19]. Xillybus is a PCIe core which facilitates data streaming to and from FPGAs across a PCIe channel on heterogeneous systems. However, compression is not applied when transmitting data across the PCIe channel.

While it is important to note that some of these implementations were evaluated on older technology (therefore they will be faster on newer technology), the ones evaluated on older technology are at least an order of magnitude slower than the sorts of channel bandwidths that are being dealt with in the heterogeneous systems that this thesis investigates (2–3GB/s).

While general purpose algorithms are capable of achieving good compression ratios, they are not necessary suitable for heterogeneous applications that require line rate performance. For line rate compression in high-bandwidth streaming systems, a fast compression speed is the most important factor rather than the compression ratio. These existing implementations are not able to meet the line rate requirements of state-of-the-art FPGA streaming systems.

Additionally, FPGA implementations of general purpose compression algorithms are typically not designed to share the FPGA resources with another streaming kernel application – instead, these implementations are designed to perform a particular compression algorithm as fast as possible (usually in comparison to a software implementation), for the data to be transmitted elsewhere. The heterogeneous compression in this thesis is intended to share the space inside the FPGA with a separate I/O bound application

and also share the CPU resources. If an existing application is I/O bound (for example, parallel applications with replicated kernels), then some of the resources that would otherwise be used for this application could now be used to apply compression instead. So this becomes a trade-off between resource usage and bandwidth.

Furthermore, an important aspect of the work done in this thesis is latency of compression. In this thesis, the end-to-end latency is the time taken from the start of data transmission from the source to the end of data transmission at the destination. In the special case of duplex transmission, the source and destination are the same and the end-to-end latency will be the round-trip time. When compression is inserted in the middle of a data transmission instance, it adds an additional stage of computation which will add to the end-to-end latency, and the design of the framework in this thesis aims to minimise the latency. This is one of the main considerations of the work in the thesis. It is similar to how compression would be applied in networking applications. Many of the existing implementations of compression are not aimed at such situations, and their primary consideration is the total throughput achieved.

There have been attempts at compression targeted at high-bandwidth FPGA applications. The Titan-R [20] hardware compressor core was shown to perform the LZ277 algorithm at 1096MB/s. This implementation makes use of fine-grained pipeline parallelism and exploits the high-speed on-chip block RAMs to perform compression at a high throughput.

The work in [21] describes an LZ277 compression core for multigigabit networks; two variations of it were shown to have a throughput of 1113MB/s and 1173MB/s. This implementation is a high throughput architecture that is applicable to a wide range of network technologies and interfaces. The compression reduces the usage of network bandwidth such that multiple applications can use the network bandwidth – and it is common in networking infrastructure for multiple applications to share bandwidth.

The work in [22] and [23] present the X-MatchPRO lossless data compressor which has an internal throughput of 100MB/s and a duplex compression and decompression performance of 200MB/s. It is implemented on a A500K130 ProASIC FPGA and it is specifically targeted gigabit data networks and storage applications. Following this, an improved version [24] of it has an internal throughput of 200MB/s and a duplex combined compression and decompression performance of 400MB/s.

The work in [25] describes applying LZSS (based on LZ277) compression when logging stream of data in embedded networking applications. Compression is performed on-the-fly at up to 52MB/s on a Xilinx Virtex 5 FPGA. The performance is achieved by exploiting the independently addressable nature of dual-port block RAM in the FPGA to speed up the processing of data.

A high throughput streaming implementation of the lossless 842B algorithm is discussed in [26]. This implementation has two compression and two decompression engines running in parallel and it was measured to have a duplex compression and decompression throughput of 2723MB/s. It achieved a 13.6x speedup compared to a reference software implementation. This algorithm has duplex bandwidth that exceeds the bandwidth of the channel and is relatively small in terms of hardware resource usage. If a sufficiently fast software version of the algorithm is also available, then this is the only existing algorithm that would work across a software-hardware channel on a streaming system. The others do not have any corresponding software implementations.

A previous investigation that is quite relevant to the work discussed in this thesis is [27] which is a quantitative analysis of heterogeneous CPU-FPGA platforms. They investigate the newer QPI-based Intel HARP platform (Stratix V) and the PCIe-based Alpha Data board (Virtex 7). HARP is a modern heterogeneous platform where both the host processor and FPGA share a common address space. Most heterogeneous platforms, on the other hand, have separate memory for the host program and the hardware accelerator, requiring memory to be copied between the host memory and hardware (unless they are SoC platforms). This is particularly useful where the hardware accelerator has to perform multiple iterations on the data that is on the memory of the FPGA card. However, where large amounts of data has to be transmitted across PCIe, the transmission proves to be a bottleneck. The systems used in this thesis are of the latter type, where the software and hardware have separate memories. Unfortunately a heterogeneous system of the former type was not available to us for comparison. The authors observed it is not possible in practice to achieve the advertised PCIe bandwidth. For example, for a PCIe channel with an advertised bandwidth of 8GB/s, the measured bandwidth was only 1.6GB/s. The investigation on the platforms used in this thesis had the same issue, though the discrepancy was not quite as bad. So there is a strong case to use compression when transmitting data across PCIe, when the achievable bandwidth is lower than advertised.

The work in [28] proposes an architecture of a parallel array of pipelined Deflate [16] compression engines for high throughput compression on an Intel HARPv2 platform with an Altera Arria 10 FPGA. The hardware accelerator kernel achieved a throughput of 12.8GB/s and a compression ratio of 0.49. The end-to-end compression throughput across the entire system achieved a speed of 3.9GB/s on the HARP platform and 10.0GB/s on the HARPv2 platform. These are heterogeneous platforms and a notable difference between these and the platforms that this thesis works with is that the FPGA card on HARP/HARPv2 can directly access the host system memory. Though the study in [28] is performed on a heterogeneous platform, it must be noted that compression here only happens within the FPGA where it reads uncompressed data from the host system memory directly. Compression is not used when data is read from the host system memory in [28], so this is somewhat different from the way compression is applied in this thesis.

The work in [29] by Microsoft discusses an HDL implementation of LZ277 and Huffman coding with a throughput of up to 5.6GB/s. This is one of the highest hardware compression throughputs in existence today. The paper also discusses the compromise between achieving a good compression ratio and minimising chip area usage – compression algorithms which give better compression ratios tend to occupy a larger chip area.

A study in [30] discusses the use of FPGA accelerators in big data computing systems in data centres, and the challenges that companies face in deploying FPGA based accelerators in their data centres. The paper discusses the Blaze framework, which simplifies the process for the software programs in data centres to access FPGA accelerators. Experimental results showed an improvement in both data throughput and energy efficiency in comparison to using only CPU-based computation.

The work in [31] describes a multi-engine hardware implementation of the Xpress9 [32] compressor by Microsoft, on an Altera Stratix V FPGA. The implementation is tested on a heterogeneous system, data is transmitted to the hardware via PCIe and the compression is performed using hardware since it has a faster computational speed than software. With 7 compression engines, the implementation achieves a throughput of up to 307MB/s. Compression is not used for the transmission of data between software and hardware.

TABLE 2.1: Comparison of some existing techniques against the framework discussed in the first chapter of this report.

Feature	Deflate [28]	842B[26]	FPGA2[33]	Threaded decomp[34]	Chapter one
Line rate compression	Yes	Yes	No	No	Yes
Line rate decompression	No	Yes	No	Yes	Yes
Software	No	No	No	Yes	Yes
Hardware	FPGA	FPGA	FPGA GPU	No	FPGA
Homogeneous	Hardware	Hardware	No	Software	Software Hardware
Heterogeneous	No	No	Yes	No	Yes

Platforms similar to what is discussed in this report have been developed in other research, albeit to perform different functions. Such a platform is FPGA2 [33], which is a framework for simplifying communication between FPGAs and GPUs via PCIe on heterogeneous hardware accelerators. The FPGA-GPU framework enables easy data transfer between a GPU and FPGA without the need for a large amount of coordination by a CPU. This is unique in the sense that most heterogeneous systems tend to couple a CPU with either a GPU or FPGA, but not both of them together.

Table 2.1 compares some of the existing methods discussed above with the method discussed in the first chapter of this thesis. The methods listed in the table have some of the qualities (but not all) that the work discussed in this thesis achieves. For example, the Deflate [28] FPGA implementation is a general compression method, and it is currently the fastest hardware compression implementation in existence today, but it does not have an equally fast corresponding software version for it to operate on a heterogeneous system. The 842B algorithm [26] is a high-bandwidth hardware implementation, but it is not a general platform and it is not designed for heterogeneous systems. The general compression platform which allows multiple compression methods and parallel compression to be used [34] is only applicable to software systems and only its decompression is fast enough to operate at line rate. FPGA2 [33] is a framework for communication between GPUs and FPGAs across a PCI-express channel. However, it does not provide compression. Out of the existing work discussed above, none of them provide a high-bandwidth framework for heterogeneous CPU-FPGA systems.

The framework discussed in this thesis could provide a benefit to homogeneous software-software and homogeneous FPGA-FPGA systems too (as shown in Table 2.1), though this has not been tested in this thesis. There are other compression algorithms in existence that are specifically optimised for homogeneous software or homogeneous hardware applications. The compression techniques that are investigated in this thesis are optimised for heterogeneous applications, so directly applying them to homogeneous applications may not provide the same level of benefit that could occur if the compression techniques were optimised for homogeneous applications.

Parallel compression has been implemented previously, but not for transmitting data between hardware and software in heterogeneous CPU-FPGA applications. Often, many of the hardware implementations mentioned above have been tweaked, specially optimised and piepelled to operate at a high-throughput. Sometimes, there are efficient implementations in software of particular algorithms, but no corresponding efficient implementation in hardware, normally as a result of loop dependencies making hardware parallelisation difficult. In other cases, there are efficient implementations in hardware through massive parallelism, but there is no efficient implementation in software. Even if there are efficient implementations in both hardware and software of particular compression algorithms, the challenge is then, whether or not it is possible to combine a hardware and software implementation to work together for online compression (such as the software compression that would happen during data transmission across a network).

One such implementation of parallel compression is a framework [34] that provides an API to incorporate multi-threaded data retrieval and decompression (including a variety of compression and decompression algorithms) into an existing application. Multi-threaded retrieval and multithreaded decompression of data is sufficiently fast to operate at line rate in this design, and it can be used to transmit or retrieve data from local and/or remote storage. Besides multithreading, it also uses prefetching and caching of data to improve the performance.

The work in [35] is an proposal that allows parallel compression of floating point data to be applied to reads and writes on certain types of HPC software applications. It allows an existing simulation program to output and store data in a compressed manner and it allows a data analysis application to read compressed data. Parallel compression of floating point data for reads and writes on HPC software applications for data analysis

and simulations on big data showed a speedup of up to 1.83 and an average speedup of 1.36.

The Massively Parallel Compression (MPC) [36] algorithm for GPUs is designed for large amounts of floating point data. The authors built this algorithm since most commercial compression algorithms (at the time) were not suitable for parallel execution. Complex algorithms such as prediction or dictionary-based methods have a large amount of internal state and are therefore impractical to parallelise. MPC achieved parallelism by using a compression technique with limited internal state, such that it is easier to parallelise by breaking up the data into smaller segments. It achieved a good compression ratio on different data sets which was comparable to the best CPU-based algorithms at the time and outperformed the CPU versions by one or two orders of magnitude in terms of throughput.

2.1.1 Rent's Rule

Bandwidth is set to become an even greater challenge in the future. As circuits become more complicated, the transistors become smaller and the number of gates in the circuit will increase. However, the number of connections between a particular group of gates and the rest of the system will not increase proportionately [37]. In the extreme case where the set of gates covers all the gates on the chip, then the number of connections with the external system will be the number of pins on the chip.

The number of physical pins on the chip does not increase proportionately with the number of gates. On architectures with a large computing power, this could result in the computation consuming and/or producing more data than what the external connections are capable of handling. Other means of transmitting sufficient amounts of data to/from the chip need to be considered if the I/O does not increase proportionately with the complexity of the circuit. Therefore, there is a strong case for compression on such circuits, and the case becomes ever more stronger as transistors become smaller.

2.2 Adaptive Compression

We have just discussed a number of generic and application-specific compression techniques. Moving on from what was discussed earlier, systems could be optimised to use what we describe as adaptive compression. This means that the compression codecs could alter their parameters on-the-fly to adapt to the nature of the data being transmitted. For example, if highly entropic data were being transmitted, compression would not be effective and it would be useful to transmit raw data to avoid a loss of speed.

The bandwidth limitations on input/output are an issue across all parallel computing applications in general, and adaptive compression is sometimes used in software-based computing applications.

An example of adaptive compression in software is the Prediction-based Auto-Adaptive Compression Model (PACM) [38], to optimise storage and performance by adaptively switching between multiple compression algorithms based on the number of packets in the transmission queue. There is an ongoing need to reduce storage space and speed up transmission in big data applications. In applications where large amounts of data are handled, it is not uncommon for the workload and data format to change. When this happens, using a single compression algorithm becomes ineffective. By using PACM, the authors observed a 2-5x improvement in write throughput. When there are a large number of packets in the queue awaiting transmission, the compression algorithm will change to one that has a slower speed but better ratio. If the queue is empty, the algorithm will change into one that has a higher compression speed, albeit with a worse ratio. While the work in that paper does not claim PACM to be suitable for line rate applications, it does discuss an increase in throughput so it may have the potential to work in line rate applications.

Adaptive compression is also useful in scientific applications where large volumes data are produced, transmitted and stored. A major challenge that applying compression encounters in scientific applications is that data is often highly entropic. When dealing with large volumes of highly entropic data that is common in scientific applications, it is difficult to compress using general purpose methods. In such cases, there are specialist

compression methods such as In-Situ Orthogonal Byte Aggregate Reduction Compression (ISOBAR) [39], which is designed for floating point data in scientific applications. It optimises compression at byte-level when it encounters data that is difficult to compress.

A platform called the Adaptive Compression Environment (ACE) [40] applies on-the-fly compression to a communication stream to improve network transfer performance. It works by trying to predict the compressed and uncompressed data transfer time for 32kB sized blocks of data. It automatically selects a compression method that best suits the technology, current performance and nature of the data. It takes into consideration the underlying networking technology and the characteristics of the data to determine the type of compression to apply. It was observed to perform better than some common compression techniques such as Bzip, ZLIB and LZO.

An FPGA implementation of an adaptive compressor called Adaptive Range Coding is discussed in [41]. This is a variation of the adaptive range coding algorithm used by the 7-zip compressor. It uses parallelisation on the FPGA and achieves a compression throughput of more than 50MB/s on a Virtex 5 FPGA. It must be noted that although this algorithm is described as adaptive, it is not adaptive in the sense of it being a platform that switches to different compression methods. Rather, it is adaptive in the sense of being able to adapt to the frequency at which certain letters occur in text. This compression throughput is not sufficient for our application, since the data channel that we use already has a bandwidth of over 2GB/s.

Other work includes adaptive compression algorithms, such as one which proposes two adaptive video encoding algorithms for selecting different Discrete Cosine Transform coefficients for compacting different blocks of a video to achieve better compression [42], though it does not discuss the equipment used for testing or whether the algorithm has any line rate capability. The work discussed in [43] proposes two more similar algorithms for video encoding, the most interesting of the two being one that dynamically finds the best coefficients and transform for the current block of data based on that of all the previous blocks. However, there is no discussion on whether such an algorithm would work at line rate and it does appear to be too computationally intensive to operate at line rate.

TABLE 2.2: Comparison of existing techniques against one of the frameworks discussed in this report

Feature	Chapter 3	ACE [40]	Video Comp [43]	Adaptive Range Coding [41]	PACM [38]	Chapter 4
Line rate	Yes	Yes	Unknown	No	Yes	Yes
Software	Yes	Yes	Yes	No	Yes	Yes
Hardware	Yes	No	No	Yes	No	Yes
Multiple codecs	No	Yes	Yes	No	Yes	Yes
Heterogeneous	Yes	No	No	No	No	Yes
Adaptive	No	Yes	No	Yes	Yes	Yes

Datacomp [44] is yet another adaptive compression platform, which transparently performs compression using a library provided by the platform where it can transmit compressed or uncompressed data depending on the compressibility of the data. The paper discusses how existing adaptive compression systems are not ideal for general purpose computers. It also uses wrappers to support arbitrary compression methods and allows the compression strength to be varied.

Table 2.2 compares some of the methods discussed above with what is proposed by this thesis. The methods in the table have some of the features that the work in this thesis achieves, but not all. For example, in Chapter 3, the work which was published in [45], discusses a framework that performs line rate compression between software and hardware, but the compression is not adaptive and it cannot be changed at run-time. The video compression techniques described in [43] are a useful comparison since it dynamically changes the coefficients of an algorithm. Adaptive Range Coding [41] is the only FPGA implementation that had some degree of adaptivity, though it is not adaptive to the extent of our proposal and it is not fast enough to work at line rate. ACE [40] and PACM [38] actually switches between different compression algorithms, but it is not applicable to hardware.

There are several compression algorithms discussed above that are adaptive to the data being transmitted aimed at meeting the massive storage and bandwidth demands of today's applications. Some of the existing work discussed above demonstrates that combining multiple algorithms to achieve the best possible compression (in terms of ratio and/or throughput) is an effective method. However, there is currently only one implementation that is applicable to heterogeneous systems, and that one is not adaptive.

2.3 Alternative Applications

Another platform that was explored in this research is OpenCL. It is another platform useful to build the sort of heterogeneous systems. This differs from Maxeler in the sense that it is a non-proprietary, open platform that can be applied systems involving CPUs, GPUs, FPGAs and SoCs [46]. The authors of [46] have carried out a thorough investigation of some of the platforms used in state-of-the-art heterogeneous computing, including how they offer high performance computing at a lower cost compared to CPUs. Like Maxeler, OpenCL provides an API for data transmission and communications between the different components of a heterogeneous system.

OpenCL gives the programmer a large degree of control over the types of memory that are used by computation kernels and a number of different ways to parallelise hardware implementations. It differs from a streaming system such as Maxeler, so it has different ways of managing the order of data being processed.

The authors of [47] have implemented low power, high-speed Gzip compression with a throughput of 3GB/s using OpenCL on a Stratix V FPGA. They found that the OpenCL implementation is only 5.3% slower than a commercial implementation of Gzip using Verilog, using 2% more logic and 25% more on-chip block RAM than the Verilog implementation.

2.4 Networking

A related problem is network bandwidth and there is an analogy between the layers of abstraction in networks and the framework that this thesis describes. We work with an HLS tool which abstracts away many details of the function of the FPGA kernel and its communication across the CPU-FPGA data channel. In general networking, there are several layers of abstraction, such as routing, packetisation, data frames and raw bits at the hardware level, and data segmentation, encoding and high-level APIs at the software level. In software, our framework functions at the encoding level and in hardware it functions on the packetisation and raw bit levels.

Networking often involves the transmission of large amounts of data between computers in a setting which is tolerant to latency. The heterogeneous CPU-FPGA platform we

work with is also designed for the transmission of large amounts of data in latency-tolerant applications.

The work in [48] describes a network of interconnected FPGAs and how a design can be partitioned across this network, taking into account the propagation delays of the channels between the FPGAs. In a different project [49], a quad-FPGA ring-based network was implemented, with the scalability of functioning with even a larger number of FPGAs. The quad-FPGA platform was evaluated on the BEEcode BEE3 computing platform. The aim of such homogeneous FPGA systems is to distribute the architecture across a number of smaller, cheaper FPGAs without needing to fit the entire architecture on a single, large one. These are homogeneous systems, and therefore quite different from the sort of systems that this thesis investigates. However, the framework proposed in this thesis could potentially be applied to inter-FPGA communication too.

Both of the major vendors, Altera and Xilinx, have SoCs where the programmable logic and the on-chip CPU are tightly coupled. This allows applications to exploit both the FPGA's parallel computation capability and the CPU's sequential capability. The work in [50] explores methods to reduce end-to-end latency in a heterogeneous CPU-FPGA framework on a Zynq System-on-Chip (SoC) through distribution of sequential and parallel computation across the on-chip CPU and the FPGA fabric respectively. The work that we do using compression is also aimed at reducing the end-to-end latency on a heterogeneous CPU-FPGA system, but the system we work with uses separate hardware chips for the CPU and FPGA instead of a SoC. The overheads of transmission between different chips are higher than within the same chip.

Data compression using FPGAs can be implemented to compress data for Ethernet communication, since Ethernet is often a bottleneck in network communication. For example, the C4 algorithm discussed in [51] was implemented on Xilinx Virtex 7 FPGA, and the authors found a reduction in data transfer time of up to 52.5%.

Project Catapult [52], the accelerator cloud by Microsoft, is a good example of a data centre network. FPGAs are placed between the servers and the network switch(es), such that all data traffic to/from a server is transmitted through the FPGA. Any operations that the FPGA performs on the data will be done at line rate.

In the type of applications we deal with, we treat the data as a continuous stream and choose the granularity by specifying the size of the segments. These segments can be compressed and decompressed at line rate in the same way as it is done for video compression in networking. Similarly, the sort of framework discussed in this thesis could be potentially applied to a cloud computing platform such as Project Catapult, if the codecs are applied on a stream of data as it is transmitted through an FPGA. It must be noted that in the hardware we work with, there is already a fixed level of granularity (PCI-express used 128 bits) – but the level of granularity introduced on top of this by the framework that we design uses much larger segment sizes (32KB to 2GB).

2.4.1 Bitstream Compression

Another common application of compression in FPGAs is bitstream compression to reduce the configuration time and memory storage requirements. Bitstream compression is not investigated in this thesis, but some discussion of it is included in this section for completeness, and some observations from the papers are relevant to streaming compression in general.

The work in [53] and [54] discuss bitstream decompression to improve FPGA configuration time by improving the effective bandwidth of nonvolatile configuration memory for applications that use runtime reconfiguration. An important observation to note from these two papers is that achieving a good average compression ratio is not always effective in the event that the worst case compression ratio is very bad. The parts of the bitstream where the worst case compression ratio occurs will occupy a disproportionately large amount of time during transmission and the configuration memory bandwidth will not be fully utilised. As a result, the authors proposed that limiting the worst case compression ratio should be prioritised, even if the best case and average compression ratio becomes slightly worse, because this will result in a faster throughput overall. This would be true across line rate compression in any application, and not just for bitstream decompression.

As noted by [55], the size of FPGA configuration bitstreams is increasing as the complexity of FPGAs increase. Applications where runtime or partial reconfiguration is used pose an even greater challenge. The memory storage requirements are not insignificant for multiple bitstreams [56], and the configuration time can have an impact on the

overall performance of an application if the configuration is too slow. Several lossless compression techniques, such as those discussed in [55] and [56] exist to mitigate both of these problems. Bitstream decompression algorithms must operate at line rate to ensure that the configuration happens sufficiently fast. The novel decompression algorithm in [56] has a throughput of up 400MB/s on a Zynq FPGA. Other algorithms can also be used, such as the pipelined decompression algorithm discussed in [57], which is based on LZ277 and Bitmask-based Compression (BMC). On a Xilinx Virtex 5 FPGA, the decompression achieved a throughput of 212MB/s.

2.5 Design Time and Run Time

The work in [58] and [59] discuss how HLS tools are used to reduce the design time/effort of a system and increase the productivity of the system designer, since the effort of the designer is very high when using RTL. The hardware kernel(s) of a heterogeneous streaming system will perform fastest (highest possible frequency) if it is designed at RTL by a skilled system designer, but this is not a practical option for large, complex systems due to the effort and time required. The work in [58] present case studies on the application of Xilinx Vivado HLS. It investigates how HLS, while reducing the effort of HDL design, also allows flexibility in the final hardware, that allows the programmer to meet various design constraints. While the work in this thesis does not specifically target HLS applications, one of the goals is to improve performance using the conceptual framework, without the programmer having to do a complex RTL implementation of compression.

The overheads of the high-level abstraction provided by HLS tools often results in the generated hardware consuming more chip area and having a lower maximum operating frequency than hardware that would be written by RTL code.

The slower performance of hardware generated by HLS tools is essentially a trade-off between design time and run time. Designing at a higher level of abstraction generally results in a faster design time albeit a slower run time.

Our work is based on the same principle. We use a high-level abstraction to incorporate additional features into streaming systems, to reduce the burden of RTL design on the

system designer. The additional features that we are adding are those which existing HLS tools do not provide.

2.6 Power Consumption

Unrelated to the above, it must be noted that the framework making use of spare resources will may cause an increase in power consumption, largely from the extra CPU activity taking place and marginally from the increased FPGA resources. However, this work is based on the assumption that the power supply is not limited and so the power consumption is not investigated in this thesis.

There is an existing implementation called Compressed PCI-express (CPCIE) [60], which is a PCIe core that contains a transparent layer of compression and it is based on the Xillybus PCIe core [19]. CPCIE uses the X-MatchPRO [24] compressor and decompressor and provides an API on the software side for streaming data to and from the FPGA kernel. The motivation behind this paper was the issue of PCIe causing a bottleneck that limits acceleration in memory-bound applications. CPCIE focuses primarily on the reduction in power consumption achieved by having to transmit fewer bits as a result of compression and it demonstrates notable reductions in power. It achieved an energy reduction of 84% during PCIe transfer and up to 20% in the whole processing chain as a result of the reduction in the number of bits that need to be transferred across the PCIe channel. It has a throughput of up to 800MB/s [61], which is still slower than the bandwidth of the PCIe channel that we work with.

2.7 Summary

This chapter has discussed heterogeneous FPGA streaming systems, some of the challenges that they face and existing attempts to tackle these. We have also discussed how HLS tools are used to design streaming systems along with the benefits and drawbacks of implementing systems by using such tools.

Previous attempts to simplify communication across channels, including PCI-express have been discussed. Attempts to tackle the bandwidth bottleneck without making changes to existing kernel code have also been discussed.

Chapter 3

A Multi-codec Platform to Enhance Data Channels in FPGA Streaming Systems

3.1 Introduction

This chapter describes a framework to add compression by using extra CPU cycles on the software side and extra FPGA resources on the hardware side of the channel in heterogeneous streaming applications. Adding compression in this manner can mitigate the bandwidth bottleneck by increasing the effective bandwidth of the channel between the CPU and FPGA. This chapter demonstrates how multi-threaded compression on the software side and pipelined codecs on the hardware side can be used to apply line-rate compression to data that is being transmitted across the channel.

The framework is applied to a Maxeler system, a commercial high-level synthesis (HLS) platform for building heterogeneous streaming applications. This is an appropriate platform since large amounts of data are often transmitted between the CPU and FPGA on applications typically built using this platform. It is also a good example of an HLS tool, since such tools are often used to design hardware accelerators. Such applications have traditionally been built at Register Transfer Level (RTL), but the time taken to use RTL to develop pipelines and coordinate data flow is becoming prohibitive due to the increasing complexity of FPGA architectures. HPC programmers (kernel designers)

now use high-level synthesis (HLS) tools to handle low-level aspects such as instantiating and re-timing pipelines in hardware kernels. In such tools, the high-level compiler will generate RTL code based on the high-level code written by the programmer. HLS tools, while increasing the productivity of the programmer who uses them, have the drawback of limiting the user of the tool to using the high-level language and library primitives provided by the tool. The kernel designer is restricted to using the language and library primitives provided by the tool to describe logic, arithmetic and stream control. The primitives provided by the tools are not always sufficient to resolve challenges that modern streaming systems face. The kernel designer must revert to an RTL description in order to implement additional features that are not included in the basic streaming services provided by the tool.

Since HLS tools cannot easily address the bandwidth issue, this chapter describes a framework that aims to tackle the issues that are currently faced by systems built by such tools.

Hence, the contributions of this chapter are:

- A multi-threaded framework to apply compute-intensive codecs to a segmented data stream by using extra resources in software and hardware for high-bandwidth applications.
- A concrete implementation of the framework in a commercial platform using C code on the CPU side and OpenSPL code on the FPGA side.
- A model for predicting the end-to-end transmission latency when the framework is applied to a streaming system.
- Evaluation of data transmission performance across a bandwidth limited PCI-express channel on a heterogeneous system with 4 compression codecs showing up to 1.58x speedup for lossless compression (an increase in the total I/O bandwidth of the channel from 2690MB/s to 4256MB/s) and 1.85x speedup for lossy compression (an increase in the total I/O bandwidth from 2690MB/s to 4973MB/s).
- A comparison of measured values with model-predicted performance for different codecs.

Section 3.2 provides a detailed description of the framework proposed in this chapter. Section 3.3 gives specifications of the codecs that are used in the framework. Section 3.4 discusses the technical details of how the system is built. Section 3.5 describes a mathematical model to predict the performance of streaming systems operating under this framework. Section 3.6 evaluates the performance of the framework on a commercial platform when different codecs were applied to streaming systems.

3.2 Proposed Framework

As discussed in the Background chapter, various work has been done by other researchers to tackle the bandwidth bottleneck on streaming systems such as FPGA2 [33], which aims to implement a general framework, though it is not suitable for line-rate, heterogeneous CPU-FPGA systems. Parallel compression has also been implemented previously as discussed in the background section, but not for heterogeneous applications that involve FPGAs.

While compression methods and other enhancements exist, they are difficult to apply to a heterogeneous streaming system at in line rate, as discussed earlier. It would be useful to bring the benefits of these enhancements to the user without adding a significant amount of complexity. Therefore, a new general framework which supports such codecs is required, which will:

- Allow the HPC programmer to apply software and hardware codecs to the data in a streaming system without major changes to HW and SW.
- Make use of unused/extra resources in software and hardware.
- Enable other independent programmers to incorporate new codecs into the framework through its extensible nature.

The nature of the system which the framework is applied to is also important. The framework makes the following assumptions about the characteristics of the system:

- The input (writing) and output (reading) of the data channel in the system are decoupled and independent of each other.

TABLE 3.1: Variables for Equation 3.1

Name	Unit	Description	Typical value
f_k	MHz	kernel clock frequency	200
w_k	bits	kernel bus width	64
c	-	compression ratio of segmented compression	0.6
r_{ch}	s/MB	line rate or channel data rate	0.0005
$1/r_{ch}$	MB/s	channel bandwidth	2000

- The data channel of the system has a natural granularity.
- The system is tolerant to latency.

These assumptions are reasonable since there are many such data channels in real-life applications which have these characteristics, such as the Maxeler Low Latency interface, Maxeler MaxRing, RAM input/output, Avalon and AXI. Such properties of data channels are also common in networking which is why the analogy to networking has been used.

The framework enables the HPC programmer to add codecs by using idle CPU cycles in software and extra FPGA resources in hardware. The data to be transmitted is split into segments for multiple threads to process different segments and each segment is processed by a software pipeline of codec threads: compression; kernel transmission and decompression. This allows the interleaving of compression, transmission and decompression on the CPU, and it also allows the use of multiple compression threads to speed up software compression. Codecs must operate faster than the channel data rate in order for compression to achieve a speedup. The codec computations should be performed at a speed greater than or equal to the kernel/software computation and existing data transmission (i.e. line rate). Otherwise there will be a loss in computation speed. If c is the compression ratio, f_k the kernel frequency, w_k the kernel bus width and r_{ch} the data rate of the PCIe bus via which data is supplied to the kernel, the following condition must be satisfied in order for the channel not to be a bottleneck (Table 3.1 defines the variables):

$$f_k \times w_k \leq \frac{1}{r_{ch}c} \quad (3.1)$$

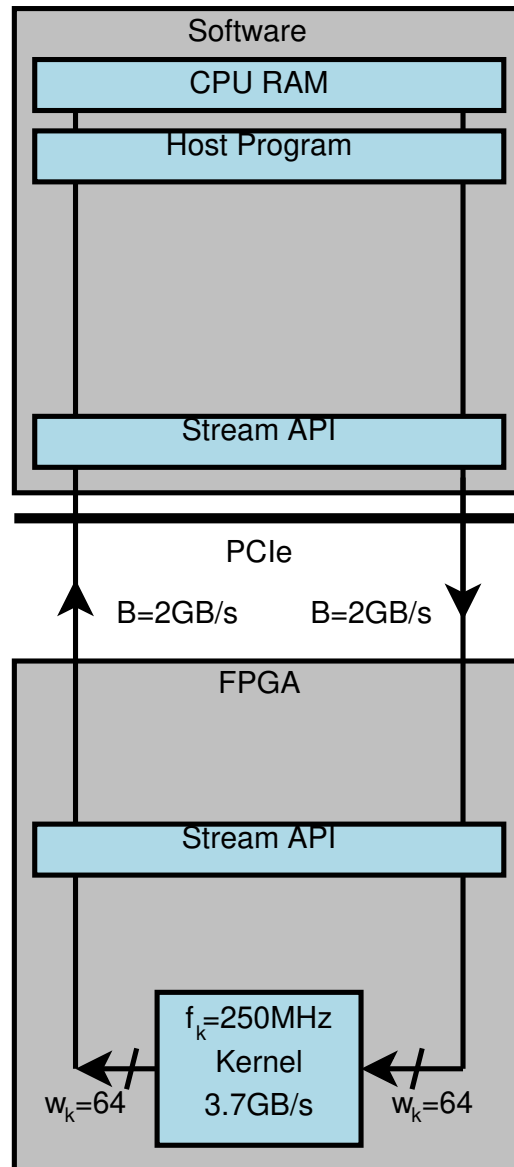


FIGURE 3.1: Block diagram of the original system before a codec is applied.

The framework is not expected to be suited to every single possible application – it will be more suitable for applications which involve transmission of large amounts of data – where the latency of the pipeline is negligible compared to the end-to-end transmission time.

Fig. 3.1 shows a block diagram of what a heterogeneous streaming system looks like without any codecs. Fig. 3.2 shows what the modified system would look like when compression and decompression are applied in both software and hardware. Fig. 3.3 shows what the modified system would look like when multiple threads in software are used when applying compression and decompression.

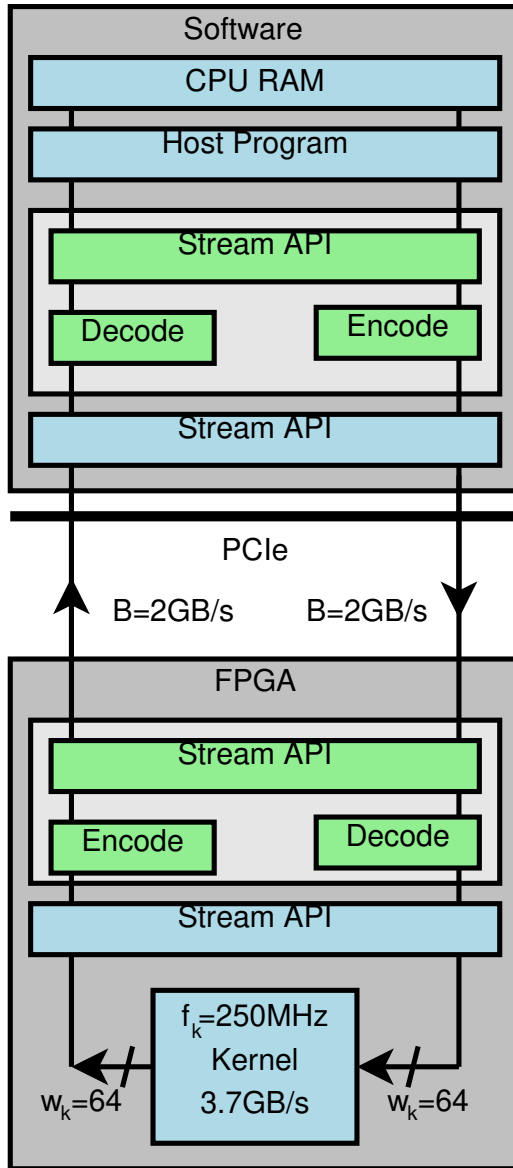


FIGURE 3.2: Block diagram of the system when a codec is applied. The green blocks represent the spare resources used by the codec and the blue blocks represent existing in-use resources.

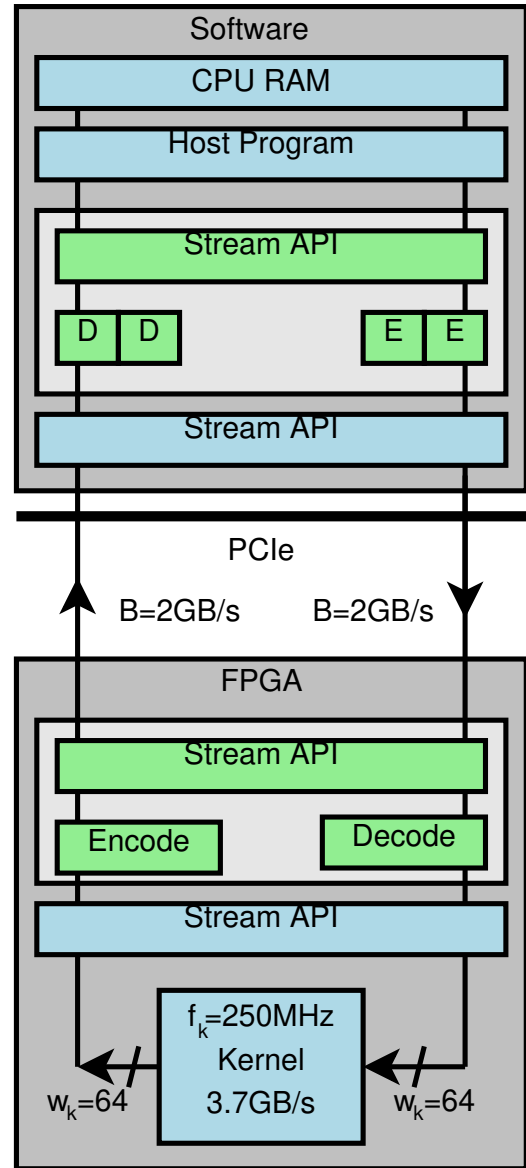


FIGURE 3.3: Block diagram of the system when a codec is applied with multiple instances of the codec in software.

3.3 Codecs

To demonstrate the framework, a set of codecs are applied to the data in a streaming system, leaving the original software and hardware intact, and performance is evaluated. When different codecs performing various transforms are designed to be added to the data transmission interface, all the codecs, both on the hardware and software side must have some common properties.

The codecs on the hardware side must meet certain conditions:

- They must be pipelined in hardware to operate at line rate.
- They must only occupy a small amount of FPGA resources.
- They must have a streaming input and output.

The codecs on the software side must also meet certain conditions:

- They must be multi-threaded to process segmented data at line rate.
- They must be able to push and pop data segments to and from a queue.
- They be fast enough to operate at line rate.

It is not expected that the compression codecs will achieve the best possible compression ratio, but rather it would demonstrate that it is possible to achieve a noticeable improvement in speed or to add other features while minimising deterioration in performance. It should be possible to apply a range of different compression methods, so a number of special purpose codecs are used. The methods to be tested are:

- Integer delta compression – where differences between adjacent values in data are computed and represented using fewer bytes.
- Lossless floating point compression – where zeroes in sparse floating point data are packed into a single byte.
- Lossy double precision – where 64-bit floating point numbers are converted to 32-bit floating point numbers.
- RGB32 to RGB24 conversion – where the alpha channel in RGB data is omitted.

The reader will note that these algorithms appear to be very straightforward. While there are plenty of existing sophisticated compression algorithms, it is not common for them to work efficiently in both software and hardware, and it is not common for the implementations to be fully compatible with each other. Codecs such as Huffman Coding were considered. Maxeler has strict requirements on granularity when transmitting data

across the channel, and Huffman Coding will likely result in data sizes that do not fit into the correct granularity. While the codecs used in this chapter are not complex in terms of their operation, and they were intentionally chosen to allow the above conditions to be met. The simplicity of these algorithms allows them to operate at line rate in software and hardware, without occupying a large amount of chip area. Additionally, they do not have extreme best-case or worst-case compression ratios. This is particularly important, since compression that has very bad worst case ratios can have negative effects on performance, even if their average compression ratio appears to be reasonable, as shown by the implementation of bitstream compression discussed in [54]. Having a small resource usage is also useful if it is necessary to replicate the compression codecs to fill larger amounts of chip area in highly I/O bound applications (though this is not investigated in this report).

These codecs are not an exhaustive list of what would be applicable through the using the conceptual framework that is proposed. These are special-purpose codecs, and general-purpose codecs such as run-length encoding could also be applied, though it has not been tested. This has been discussed in the future work section.

3.3.1 Delta Compression

This delta compression codec compresses 64-bit integers and it is designed with the granularity of PCIe in mind. Initially, it calculates the differences (delta values) between adjacent values in the data to be transmitted. The delta values are then checked to see whether they are small enough to be represented using fewer bits. If two adjacent 64-bit delta values are sufficiently small, they are both reduced to two 30-bit numbers and they are stored in a 64-bit packet. This gives a best-case compression ratio of 0.5. There is a single bit at the start of the packet to indicate whether or not it is compressed. If the delta value is not small enough to be compressed, the number is transmitted as 64-bit packet and another 64-bit packet is created which stores metadata to indicate that the data is not compressed. So this will occupy 128 bits of data in total and it gives a worst-case compression ratio of 2. The hardware bus width is 64 bits, and the compiler automatically inserts buffers to handle the discrepancy with the native PCIe bus width of 128 bits. Delta compression is suitable for smooth data.

3.3.2 Lossless Floating Point Compression

This codec compresses single precision floating point numbers. It works by initially checking the first byte in each element in the block of data for zero values, and then representing them using a single zero byte instead of the original 4 bytes. If the number is non-zero, the original number is kept. At the start of the block of data, there will be a 64-bit header indicating the number of floating point values in the encoded block of data. Packetisation is not manually done for this codec, but it is automatically handled by the granularity of PCIe. This is almost lossless with the exception that denormal numbers are represented as zeroes by this compression algorithm. MaxCompiler does not support denormal numbers, so removing denormal numbers and replacing them with zero will not cause problems. In the event that data is not compressible, expansion happens as a result of the metadata that is present in the stream. The best case compression ratio of approximately 0.25 will occur if all the numbers are zero, such that each of them will be reduced to a single byte. This assumes that the size of the block of data being compressed is significantly larger than the 64-bit header. The worst case compression ratio depends again on the size of the data block being encoded. If the block of data is much larger than the header, then the compression ratio will be slightly above 1, due to the presence of the header. When decompressing, a byte of data is checked to see whether or not it is zero. If it is zero, it will be expanded to a 32-bit floating point zero. If the byte is non-zero, then 4 bytes will be read from the encoded data.

3.3.3 Lossy Double Precision

Lossy double precision converts from IEEE 64-bit floating point to IEEE 32-bit floating point, hence a fixed compression ratio of 0.5. The mantissa gets truncated so precision is lost. In the event that there are not enough bits in the exponent to represent the number when it is converted (i.e. overflow), the floating point number is set to infinity. This is a major trade-off of precision for performance. A loss of precision such as this could be acceptable in neural network applications or some Monte Carlo simulations. It could also be suitable for signal processing applications such as FIR filters or Gaussian blur. When converting from double precision to single precision, another issue that arises is the reduced dynamic range. Lossy compression would therefore not be suited for applications that exceed the dynamic range of single precision floating point. The software raises an

exception where the exponent is too large. The hardware can be designed to raise an exception too, though it was not done in the particular implementation used in this chapter (a commercial version will need to have this).

There is the counter-argument, that the entire application should then be implemented in single precision floating point, and there is no point converting the single precision back to double after it has been transmitted. There would indeed be nothing wrong with doing that and this is something that was considered. However, the work in this chapter is based on the idea that the original computation in the kernel is not modified – it may well be modified in other cases, if it proves to be a more sensible solution than compression. For example, if the programmer wants to test if their kernel produces accurate results at all using single precision data, it may be worth using this framework, instead of rewriting the kernel to test it.

3.3.4 RGB32 to RGB24 Conversion

RGB32-RGB24 omits the alpha channel in image processing applications, hence it gives a compression ratio of 0.67. Pixels that would normally be represented using 32 bits will be represented using 24 bits. The 24-bit data is placed contiguously in memory so that it can be sent as a single stream. When the hardware has to receive data where the bus width is a multiple of 3, MaxCompiler automatically inserts buffers to handle this since this is not a multiple of the native width of PCIe. When data is decompressed, the alpha values are recreated, the data bus is widened and the extra bits (of the alpha channel) are set to zero.

3.4 Implementation

The framework is built using a commercial HLS tool called MaxCompiler [2] by Maxeler Technologies. The Maxeler machine has 12 CPU cores and a Xilinx Virtex 6 FPGA which are interfaced together via a PCI-express channel. This machine is well suited to demonstrate our proposal since large amounts of data have to be transmitted via a PCI-express channel. It includes high-level libraries for building commercial heterogeneous HPC applications and allows high-speed streaming of data across the channel. This platform is a suitable match for the requirements of this investigation.

Maxeler offers two modes of data transmission. The conventional and more commonly used mode requires a hardware interrupt to be generated for each data transfer, where the application is suspended until the interrupt is dealt with by the operating system. While this is suitable for transmitting large contiguous blocks of data, it is not suitable for our application, where the data is split into multiple smaller segments since there will be a significant latency added to each segment by the interrupts. The second mode is the ‘Low latency’ [3] interface, which is optimised for short and frequent data transfers – the FPGA kernel is permanently running and data segments can be transmitted to/from the kernel as and when it is required. The second mode is ideal for the segmented data that is used here.

The additional services such as compression are added to the software by making use of idle CPU cycles through the use of multi-threading and to the hardware by making use of unused FPGA resources.

While we implement this system on the Maxeler platform, the framework proposed here does not have to be restricted to Maxeler – there is no reason why this cannot be used on a different bandwidth bottlenecked heterogeneous platform or a system built using a different HLS tool. The expectation is that the framework can provide a benefit in any bandwidth bottlenecked heterogeneous system which involves communication between different physical devices. This is explored in more detail in Chapter 5.

Fig. 3.4 shows how multiple threads access data. In software: encode thread(s) compress and push segments into an input queue; the transmit thread pops data from the input queue, transmits data to the FPGA and receives processed segments from the FPGA and pushes it onto an output queue; the decode thread(s) pop segments from the output queue and perform decompression. Segmenting the data is what enables the framework to perform encoding, transmission and decoding simultaneously by using separate threads for each of the three actions with the option of using different amounts of threads to perform encoding and decoding.

Codecs that are being added to hardware should be added at the points between the kernel and where it communicates with the outside world. Software codecs should be added into the program before the point where data is transmitted into the kernel (done by invoking the compression threads). The compression is applied to the data array from which the data would be transmitted to the kernel and then the compressed data

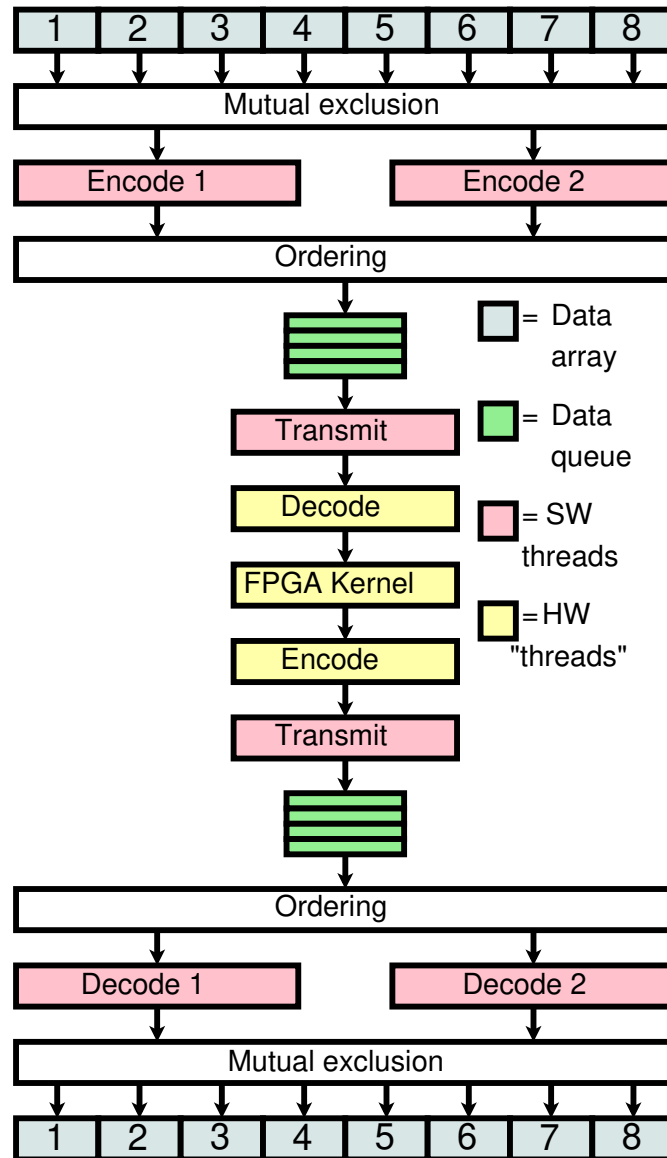


FIGURE 3.4: Diagram showing how queues are managed by the threads.

is transmitted. Similarly, the data that the software program receives from the hardware will be decompressed into the original data array where the data would have gone.

When a segment needs to be compressed, any idle encode threads will compete to lock onto that segment, and similarly, any idle decode threads will compete for segments of data to decompress. If the input queue of data being transmitted to the hardware is full, the encode threads must wait until space is available; when space becomes available, they will proceed to compress the next segment and push it into the queue.

The software transmit thread continuously polls the data channel to check if the hardware pipeline has space available at the input and if it has data available at the output.

If there is space available at the input, the transmit thread will pop data from the input queue and send it to the hardware. If there is data available at the output, the transmit thread will request this data from the hardware and push the data into the output queue when the data arrives. This results in the transmit thread being constantly active even if there is no data to transmit, which is not ideal. This is the mechanism by which the Maxeler Low Latency interface works. The conventional interface of Maxeler uses hardware interrupts to handle data transfers and it informs the software program when data is available at but output. The Low Latency interface does not support hardware interrupts, so it is necessary to keep polling the channel. In many existing bandwidth bottlenecked applications, it would be the case that a token of data is not necessarily processed on every cycle. Such applications are where data is transmitted in bursts instead of being continuously transmitted. The constant polling of the channel would be a disadvantage since it would increase power consumption while not actually transmitting data.

Decode threads will wait for compressed data to appear on the output data queue if it happens to be empty; they will compete to lock onto the next segment that becomes available and it will be popped from the queue to decompress it. When the software is properly operating at line rate, one encode and one decode thread should always be idle and waiting for new data to be available, in the ideal case.

In hardware: pipelined decode and encode modules are to be inserted into the pipeline before and after the computation kernel respectively. Data passes through the decoder module at the start of the pipeline, into the kernel, then into the encoder module towards the end of the pipeline. The hardware codecs must have a throughput greater than or equal to the throughput of the existing bottleneck (in our case, this is the bandwidth).

Since MaxCompiler is an HLS tool, it automatically handles some of the data flow through the pipeline, while giving the programmer access to certain high-level primitives to manually control the flow of data including backpressure and data bus width. The PCIe channel has a native width of 128 bits, and the compiler will insert FIFO buffers to handle any change in bus width between the external interface of the FPGA and between FPGA kernels. Also, when inserting codecs into the pipeline on the hardware side, MaxCompiler has the capability to automatically handle backpressuring between chained kernels. While this can simplify some implementations, this can reduce the

amount of flexibility available to implementations that require more control. Kernels that perform compression involve unpredictable changes in data flow due to contraction and expansion, and this requires manual control of the data flow. The Manager State Machines provided by MaxCompiler allow the programmer some degree of manual control over data flow, and these were used to manually switch the flow of data on and off. The later versions of MaxCompiler provide additional primitives that give programmer more control over backpressuring on individual cycles, but the work in this chapter predates this.

A challenge that the implementation of this framework would face is that the existing FPGA kernel does not know the increased length of the hardware pipeline as a result of the codecs – this can reduce the benefit of prefetching if the prefetch request is not completed in time. While the framework aims to be transparent to the existing streaming application, the kernel will need to be aware of any additional delay that would be incurred. This is an issue that requires further investigation, and the automatic prefetch microarchitecture proposed in [62] would be a useful solution, though it is not used by the work in this thesis. The techniques in the prefetch microarchitecture [62] automatically increases or decreases the prefetch lookahead distance based on scratchpad performance, such that it will be able to handle increased latency. So this prefetching technique would be tolerant to the increased latency of the framework.

Another challenge that this implementation would face is if an existing application had a tight closed loop with rapid feedback, it will not work well with this framework, since the kernel may spend too long waiting for data to propagate through the loop. However, this is not the intended use case.

Fig. 3.5 shows the design flow of the HPC programmer when applying the framework to a streaming application:

- Initially, the programmer needs to establish whether or not the application is bandwidth bottlenecked, by seeing if large amounts of data are being transmitted and whether a significant proportion of the total run time of the program is spent transmitting data.
- Next, the programmer needs to check the type and nature of the data being transmitted to see if it would be compressible. This includes having some knowledge

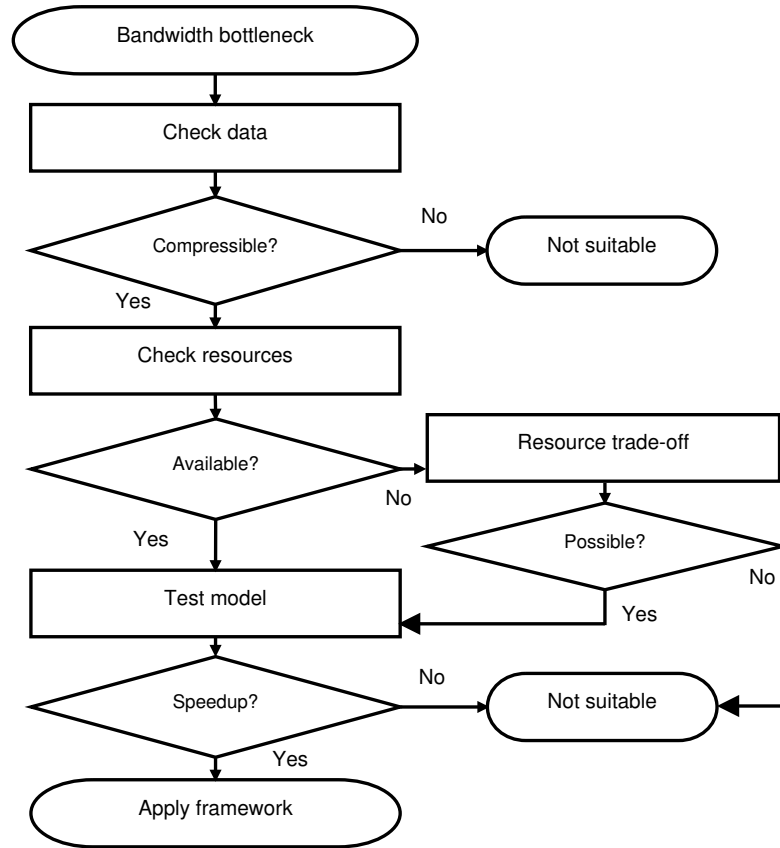


FIGURE 3.5: The design flow when codecs are applied using the framework.

of the source of the data, to establish whether or not the data is highly entropic. If the data is likely to be smooth, then an algorithm such as delta compression would be suitable. In floating point data, if the data is likely to be sparse, then the lossless floating point algorithm would be appropriate. If lossy compression is to be used, caution must be exercised. For example, RGB32-RGB24 conversion should not be used in an application where the alpha values are necessary and lossy floating point compression should not be used if double precision data is absolutely necessary.

- Then the existing resource usage needs to be assessed on both the software and the hardware sides. On the software size, the CPU is likely to have some idle cycles if the entire system is awaiting data transmission. The hardware side should be checked to see if spare resources are available. Even if spare resources are not available, the FPGA kernel may not be able to achieve its full potential computational capability if the channel cannot supply data at a sufficient speed. This would mean balancing the resource usage between the existing computation kernel and the framework to apply compression.

- Afterwards, the model should be applied to see if there can be an improvement in performance from the framework. If yes, then the programmer can go ahead and apply compression using the framework.

It is also important to note that the implementations of these codecs do not use SIMD extensions when performing compression, and this decision was taken due to a number of reasons. Currently, when the parallelisation is performed across segments, the result of each segment being compressed is placed in a different destination address. If compression is to be vectorised across segments, the packing of data in memory will have to be changed. Adjacent elements in the data stream will no longer be adjacent if this happens, but compression needs to have access to adjacent data to identify patterns. Another reason is the difficulty of handling unknown expansion and contraction of data – if the expansion or contraction was the same on all the elements which a SIMD operation is performed, this would be acceptable, but this is unlikely to be the case. This is likely to result in memory alignment issues. In order to prevent such issues, along with maintaining the capability of the codecs to be portable across different processors, the decision was taken not to use SIMD extensions. One could argue that RGB32-RGB24 conversion could be done using SIMD since there is an instruction available to perform the conversion. However, we felt that it is necessary to maintain portability between different platforms, and using SIMD would reduce this portability, and will require different implementations for different software platforms. Not using SIMD extensions is likely to increase the load on the CPU, but during testing, it was observed that not all the threads are active at the same time when the software compression runs at line rate.

For the purpose of evaluating performance, an identity kernel clocked at 200MHz was taken to be the existing hardware streaming kernel for most cases. When codecs are applied, the assumption is that there is a high-speed computation kernel in the FPGA, where the bandwidth of the channel supplying data to and from the kernel is a bottleneck. The exact computation that takes place in the kernel is not particularly relevant and so it is treated as a black box. An identity kernel is appropriate since it meets the requirement of transmitting data at a high throughput and 200MHz is a typical clock frequency that FPGA kernels operate at in HPC applications. We also apply the concept we propose to an existing application; namely Fast Fourier Transform (FFT), operating at 200MHz. Open source implementations of Fast Fourier Transform (FFT) are provided by Maxeler in the Maxeler AppGallery [63]. FFT is computationally intensive and therefore an

TABLE 3.2: Parameters for model and their typical values

Name	Unit	Description	Typical value
S	MB	size of entire data	2048
s_s	MB	size of segment	8
t_k	s	time overhead of initiating kernel transmit	0.0003
t_{th}	s	time overhead of thread start	0.0002
c	-	compression ratio of segmented compression	0.6
a	-	CPU resource contention exponent	0.82
r_{ch}	s/MB	line rate or channel data rate	0.0005
r_e	s/MB	encode rate in a single thread	0.002
r_d	s/MB	decode rate in a single thread	0.002
n_e	-	number of encode threads	2
n_d	-	number of decode threads	2
r_e/n_e^a	s/MB	encode rate for multiple threads	0.001
r_d/n_d^a	s/MB	decode rate for multiple threads	0.001
S/s_s	-	number of segments	256

appropriate algorithm to be tested in an FPGA streaming system. The codecs are inserted into the software at the points where data is sent and received, and they are similarly inserted into the hardware at the same respective points. It is essential that the framework is tested on a real streaming application that has been implemented by other programmers – since the codecs must be able to function on streaming systems that have not been designed by the programmer who designed the codec.

3.5 Mathematical Model

Now that the specifications of the framework have been defined, a model is developed for how the performance of a streaming system will be affected by applying compression using this framework, and this model assumes that there are an infinite number of CPU cores available.

Table 3.2 defines the parameters that are used in the model. The size of entire data and segments are user-selected. The time overheads of initiating kernel transmit and thread start were measured empirically. The compression ratio is determined from the data that is used. The CPU resource contention exponent (due to multiple threads) was empirically determined to be 0.82 by measuring encode time as the number of threads were varied. The model does not place a cap on the number of CPU cores that are

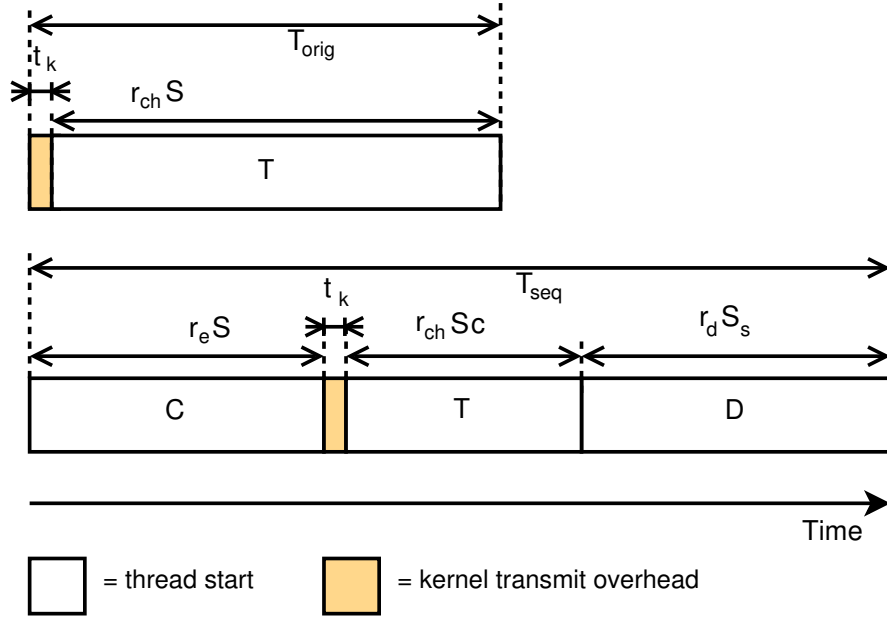


FIGURE 3.6: Conceptual timing diagram showing performance of original implementation and the performance of sequential compression, without using the framework.

available. The line rate, encode rate and decode rate were empirically measured. The number of encode and decode threads are selected by the user.

Figs. 3.7-3.10 show the interleaving of compression (C), transmission (T) and decompression (D) by different threads. The timing diagrams are conceptual since the exact sequence of events in the multi-threaded framework is non-deterministic. It shows how 8 segments of data could possibly be handled by the framework, in the case of compression being applied – while the diagram shows 8 segments, in reality, the stream of data may be infinitely long and could function in a steady-state manner. The model assumes that input and out data transmissions happen simultaneously.

1. Blue sections show the overhead in launching each software thread, so that the starting point of subsequent threads have a cumulative delay.
2. The pink sections show:
 - (a) Compression threads being blocked until space is available in the input queue (a maximum queue length of 3 is chosen for this example).
 - (b) The transmission thread being locked until there is data present in the input queue.
 - (c) Decompression threads blocked until data is available in the output queue.

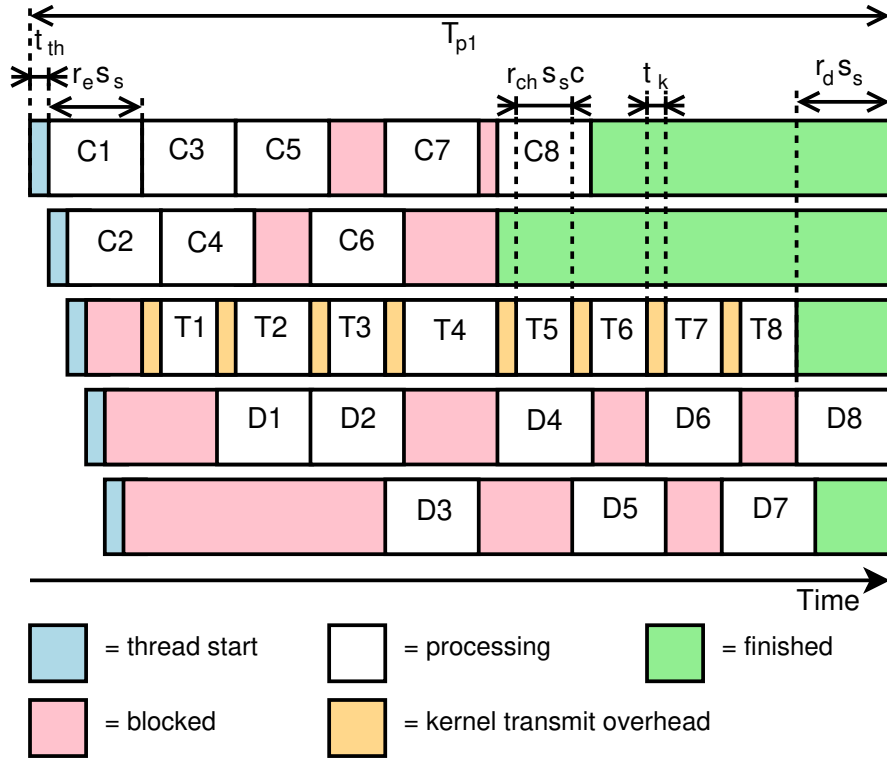


FIGURE 3.7: Conceptual timing diagram showing interleaved compression, kernel computation and decompression with a saturated kernel operating at maximum capacity. Number of encode threads, $n_e = 2$, $n_d = 2$, $r_e/n_e^a = 0.001s/MB$, $r_d/n_d^a = 0.001s/MB$

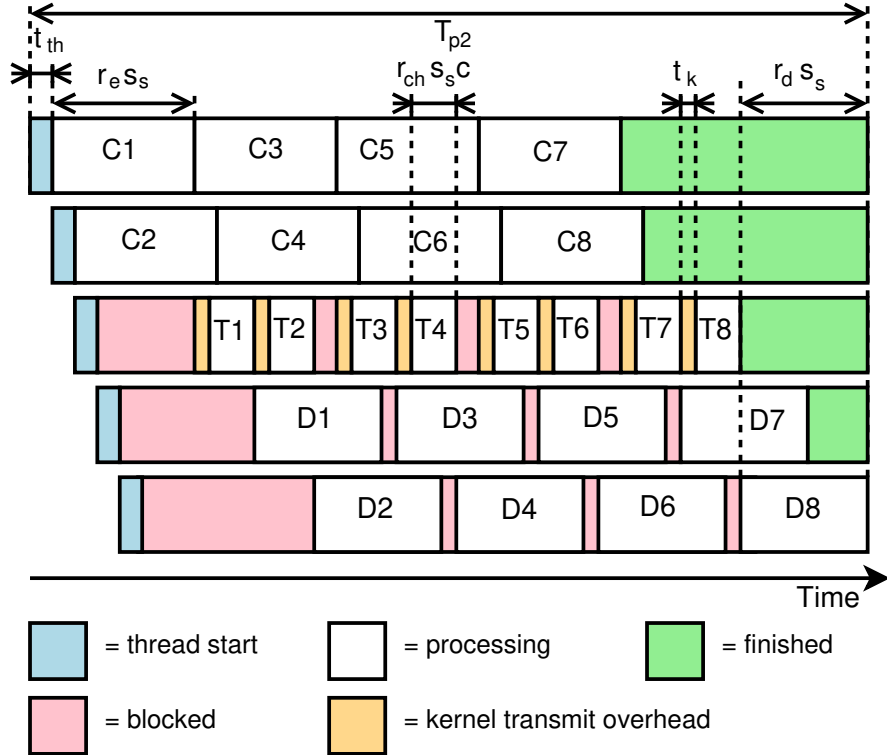


FIGURE 3.8: Conceptual timing diagram showing interleaved compression, kernel computation and decompression where the compression is too slow to saturate the kernel. $n_e = 2$, $n_d = 2$, $r_e/n_e^a = 0.002s/MB$, $r_d/n_d^a = 0.001s/MB$

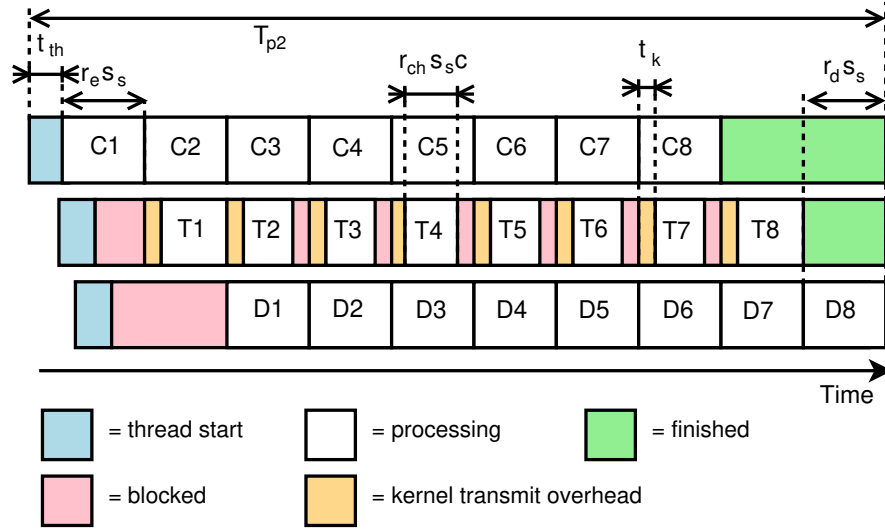


FIGURE 3.9: Conceptual timing diagram showing interleaved compression, kernel computation and decompression where both the compression and decompression are too slow to saturate the kernel. $n_e = 1$, $n_d = 1$, $r_e = 0.002s/MB$, $r_d = 0.002s/MB$

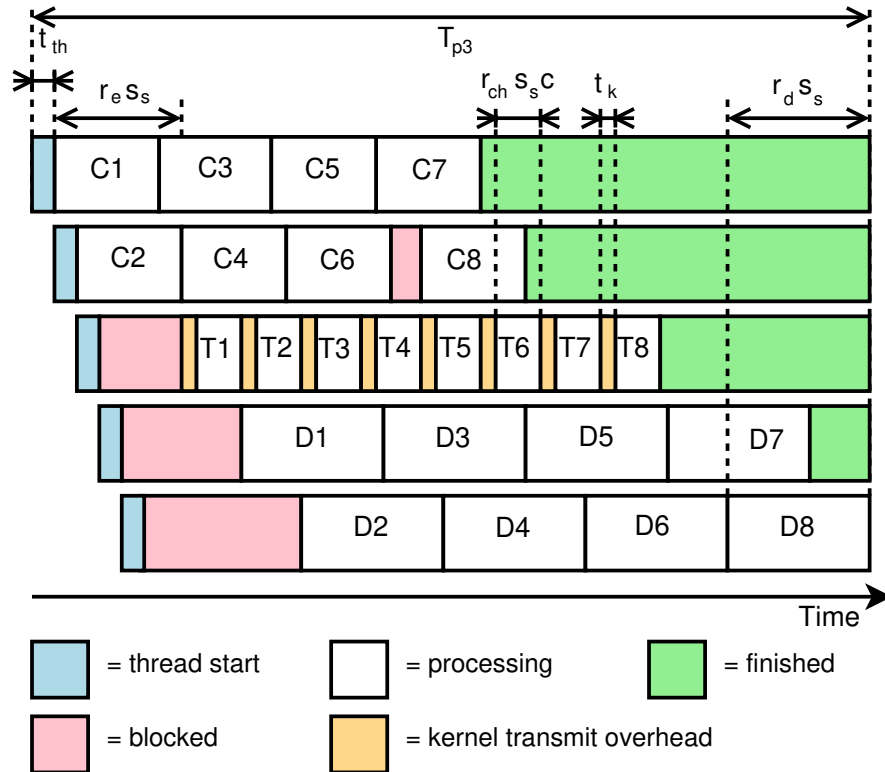


FIGURE 3.10: Conceptual timing diagram showing interleaved compression, kernel computation and decompression where the decompression is too slow to saturate the kernel. $n_e = 2$, $n_d = 2$, $r_e/n_e^a = 0.001s/MB$, $r_d/n_d^a = 0.002s/MB$

3. Green sections show threads being idle after completing their task.
4. White sections show processing taking place. In the transmission thread, the transmission time varies for each segment depending on its compression ratio.

The symbols and parameters used in the following equations are defined in Table 3.2.

Fig. 3.6 (top) shows the timing diagram of the end-to-end latency for original, unsegmented, unencoded data including the overheads. Its equation is:

$$T_{orig} = r_{ch}S + t_k \quad (3.2)$$

If the entire block of data is sequentially encoded, transmitted and decoded, the end-to-end latency is and its timing diagram is shown in Fig. 3.6 (bottom):

$$T_{seq} = S(r_e + r_{ch}c + r_d) + t_k \quad (3.3)$$

When threaded software pipelining is applied, there are 3 possible cases. The ideal case is illustrated by Fig. 3.7, where there are as many encode/decode software threads as necessary for software codecs to work faster than the channel data rate such that the channel bandwidth is saturated:

$$\frac{r_e}{n_e}, \frac{r_d}{n_d} < r_{ch}$$

The end-to-end latency for this ideal case is:

$$T_{p1} = t_{th} + r_e s_s + (t_k + r_{ch} s_s c) \frac{S}{s_s} + r_d s_s \quad (3.4)$$

The second case, where encoding (compression) does not happen fast enough is illustrated in Fig. 3.8 and Fig. 3.9. This happens if the compression is slow either because of too few compression threads or due to CPU resource contention reducing the benefit of multi-threading. Then the overall end-to-end latency is:

$$T_{p2} = t_{th} + r_e s_s \frac{1}{n_e^a} \frac{S}{s_s} + n_e(t_k + r_{ch} s_s c) + r_d s_s \quad (3.5)$$

The third case, shown in Fig. 3.9 (which also represented the second case) and Fig. 3.10, is where decoding (decompression) is too slow:

$$T_{p3} = t_{th} + r_e s_s + (t_k + r_{ch} s_s c) n_d + r_d s_s \frac{1}{n_d^a} \frac{S}{s_s} \quad (3.6)$$

It be must noted that Fig. 3.9 illustrates a particular situation in which both the second and third cases apply where neither the compression nor the decompression are fast enough

Overall, the threaded, software-pipelined end-to-end latency is defined as:

$$T_{pipe} = \max(T_{p1}, T_{p2}, T_{p3}) \quad (3.7)$$

When compression is applied using the framework, the end-to-end threaded time must not exceed the time for sending uncompressed data:

$$T_{pipe} < T_{orig} \quad (3.8)$$

This framework is suited in applications where large amounts of data are transmitted to fully benefit from compression and ensure that the hardware pipeline latency is negligible compared to the end-to-end transmission latency.

Furthermore, while compression-based codecs are heavily discussed, designing a state-of-the-art compression algorithm is not our goal; compression is considered as a useful case study. The emphasis is on a framework with the ability to add different features/codecs to a streaming system with low effort from the programmer.

A multi-threaded framework to apply compute-intensive codecs to a segmented data stream by the use of spare resources and the specifications required of such a framework have been discussed. A model for predicting performance of streaming systems when codecs are applied without the need to do a full implementation and benchmark has

been developed. The next section discusses the performance of the framework under various conditions.

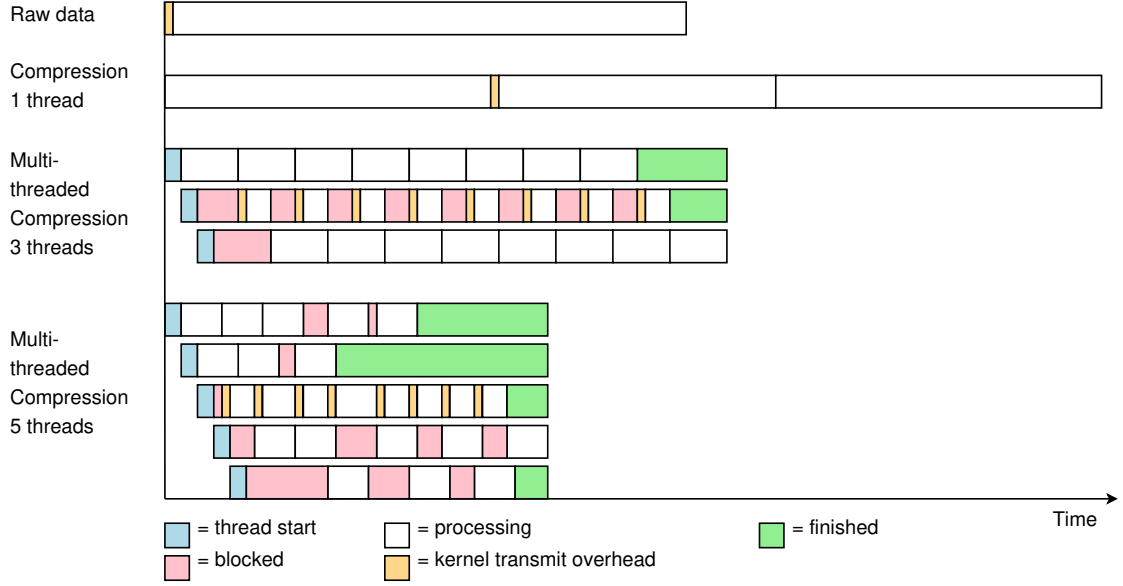


FIGURE 3.11: Combined timing diagrams showing relative performance of transmitting raw data, sequential compression (1 thread), and multi-threaded compression. The image aims to illustrate that with a sufficient number of compression and decompression threads, a speedup can be achieved. If the number of threads are insufficient, the performance will slow down rather than speed up.

Fig. 3.11 combines Fig. 3.6, Fig. 3.7 and Fig. 3.9 into one diagram, showing how long each will take relative to each other.

3.6 Evaluation

To evaluate the proposed framework, the performance when different codecs were applied to the transmission channel was measured. The experiments were performed on a Maxeler MPC-C500 machine. The machine has an Intel Xeon X5650 2.67GHz with 12 physical CPU cores and hyperthreading, and a Maxeler card running MaxelerOS 2015.2 with a Xilinx Virtex 6 FPGA. The metrics that are used to assess performance are end-to-end transmission time, effective bandwidth, speedup and compression ratio.

While different compression codecs are applied, the effect on performance of changing various parameters were also observed. The parameters to be varied are:

- Segment size.

- Number of threads.
- Total data size.

From a conceptual perspective, when the segment size is varied, the expectation is that there will be an optimum segment size where the performance will peak. If the segment size is too small, there will be a large number of segments, and the overhead of transmitting each individual segment should cause poor performance. Additionally, it will be more difficult for compression codecs to identify patterns in small segments compared to larger ones. If the segment size is very large, there is an expected slowdown in performance since there will be only a small number of segments, so potential parallelism is reduced, and the transmission thread will have to wait for a long time until the large segments are compressed.

As the number of threads are varied, it is expected that the performance will improve, until the number of threads approach the number of physical CPU cores available. It is expected that too large a number of threads will cause a slowdown in performance due to CPU resource contention.

Better improvements in performance are expected for larger total data sizes in a streaming system – there are some fixed overheads involved in transmitting data, so the best performance would be achieved if the majority of the time is spent on transmitting data rather than handling overheads.

The performance when varying different parameters is tested by using delta compression and lossless float compression throughout this section. Delta compression and lossless float compression are compute intensive if large amounts of data are to be compressed, so they are appropriate for testing the framework. To a lesser extent, the performance is also tested on an RGB32-24 image conversion algorithm and lossy double precision floating point to single precision floating point algorithm. These are already very fast algorithms since they are not compute intensive, but it is useful to see if a speedup can be achieved using an algorithm that is already very fast.

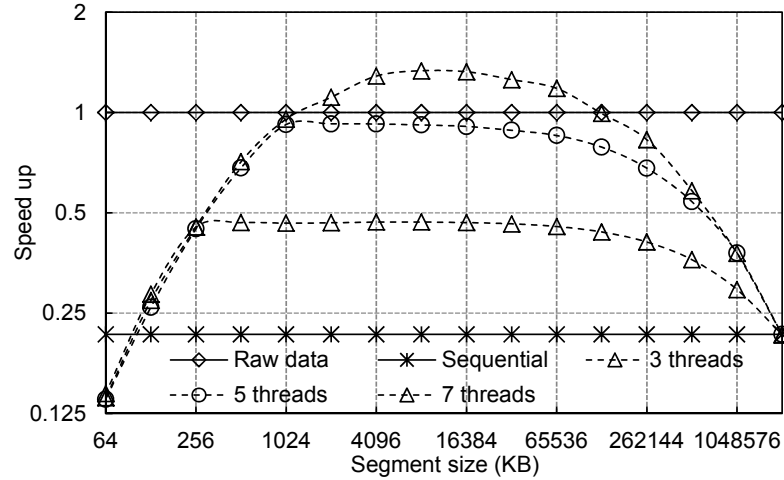


FIGURE 3.12: Speedup for 2GB data against segment size for lossless float compression. The peak performance of 1.33x was achieved with 7 threads. At 9 threads (not shown on graph), the performance was very slightly slower than with 7 threads, and it was not clearly distinguishable on the graph – hence the graph for 9 threads has not been included.

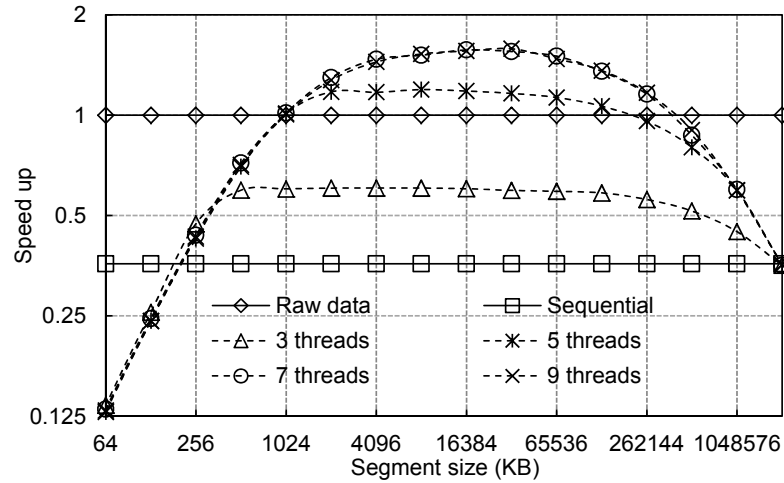


FIGURE 3.13: Speedup for 2GB data against segment size for delta compression. The peak performance of 1.58x was achieved with 9 threads. The performance for 9 threads was marginally higher than the performance for 7 threads.

3.6.1 Varying Segment Size with Number of Threads

Fig. 3.12 and Fig. 3.13 illustrates the speedup from different segment sizes and different threads for floating point and delta compression using a fixed total data size of 2GB. This is an appropriate size to use because the performance of the system is assessed under the assumption that it will be used in streaming applications where large amounts of data are transmitted. It shows a slowdown of an order of magnitude for unthreaded (sequential) compression. There is always 1 transmit thread and an equal number of encode/decode threads – the graph shows the total number of threads. The best speed

was achieved at 7 threads (3 encode, 3 decode, 1 transmit) at a segment size of 8192kB for float compression and 9 threads with 32768kB segments for delta compression. It must be noted that these optimum segment sizes are applicable for a total data size of 2GB, and for a different total data size, the optimum segment size would be different (this is discussed further ahead).

In each of Figs. 3.12 and 3.13, the number of threads are gradually increased until the performance stops increasing any further – for different codecs, this happens at a different number of threads.

It is observed that the speed generally increases with the number of threads. For 3 threads (1 encode, 1 decode, 1 transmit), the system performs slower than transmitting raw data, since the speed of compression and decompression in software is slower than the channel bandwidth. Using a larger number of threads results in a speedup as the multithreaded parallel compression and decompression is faster than the channel bandwidth. This can be compared with the timing diagrams presented earlier, where the performance of raw data, sequential compression, 3 threads and 5 threads were qualitatively considered – which graphically showed that 3 threads will be slower than transmitting raw data while 5 or more threads could result in a speedup. It is intuitive that when compressing and decompressing large amounts of data, parallelism using multiple threads will achieve faster compression. While not discussed in detail in this chapter, using multithreading with small total data sizes will not give any benefit, and it is discussed later that only data sizes above a certain threshold (depending on compression type) will see a speedup occur at all when compression is applied using our framework.

As mentioned earlier, these were measured on a machine with 12 Intel Xeon 2.67GHz physical CPU cores. Delta compression was also tested with 11 threads, and it was found to perform slightly worse than 9 threads – it was not included in the graph since the curves for the two will not be clearly legible due to overlap. This can be explained by resource contention as the number of threads approach the maximum number of physical CPU cores. For float compression, 9 threads performed slightly worse than 7 threads but this is not included in Fig. 3.12 due to the data almost overlapping and making the graph difficult to read. It is apparent that for all three of the figures, the

optimum number of threads does not appear to vary significantly as the segment size is varied.

The raw data measurement is the baseline, where data is transmitted without any compression or encoding. The sequential measurement involves encoding the entire block of data, sending/receiving it to/from the kernel, and decompressing the entire block of data. With regards to the general pattern observed for threaded data, the data shows that the performance is slow at both small and large segment sizes. Where smaller segment sizes are used, there will be a larger number of segments – the overhead of initiating multiple kernel transmissions becomes dominant and this causes the speed to slow down. Where larger segment sizes are used, such as the worst case scenario where the segment size is equal to the total data size, the system essentially becomes sequential. In this case, the transmission thread will have to wait until the entire block of data is compressed before it is transmitted and it will take a long time to decompress the block after transmission, therefore the speedup becomes equal to the sequential measurement. If the optimum segment size is used, it becomes a trade-off between the compression/decompression overhead and the multiple transmission overhead. This achieves the fastest performance. It may or may not saturate the bandwidth of the channel, depending on factors such as the compression speed and compression ratio.

Fig. 3.14 is included for the purpose of comparing the best performance of the two codecs represented by Figs. 3.12 and 3.13.

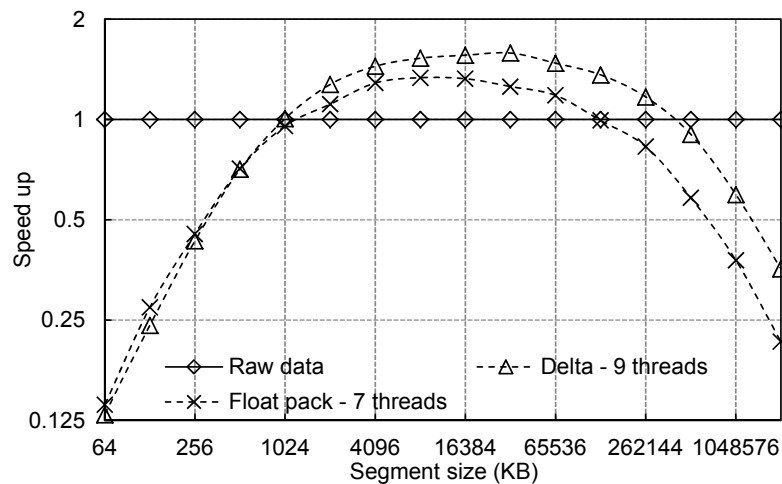


FIGURE 3.14: Comparison of the performance of both codecs under threaded conditions.

3.6.2 Varying Segment Size with Total Data Size

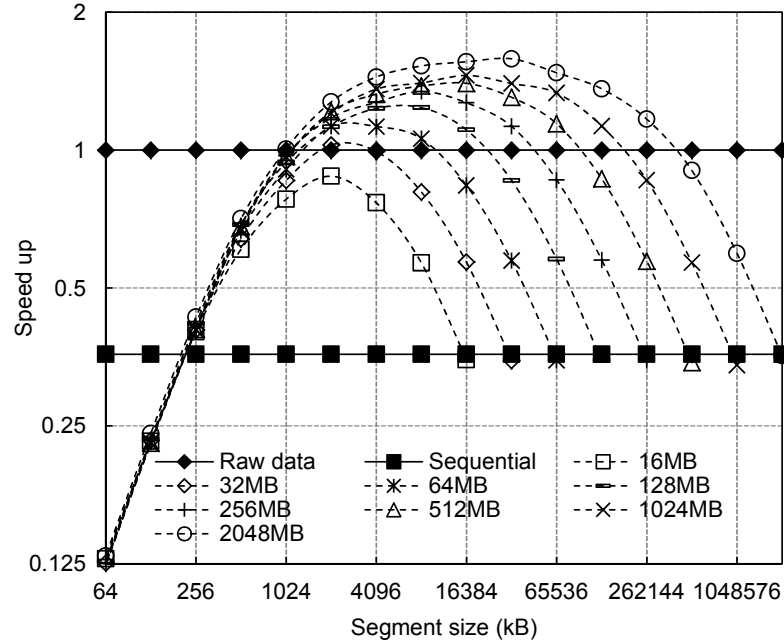


FIGURE 3.15: Comparison of speedup against segment size for different total data sizes when using delta compression.

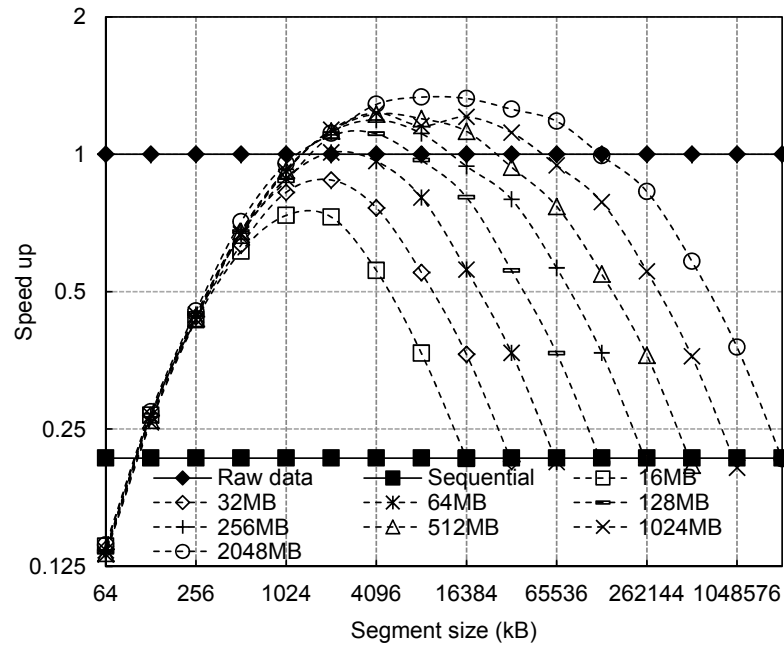


FIGURE 3.16: Comparison of speedup against segment size for different total data sizes when using float compression.

Fig. 3.15 and Fig. 3.16 illustrates the speedup from different segment sizes achieved when compression was applied to different total data sizes. The number of threads were fixed at 9 threads for delta compression and 7 threads for float compression, as these gave the peak performance for large total data sizes. It is evident from the graphs that

for larger total data sizes, the peak performance is also achieved at larger segment sizes, and the peak is higher overall for larger total data sizes. For small total data sizes, the framework is no longer suitable as the speedup is below 1.

3.6.3 Varying Total Data Size

Fig. 3.17 and Fig. 3.18 show how the speedup is affected by lossless single-precision floating point and delta compression in comparison to transmitting raw data for varying total data sizes. For lossless float compression in Fig. 3.17, ‘normal’ data is a set where there are some zero values and the random data is obtained from a random number generator (RNG). Similarly, for delta compression in Fig. 3.18, ‘normal’ data is smooth real-world data and the random data is from an RNG. Data from an RNG is used to show the worst case scenario, where the data is not compressible. For lossless float compression, the normal data and random data had 1.33x and 0.899x changes in speed for a data block of size 2GB. For delta compression, the normal data and random data had 1.58x and 0.888x changes in speed for a data block of size 2GB. If a compression method appropriate for the type of data is used, then this shows that a speedup can be achieved with low effort.

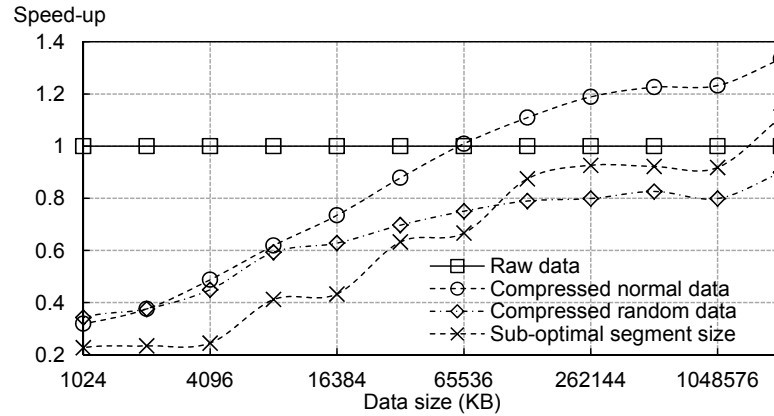


FIGURE 3.17: Speedup against total data size for lossless float compression.

The measurements were done with 9 threads and for delta compression and 7 threads for float compression as these were previously determined as having the fastest speed. For the normal data and random data, the optimum segment size for each total data size measurement was used to show the best possible performance for each total data size.

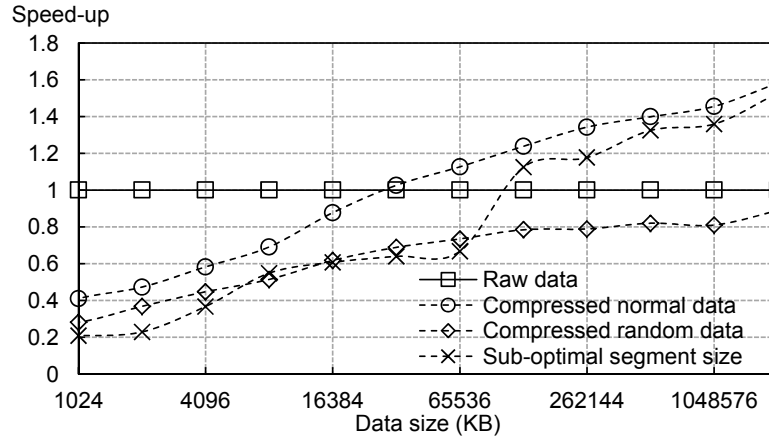


FIGURE 3.18: Speedup against total data size for integer delta compression.

The compressed normal data curves in Fig. 3.17 and Fig. 3.18 show that the general trend is that a higher speedup is achieved at larger total data sizes and a slowdown happens at smaller total data sizes, as it was shown previously. It clearly indicates the point at which compression performs faster than sending raw data. At smaller total data sizes, the overheads of compression, initiating multiple data transmissions and decompression become dominant compared to the time spent actually transmitting the data and therefore a slowdown is observed. This is despite using the optimum segment size for these particular total data sizes. It should be noted that for small data sizes, FPGA acceleration may not be necessary at all, since FPGAs in streaming applications are normally used where large amounts of data are processed.

The curves for the sub-optimal segment size uses a size that is a quarter of the ideal segment size for its respective total data sizes – the aim is to show how the performance will be affected if the segment size selected deviated from the ideal. The curves have a pattern that looks somewhat like a staircase. The flat sections of the curve shows the points where the segment size limits the speed because the size is too small. Although not included on the diagrams for tidiness, if a segment size that was slightly too large was used, a similar staircase pattern appears, where the speed is limited by the large segments. All 3 figures also show that even if the segment size deviates, a decent performance can still be achieved.

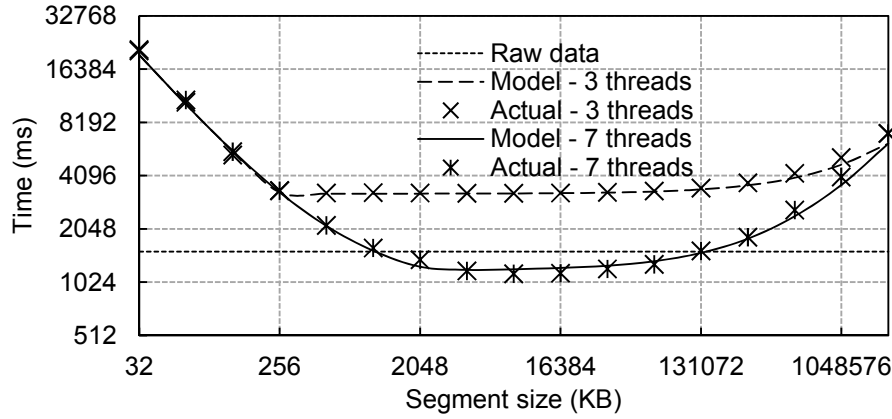


FIGURE 3.19: Modelled/actual time for lossless float compression with 3 and 7 threads. 7 threads are used here since it gave the best empirical performance.

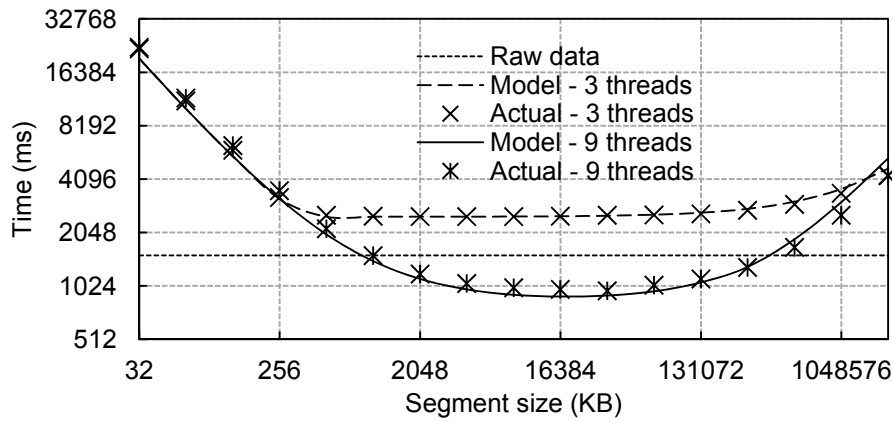


FIGURE 3.20: Modelled/actual time for delta compression with 3 and 9 threads. 9 threads are used here since it gave the best empirical performance.

3.6.4 Comparison with Model

Fig. 3.19 and Fig. 3.20 compares the measured end-to-end execution time against the time predicted by the mathematical model. The performance of three threads are included to show the result of minimal parallelism. Seven threads and nine threads are included for float compression and delta compression respectively since these were found to have the fastest empirical performance. For delta compression – the maximum discrepancy within the region of interest (where segment sizes are neither too large nor too small) is 8.70% and the maximum discrepancy overall is 25.4%, for a segment size of 2GB (which will not be used in reality). For float compression – the maximum discrepancy within the region of interest is 8.74% and the maximum discrepancy overall is 12.3%, for a segment size of 2GB.

The model does not consider the latency of the hardware pipeline in the FPGA. The

transmit overhead in software is around $200\text{-}300\mu\text{s}$; equivalent to $40000\text{-}60000$ clock cycles in the FPGA. Therefore the latency of hardware codecs are orders of magnitude lower than the software latency. The model is able to predict and evaluate the end-to-end latency of a system without the need to fully build and benchmark. The limitation is that the compression ratio needs to be known in order to make predictions with the model, which requires the user of the model to know the nature of the data that will be processed.

3.6.5 Miscellaneous

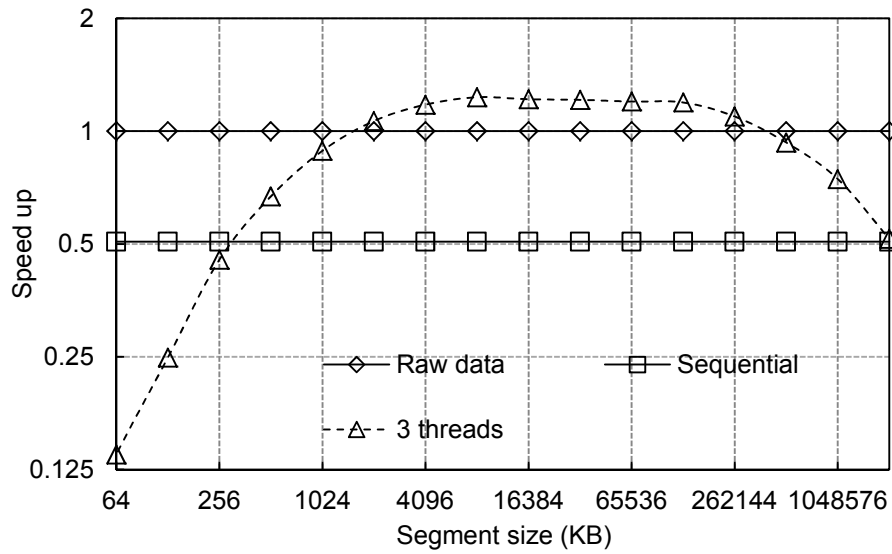


FIGURE 3.21: Speedup for 2GB data against segment size for RGB32-24 image conversion showing a peak of 1.23x

Fig. 3.21 and Fig 3.22 shows the performance of two very fast data conversion codecs, where the size of the data changes.

Fig. 3.21 shows the speedup for different segment sizes and threads for RGB32-24 conversion, for a total data size of 2GB. It must be noted that this is not a compression algorithm, but an image conversion to omit the alpha channel in applications where the alpha channel is not important. The framework is still applicable. The figure shows that the best speed of 1.23x was achieved at 3 threads (1 encode, 1 decode, 1 transmit) at a segment size of 8MB. The theoretical maximum speedup is 1.33x from the ratio of input and output data size. 5 threads performed marginally worse than 3 threads and it has not been included on the graph since the curves would otherwise overlap and make the graph difficult to read. Since RGB32-24 is a very fast conversion algorithm

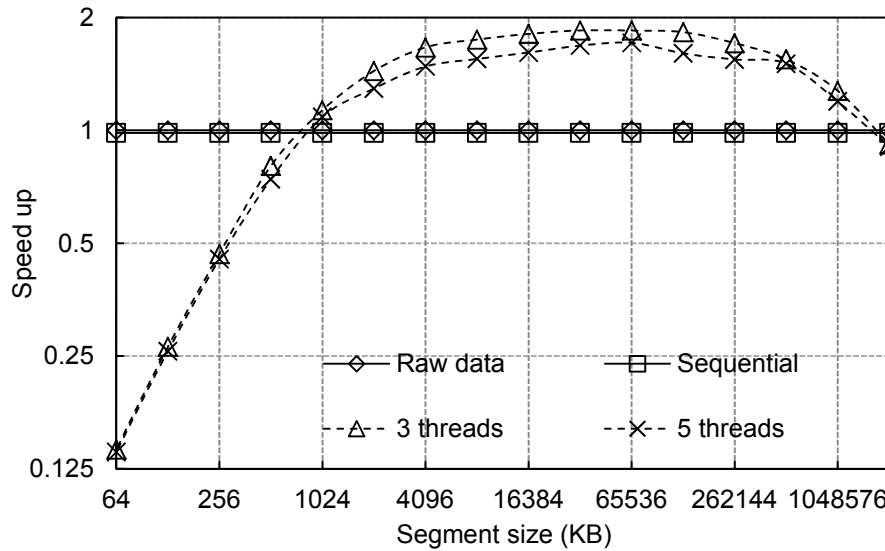


FIGURE 3.22: Speedup for 2GB data against segment size for double-single float conversion showing a peak of 1.85x

(rather than the slower compression algorithms illustrated by Fig. 3.12 and Fig. 3.13 earlier on), 3 threads are sufficient to achieve the best performance; additional threads do not provide any further improvement, but they only add to the overheads. The peak also occurs at a larger segment size, since larger segments can be converted fast due to the simplicity of the algorithm. It is observed that for sequential conversion, there is a slowdown, although not as severe as for delta compression and floating point conversion, due to the fast nature of the conversion.

Similarly, Fig 3.22 shows similar behaviour, where the peak is observed at 3 threads (1 encode, 1 decode, 1 transmit) for a segment size of 32MB. The theoretical maximum speedup is 2x.

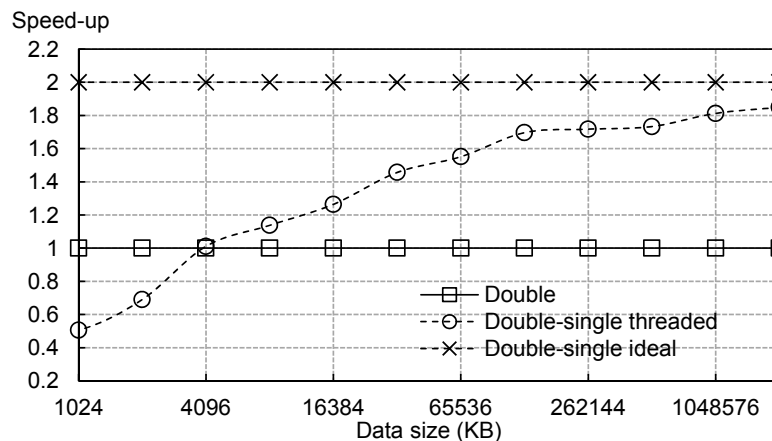


FIGURE 3.23: Ratio of time taken for lossy double precision to single precision conversion

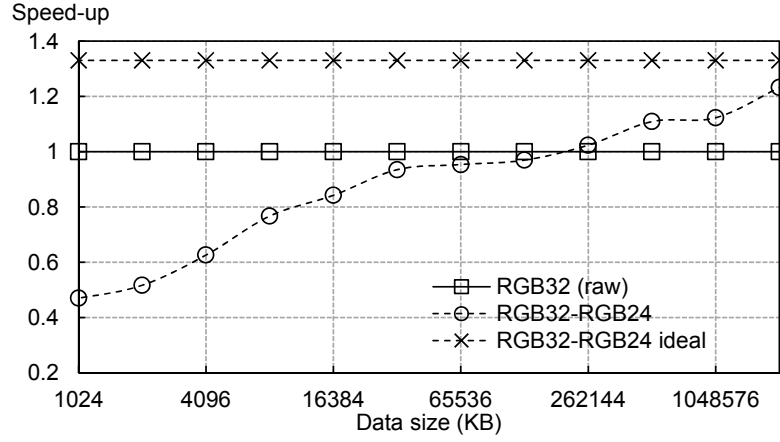


FIGURE 3.24: Ratio of time taken for RGB32-RGB24 image conversion

Figs. 3.23 and 3.24 compares the speedup achieved from lossy double precision to single precision conversion and RGB32 image conversion as the total data size is varied. The ‘ideal’ plots are baseline showing the theoretical best speedup. All the curves are asymptotic towards their theoretical best as the data becomes larger – the theoretical best is the reciprocal of the compression ratio. Therefore, the graph for RGB32-RGB24 tends towards 1.33x and the graph for double-single tends towards 2x.

3.6.6 A Real-life Application

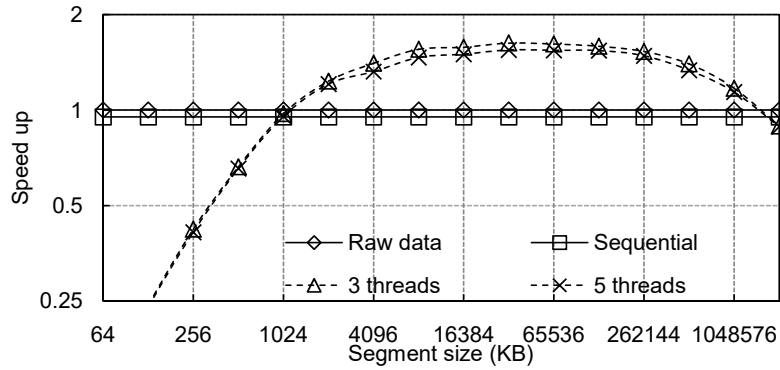


FIGURE 3.25: Speedup for 2GB data against segment size for double-single float conversion applied to an FFT kernel

Fig. 3.25 shows the performance of a real application, namely a heterogeneous application with a double-precision FFT kernel, when the framework is applied. The application runs on a similar Maxeler machine. It shows the speedup for different segment sizes and threads when double precision to single precision conversion is applied to the data transmission, hence the data is transmitted with a reduced size. A speedup of 1.62x was achieved at 3 threads (1 encode, 1 decode and 1 transmit) at a segment size

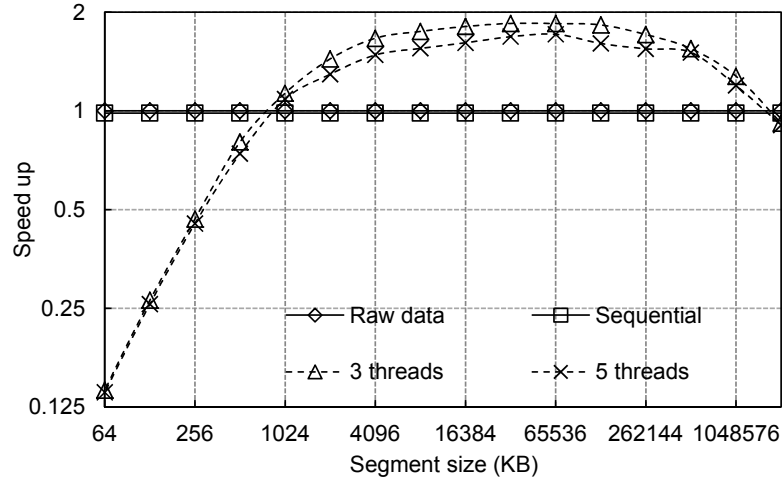


FIGURE 3.26: Speedup for 2GB data against segment size for double-single float conversion

of 32768kB. It also shows that 5 threads performed worse than 3 threads. This is due to the conversion from double to single precision being very fast, and additional threads only cause overheads. However, it is still faster than applying codecs sequentially. The peak occurs at a large segment size, as it did for RGB32-24, since larger segments can be encoded quickly due to the speed of the conversion. The graph of the FFT is not smooth for 5 threads, and the suspected cause of this is memory or cache performance. For a comparison, Fig 3.26 (the same graph was included in the previous chapter as Fig 3.22) shows the performance when the same conversion is applied to the heterogeneous system, when the FFT kernel is replaced with an identity kernel. In this case, a speedup of 1.85x is obtained, which is taken to be the maximum speedup achievable using this conversion method.

Fig. 3.27 tests the FFT for different total data sizes and different segment sizes when data conversion was applied. The number of threads were fixed at 3 since it was observed to be the most efficient number of threads earlier. Larger total data sizes result in a greater speedup, but even for smaller total data sizes, a reasonable speedup is still observed. For example, a 1.62x speedup is observed when the total data size is 2GB and a 1.34x speed is still observed when the total data size is 32MB.

An important point to note is that the speedup in this case has been achieved at the expense of precision. As it was mentioned earlier, a loss of precision in floating point calculations is tolerable in applications such as neural networks and Monte Carlo simulations. An FFT in that sense, may not be the best example of an existing application on

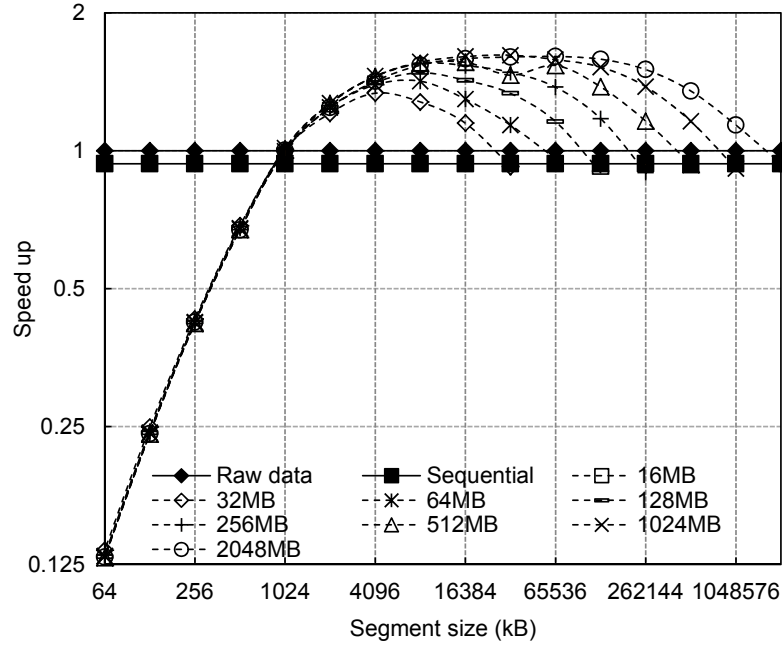


FIGURE 3.27: Comparison of speedup against segment size for different total data sizes when using double-single float conversion applied to an FFT kernel.

which to test the lossy double precision. Since FFTs are often aimed at signal processing applications, we could consider some signal processing applications that can tolerate loss of precision – for example, a finite impulse response (FIR) filter or Gaussian blur.

However, we chose an application from the Maxeler App Gallery that had a high throughput where data is processed on every clock cycle, such that its performance is largely determined by the channel bandwidth. Additionally, in some cases, it may be beneficial to use a single-precision implementation of the FFT kernel to measure the performance with double-single conversion and see how it compares with a double precision kernel where the data is converted back to double-precision – this would be future work.

3.6.7 Summary of Performance Metrics

Table 3.3 shows a summary of how the performance metrics are affected when compression/conversion codecs are applied in a sequential or a threaded manner. Applying codecs sequentially achieves no benefit but applying them in a threaded manner achieves significant benefits.

Considering the delta compression, for example: Where an effective bandwidth of 4256MB/s is achieved, the actual bandwidth of the channel utilised averages out at 2303MB/s,

TABLE 3.3: Comparing the original bandwidth, effective bandwidth, end-to-end latency, speedup and compression ratio for different compression techniques. The measurements were taken using a MAX3 card running MaxelerOS 2015.2.

Compression type	Original Lat. (ms)	New Lat. (ms)	Original BW (MB/s)	Eff. BW (MB/s)	Speed up	Ratio
RGB32-RGB24 threaded	1523	1236	2690	3313	1.23x	0.750
RGB32-RGB24 sequential	1523	3003	2690	1364	0.507x	0.750
Delta threaded	1523	962	2690	4256	1.58x	0.541
Delta sequential	1523	4250	2690	964	0.358x	0.541
Float lossless threaded	1523	1141	2690	3589	1.33x	0.625
Float lossless sequential	1523	7054	2690	580	0.216x	0.625
Double-single threaded	1523	823	2690	4973	1.85x	0.500
Double-single sequential	1523	1547	2690	2647	0.984x	0.500

based on the compression ratio of 0.541. This is reasonably close to the actual bandwidth of the channel, 2690MB/s.

3.7 Conclusion

The capabilities of the framework have been demonstrated using two lossless compression codecs and two data conversion codecs on a Maxeler system. The framework allows codecs to be applied to the data channel between an FPGA chip and a CPU on a heterogeneous CPU-FPGA system, in an analogous way to networking. The results have shown that it is possible to add codecs to perform transformations such as compression on data at line rate as they are streamed through data channels between a CPU and an FPGA, by making use of spare CPU cycles through multi-threading and FPGA resources. Multi-threading allowed encoding, data kernel transmission and decoding to happen simultaneously. Codecs were designed with common interface properties while performing different transformations. Adding compression by the means of this framework has been shown to give an improvement in performance, such as a 1.58x

speedup using delta compression and a 1.85x speedup using double to single precision floating point conversion when the framework is applied to an identity kernel.

Chapter 4

An Adaptive Compression Framework for FPGA Applications

4.1 Introduction

Chapter 3 demonstrated how performance can be improved by using extra resources in both software and hardware. This chapter now expands on the work described in the Chapter 3 by adding adaptivity to compression in heterogeneous streaming systems.

Chapter 2, the background chapter discussed some of the reasons why adaptive compression is useful in applications which handle large amounts of data. In scientific computing applications, large volumes of data are produced and transmitted, and so they can often experience a bandwidth bottleneck. Though such applications can benefit from compression, compression can be difficult to apply because the data is often highly entropic and therefore difficult to compress. Additionally, in big data applications, it is not uncommon for the data format to change, and using a single compression algorithm is not always beneficial. Adaptive compression has been shown to be beneficial in cases where applying a single compression algorithm does not always yield benefits, such as ISOBAR[39] and PACM [38].

The work in Chapter 3 is applicable in the cases where the type and nature of the data being processed is known. When single codecs were tested using randomly generated, incompressible data, there was a noticeable reduction in performance, so there is a strong case to make sure that ineffective compression is not applied to highly entropic data. This means that in such cases, either raw data should be transmitted, or a different type of compression should be used. What would be useful, is compression that can be switched on and off when necessary, or compression where the algorithm can adapt to the data being processed. The aim is to achieve a faster throughput and better compression ratio than by using just a single codec. This would require multiple compression codecs to be present, including an identity codec for cases where transmitting raw data happens to be the most suitable option. This chapter deals with cases where limited information about the data may be available, such that it is not possible to decide exactly which compression codec to use. This chapter explores different ways to use additional resources to achieve improvements in performance. It must be noted that this chapter focuses on simplex transmission while the previous chapter focused on duplex transmission. Applying multiple compression codecs simultaneously is more resource intensive in terms of both software and hardware, such that we are limited to simplex compression.

We propose a platform that can adaptively select between compression codecs at run-time to choose the optimum codec for the data being transmitted between software and hardware sections of a heterogeneous system.

Hence, the contributions of this chapter are:

- An adaptive, multi-threaded system which can apply multiple compression codecs into the software-hardware data channels used to transmit data into an FPGA during run-time and the ability to switch between codecs at run-time.
- A concrete implementation of the adaptive framework on a commercial platform using C code on the CPU side and OpenSPL code on the FPGA side.
- A modified version of the model for predicting the end-to-end transmission latency when the framework is applied to a streaming system.
- Evaluation of data transmission performance on a bandwidth bottlenecked PCI-express channel on a heterogeneous system.

4.2 Proposed Framework

While the previous chapter helped to mitigate some of the bandwidth problems that streaming systems face, this chapter explores additional methods to mitigate the problem in situations where the techniques discussed in the previous chapter may not be effective. The work discussed in this chapter also makes use of extra resources in both software and hardware, but it aims to use the resources in different ways. Most importantly, the compression type in the previous chapter had to be decided at compile time, which would be useful if the nature of the data is known. This chapter aims to tackle situations where the nature of the data is not known, so the programmer cannot be certain what type of compression to use, or whether to use compression at all. Hence, a platform that can support adaptive compression across a software-hardware boundary is necessary. Such a platform will allow the programmer to:

- Apply compression adaptively across a software-hardware boundary.
- Allow the compression algorithm to change at run-time according to the nature of the data being transmitted.

The data to be transmitted is segmented and compression is applied to these segments. A header is appended at the start of the segment, containing metadata including the compressed/uncompressed size, start of stream indicator, end of stream indicator, the ID of the codec, the ID of the segment and the type of compression. The software compresses each data segment using multiple methods and transmits the data that has the lowest size. This requires the use of multiple compression threads and there is a need to balance the load of different compression methods across the threads.

Fig. 4.1 shows a block diagram of the system when codecs are applied to the channel where data is transmitted from the software into the FPGA. When the data to be transmitted is compressed by two different codecs, and the compressed data of the lowest size is selected to be transmitted across the PCIe channel. Once the data is transmitted, the data is directed to the correct decoder based on the type of encoding that the data has. After decoding, the data is merged into a single stream and directed to its destination (e.g. an FPGA kernel).

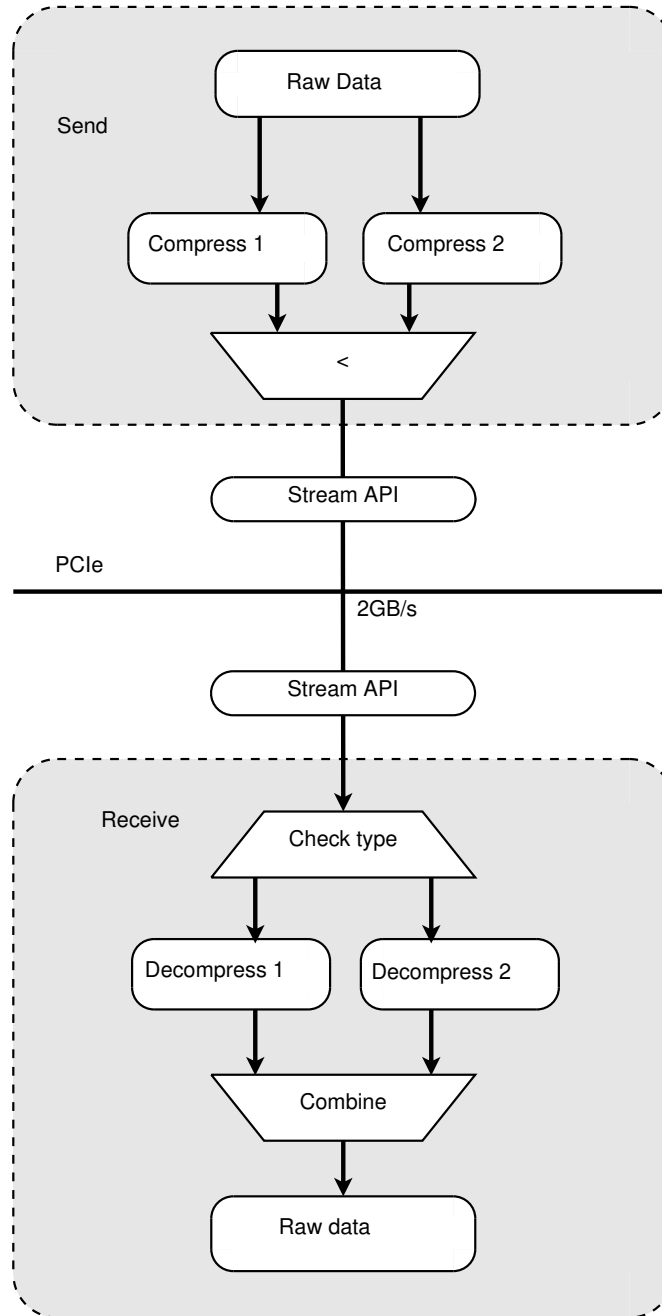


FIGURE 4.1: Block diagram showing dataflow in the system when codecs are applied.

As it was the case in the previous chapter, the compression framework is primarily suited for applications which process and transmit large amounts of data. Since there are various overheads involved when applying these codecs, the overheads will be negligible compared to the total data transmission time in such applications.

One of the major challenges in this implementation is the issue of each software thread placing a greater demand of CPU resources. In some of the frameworks discussed in the

background section (plus Chapter 3), multiple threads were used to parallelise compression – but they were all used to perform the same type of compression. It was observed in Chapter 3 that performance could not be improved further when the total number of software threads approached the total number of physical CPU cores. However, the work discussed in this chapter has different types of parallel compression running at the same time across different software compression threads (in order to determine the best compression method for each segment). This adds significantly more work for each thread to do. Since compression performed here is simplex, it mitigates the demand on software resources.

This section has proposed the idea of selecting compression codecs at run time to a data stream between a CPU and FPGA on a heterogeneous system. The compression codecs operate using spare resources in software and hardware to improve the compression ratio and throughput. The next section will discuss the physical details of how the system is implemented.

4.3 Implementation Details

The adaptive framework is tested using the MaxCompiler [2] tool produced by Maxeler Technologies. The Maxeler machine is the same one used in the previous chapter, and it contains an FPGA and 12 CPU cores interfaced together via PCI-express. While the framework here is tested on the Maxeler system, any streaming system can potentially benefit from the concept discussed in this chapter.

The adaptive implementation of compression allows the codec to be changed during run time. Two versions are implemented. The first one will switch compression on and off depending on the nature of the data – i.e. whether or not the data is compressible – this is done by compressing the data and selecting between whether to transmit the compressed version of the data or the raw data. The second one has three codecs – a delta compression codec, a floating point compression codec and an identity codec. All three codecs are applied to a segment and the size of the processed segment obtained from each of the three codecs is tested. This means that the data being transmitted will have a mixture of different compression methods. The processed segment with the lowest size is transmitted. Multi-threading is used to ensure that this additional compute load does

not cause a reduction in performance. Having multiple codecs operating together will place a greater demand on resources and power than the single codec implementation in the previous chapter – therefore, this adaptive implementation is better suited for simplex transmission rather than duplex.

For codecs in hardware, this would require encode and decode modules to be inserted into the pipeline at the external interface of the FPGA chip. These can be thought of as hardware “threads” in the same way that we have multiple software threads performing encoding and decoding. The hardware codecs must have a throughput greater than or equal to the throughput of the existing bottleneck (in our case this is usually the channel bandwidth). If complicated hardware codecs with iterative loops in them were to be used, their throughput is likely to be lower and the codec is unlikely to operate at line rate.

4.4 Codecs

To demonstrate the framework, a set of compression codecs are applied to the data in a streaming system, leaving the original software and hardware intact, and the performance is evaluated. The methods to be tested are:

- Integer delta compression – where differences between adjacent values in data are computed and represented using fewer bytes.
- Lossless floating point compression – where zeroes in sparse floating point data are packed into a single byte.
- Identity codec – where the data is preserved as it is.

The codecs have been modified slightly from the previous chapter, to meet the needs of the experiments in this chapter.

4.4.1 Delta Compression

The delta compression algorithm has been modified to increase the width of the data stream to 128-bits from 64 bits. This was a decision made partly in hindsight since the

native width of the PCI-express channel is 128 bits and it is also necessary to bring it into line with the other codecs to simplify dataflow control when it operates alongside other codecs. The algorithm is designed such that the compression ratio is 0.5 in the ideal case and the ratio is 2 if the data is completely incompressible. The latter gives a good indicator of how the system will perform in a worst-case scenario where the data is not compressible.

4.4.2 Lossless Floating Point Compression

The lossless floating point algorithm has been modified slightly from the previous chapter. In the previous chapter, this algorithm only checked the first byte and assumed that the rest of the bytes will be zero if the first byte is zero – hence denormal numbers are considered to be zero. This is a reasonable assumption if the data type is known to be 32-bit floating point, since FPGA hardware often equates denormal numbers to be zero. However, this chapter also deals with cases where the data type is unknown, so this assumption cannot be made. When applying multiple codecs simultaneously and deciding which codec is best, we cannot assume the other bytes will be zero if the most significant byte is zero, because the data type is unknown. Therefore, all four bytes must be checked to ensure that the data item is equal to floating point zero. Otherwise there is a risk of corrupting data if an integer value or other data type is compressed using the floating point algorithm. This change ensures that the floating point algorithm is completely lossless.

4.4.3 Identity Codec

The identity codec in software simply copies data from the source array to the array from which data will be transmitted. In hardware, data is streamed straight through the codec.

4.5 Applications

For the purpose of evaluating the performance so far, the same 200MHz identity kernel from the Chapter 3 is reused.

TABLE 4.1: Parameters for model and their typical values

Name	Unit	Description	Typical value
S	MB	size of entire data	2048
s_s	MB	size of segment	8
t_k	s	time overhead of initiating kernel transmit	0.0003
t_{th}	s	time overhead of thread start	0.0002
t_c	s	time overhead of initiating software compression	0.0003
c	-	compression ratio of segmented compression	0.6
a	-	CPU resource contention exponent	0.91
r_{ch}	s/MB	line rate or channel data rate	0.0005
r_e	s/MB	encode rate in a single thread	0.002
r_d	s/MB	decode rate in a single thread	0.002
n_e	-	number of encode threads	2
n_d	-	number of decode threads	2
r_e/n_e^a	s/MB	encode rate for multiple threads	0.001
S/s_s	-	number of segments	256

4.6 Mathematical Model

The model developed in the previous chapter is now modified to be used for simplex transmission, and Table 4.1 defines the parameters that are used in the modified model. Most of the parameters remain the same, with the exception of the time overhead for thread start being removed and the time overhead of initiating software compression being included. As it was in the previous chapter, the model still assumes that there are an infinite number of CPU cores available. The encode rate in this model is taken to be the value for 1 codec – since a commercial adaptive implementation will not have 3 codecs compressing every single segment. Though for the purpose of evaluation later on, some of the data is taken with three codecs running simultaneously for every segment. The CPU resource contention exponent was empirically determined to be 0.91 by measuring the time taken to compress data as the number of threads were varied (as opposed to 0.82 in the previous version).

The time taken to transmit raw data into the hardware from the software is the same as it was in the previous chapter.

$$T_{orig} = r_{ch}S + t_k \quad (4.1)$$

If the entire block of data is sequentially encoded before transmitting into the kernel, the time taken is given by the following equation.

$$T_{seq} = S(r_e + r_{ch}c) + t_k \quad (4.2)$$

It is similar to the previous chapter, except that the decode rate, r_d , has been removed. For comparison, the equation in the previous chapter was:

$$T_{seq} = S(r_e + r_{ch}c + r_d) + t_k$$

When threaded software pipelining is applied, there are 2 possible cases. The ideal case is where there are as many encode software threads as necessary for software codecs to work faster than the line rate such that the channel bandwidth is saturated:

$$\frac{r_e}{n_e} < r_{ch}$$

Again, it is similar to the previous chapter, except that the decode rate, r_d , and the decode threads, n_d , are not there. The equation from the previous chapter was:

$$\frac{r_e}{n_e}, \frac{r_d}{n_d} < r_{ch}$$

The transmission time for the ideal case mentioned above is shown by the following equation. The decode rate, r_d , and thread start overhead, t_{th} , have been dropped from the previous version.

$$T_{p1} = r_e s_s + (t_k + r_{ch} s_s c) \frac{S}{s_s} \quad (4.3)$$

The equation from the previous chapter is:

$$T_{p1} = t_{th} + r_e s_s + (t_k + r_{ch} s_s c) \frac{S}{s_s} + r_d s_s$$

The second case, is where encoding (compression) does not happen fast enough. This would be the case if there are too few compression threads or too many compression threads (hence CPU resource contention slowing down the compression). The decode rate, r_d , and thread start overhead, t_{th} , have been dropped from the previous version. A new parameter, t_c , the software compression overhead, has been added. The transmission time will then be:

$$T_{p2} = \frac{1}{n_e^a} \frac{S}{s_s} (r_e s_s + t_c) + n_e (t_k + r_{ch} s_s c) \quad (4.4)$$

The previous version is:

$$T_{p2} = t_{th} + r_e s_s \frac{1}{n_e^a} \frac{S}{s_s} + n_e (t_k + r_{ch} s_s c) + r_d s_s$$

Overall, the threaded, software-pipelined end-to-end latency is defined as:

$$T_{pipe} = \max(T_{p1}, T_{p2}) \quad (4.5)$$

Note that there are two equations here, as opposed to three equations in the previous version. Whichever is the slower of the two will determine the transmission time.

4.7 Evaluation

The system is built in order to perform experiments to test our hypotheses. We must decide what data to extract from the system and how the performance will be evaluated to test our hypotheses. The experiments in this chapter were also performed on a Maxeler MPC-500 machine, with a Xilinx Virtex 6 FPGA and an Intel Xeon X5650 2.67GHz 12-core CPU with hyperthreading.

The metrics that are used to evaluate the performance are compression ratio, speedup, and transmission time. The input bandwidth is measured by transmitting data into an FPGA kernel which receives the data and discards the data.

In order to evaluate the performance of the system when different codecs are applied using the framework, we run the streaming application under different parameters and take various measurements. Some of the parameters and other factors to be varied are:

- Segment size while total data size is fixed.
- Total data size while segment size is fixed.
- Number of threads.

Varying segment sizes is important since the peak performance will occur at the optimum segment size. Varying the total data size is important, but to a lesser extent since we generally work under the assumption that large amounts of data are transmitted such that many one-off overheads are hidden. Despite this, it is still important to investigate just how large the data size needs to be for our framework to be effective.

Varying the number of threads is especially important since software multi-threading is one of the main ideas we work with – it was observed that applying codecs, even compression, sequentially increased the end-to-end computation time.

The measurements for raw data, without any codecs being applied, is taken as the baseline for the experiment.

4.7.1 Comparison with Model

We compare empirical data against theoretical data from the model. This is useful to establish how accurately the model can predict the performance of a streaming system with the framework applied. Then for future streaming applications, a HPC programmer could determine from the model whether there will be any set of parameters for which the performance of a particular streaming system can be enhanced.

Fig. 4.2 compares the measured transmission time against the time predicted by the mathematical model for 2, 3 and 4 threads. 2 threads means that there is 1 compression thread and 1 transmission thread; 3 threads means that there are 2 compression threads and 1 transmission thread; 4 threads mean that there are 3 compression threads and 1 transmission thread. The model in this case is actually more accurate where there are

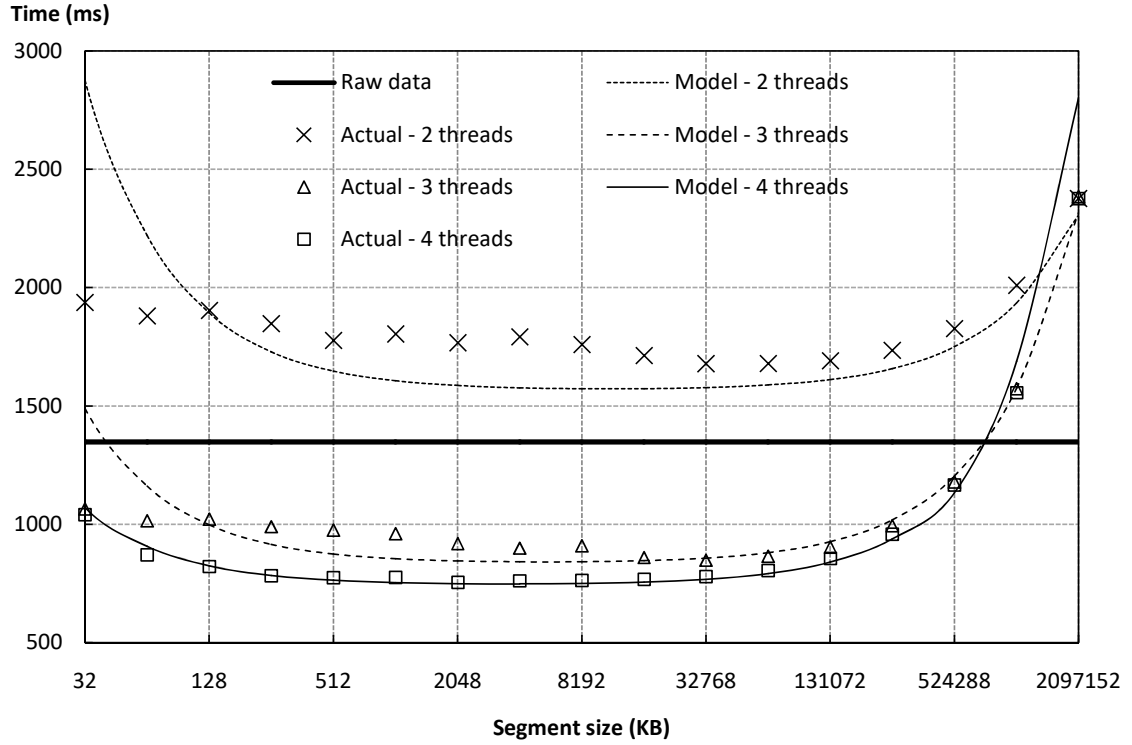


FIGURE 4.2: Modelled/actual time for delta compression with 2, 3, 4 threads.

a larger number of threads, as evident from the diagram. For 4 threads, the maximum discrepancy within the region of interest (where the segment sizes are not too large or small) is 5.56% and the maximum discrepancy overall is 43.0% for a segment size of 2GB. For 2 threads, the maximum discrepancy within the region of interest is 11.0% while the maximum discrepancy overall is 39.8% for a segment size of 32kB.

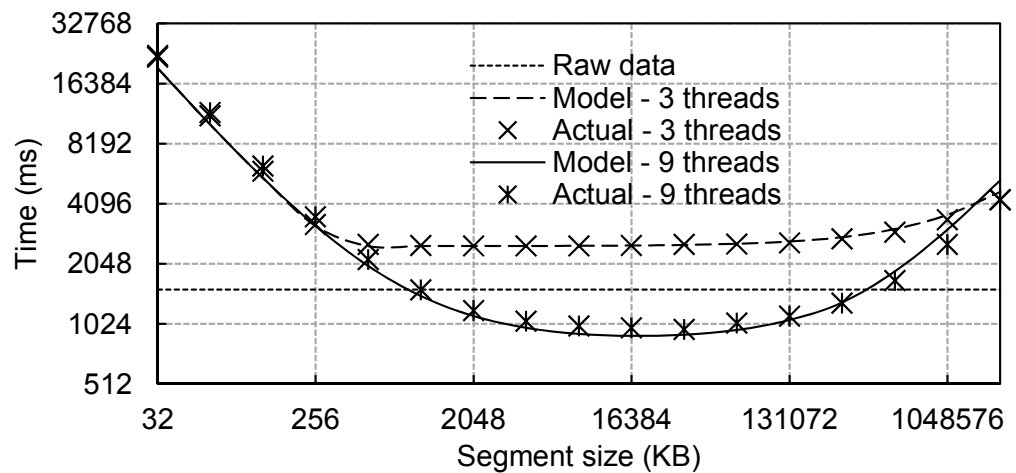


FIGURE 4.3: Modelled/actual time for delta compression with 3 and 9 threads. 9 threads are used here since it gave the best empirical performance.

One thing this model took into consideration which the model in the previous chapter did not is the time overhead of initiating a software compression call. The transmission

times being dealt with in this chapter are smaller than in the previous chapter, hence the overhead of the software compression call where there are a large number of segments contributes towards a sizeable proportion of the total time. Fig 4.3 is shown here for comparison of the transmission times (see the vertical axis), which is one of the graphs from the previous chapter. When the transmission time itself is very large, the overhead of the software compression call is negligible and so it was not included in the initial model.

4.7.2 Adaptive Compression

Two versions of the performance of the system operating under adaptive compression was tested.

The first one used a delta compression codec and an identity codec. Fig. 4.4 shows the measured performance.

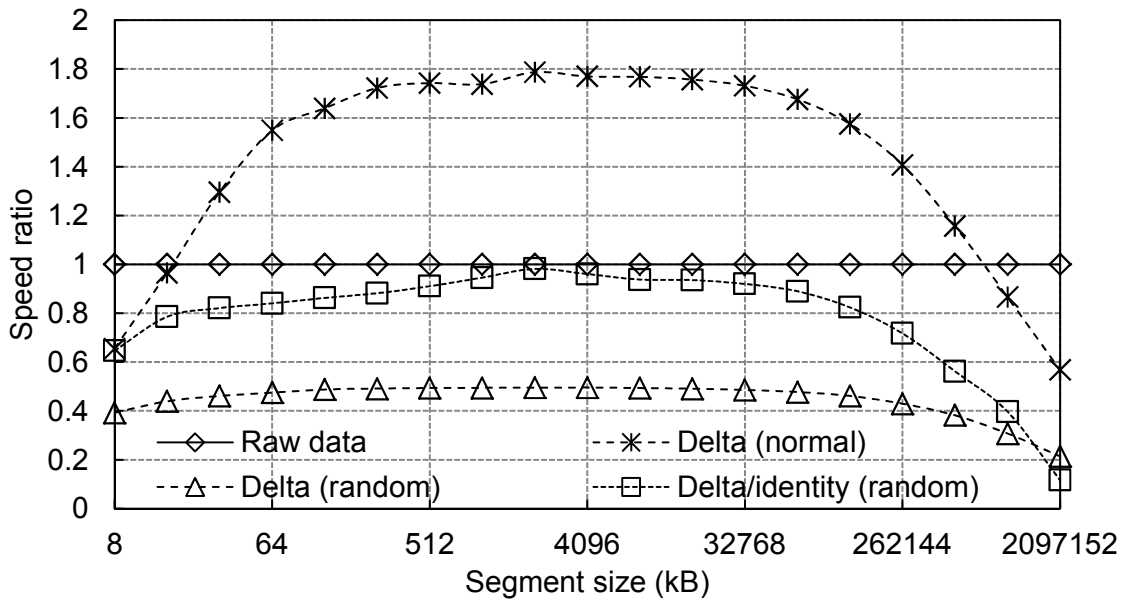


FIGURE 4.4: Graph showing performance of adaptive compression with two codecs. “Delta (normal)” shows delta compression applied to compressible data. “Delta (random)” shows the worst case scenario, where the delta codec was applied to incompressible random data. “Delta/identity random” shows the compression being overridden by the identity codec where the data is random and incompressible.

Delta (normal) shows the performance of the delta compression algorithm operating on its own using compressible data. With 3 compression threads and a compression ratio of 0.55, a speedup of 1.78x was observed at a segment size of 2MB.

Delta (random) shows the worst case scenario if delta compression were applied to randomly generated incompressible data. With 3 compression threads and a compression ratio of 2 (the data size gets doubled) speed ratio of 0.496x was observed at a segment size of 4MB. The expansion is high when the data is incompressible, so this gives a good indicator of how the framework performs if it is not used correctly.

Delta/identity (random) is where incompressible random data is used and adaptive compression overrides the delta compression to transmit raw data instead of compressed data. This aims to mitigate the effect of the slowdown caused by data expansion when the data is incompressible. With 3 encode threads (where each thread applies both of the codecs to a particular segment), a speed ratio of 0.981x was observed at a segment size of 2MB. Hence the adaptive framework is effective at mitigating loss of speed in the case where data is incompressible.

The second version, Fig. 4.5, used a delta compression codec, a floating point compression codec and an identity codec. The encoded version with the lowest size will be transmitted. The data content and data type of the stream of data being used in this experiment changes over time. The data alternates between integer data, floating point data, and random data (which is incompressible by both algorithms). Therefore, the framework should now automatically choose whichever codec is the most appropriate for the data based on the compression ratio. Applying just a single codec, such as delta compression, is not appropriate since it will not be able to compress the data most of the time. The best performance achieved using only a delta compression codec is a speed ratio of 0.851x. However, when 3 codecs were applied simultaneously, a speedup of up to 1.34x was observed. This shows that using adaptive compression is an effective way of compressing and transmitting data where the nature of the data is unknown. Furthermore, it shows how using extra software and hardware resources (if available) can be used to improve performance. The reason for why the graph is not smooth has not been determined.

The random data used in both of these experiments was dense floating point data that was found to be incompressible with both the floating point compression algorithm and the delta compression algorithm. Interestingly, there are some cases where floating point data may be better compressed by the delta compression algorithm and there are some

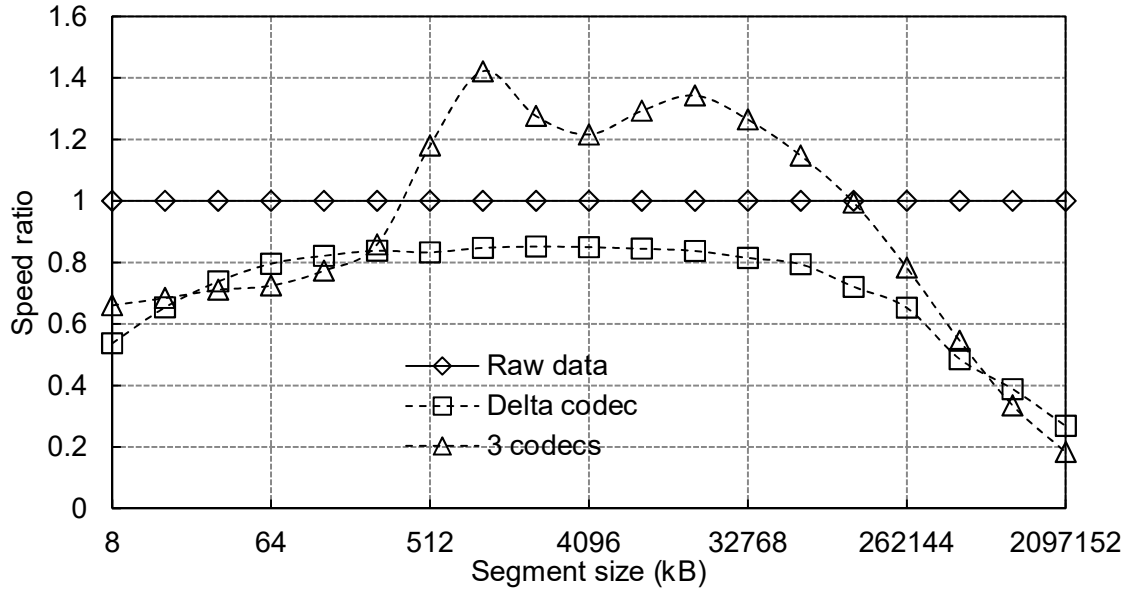


FIGURE 4.5: Graph showing performance for mixed data using a single codec and using three adaptive codecs.

cases where integer data may be better compressed by the floating point compression algorithm.

4.8 Applications

In the previous chapter, the framework was applied to an FFT, and a speedup was observed in comparison to raw data being fed into the FFT. The speedup achieved on the FFT was slightly less than what was achieved on a black box kernel.

Although not tested here, if adaptive compression were similarly applied to an FFT, the expectation is that there would be a speedup, but slightly slower than what it would be when compression is applied to a black box.

4.9 Conclusion

In this chapter, the framework has been extended to use simultaneous multiple codecs, and an alternative version of the model has been developed for simplex compression.

The extended version of the framework allows multiple codecs to be simultaneously applied to the data channel between an FPGA chip and a CPU on a heterogeneous

CPU-FPGA system. Multi-threading allows simultaneous compression techniques to operate in software with minimal overhead. Using extra resources on the FPGA allows fast and small compression codecs to be implemented on hardware.

On a segmented data stream where the nature of the data changes at run time, a speed-up of 1.34x was observed when using 3 simultaneous codecs. It was observed that using an identity codec with a compression codec can also prevent a loss of speed in the event that the data is incompressible.

There are however, concerns regarding power consumption, since a large amount of software computation is performed to apply multiple compression codecs to every segment in order to test the compression ratio before transmitting. In a commercial implementation, applying multiple compression codecs to every single segment would be too power intensive to be practical. Therefore, it would be more sensible to test the compression ratio of the data occasionally, instead of testing it for every segment. This could also make the adaptive framework viable for duplex compression. For example, the compression could be tested every 128 segments, which would use significantly less power. More ideas on adaptive compression are discussed in Chapter 7 (Future Work).

Chapter 5

Application of the Framework to Non-streaming Platforms

5.1 Introduction

The previous two chapters described how codecs can be applied to the data channel on a Maxeler streaming system to improve performance. The results showed performance of the system operating under various codecs and combinations of codecs when applied to a commercial platform.

This chapter explores OpenCL, an alternative platform for heterogeneous systems. OpenCL is a popular HLS platform for implementing HPC applications. It is well suited for HPC applications since it simplifies the design process and it allows the programmer to use pipeline parallelism and replicated pipelines to achieve a high throughput. The common usage of the platform means that knowledge of the platform is widely available. The flexibility of OpenCL allows it to operate across a range of different platforms, especially heterogeneous applications. These include GPUs, FPGAs and SoCs.

HPC applications designed in OpenCL still face the same challenge of a bandwidth bottleneck, if they need to transmit data between the software and hardware accelerator. When a kernel designed in OpenCL contains constructs such as work groups, loop unrolling and multiple compute units, the kernel will have a massive throughput. Work groups are blocks of data that have the same calculations performed on them. Loop

unrolling allows sequential calculations to be parallelised, as long as there are no dependencies between the loop iterations. Compute units are a construct to replicate the computation hardware. However, the throughput is often limited by the bandwidth of the data source which supplies the kernel, and/or the data sink which receives the data that is produced by the kernel. This would usually be memory or another data channel such as PCI-express. This prevents the programmer from making full use of the kernel's computational capabilities. This necessitates the use of compression to increase the effective data bandwidth, to allow the kernel to achieve a higher throughput.

Like other HLS platforms, OpenCL lacks the primitives to deal with the bandwidth issue. It also knows limited information about the data coming into the kernel from outside.

This chapter takes the concepts proposed in previous chapters and applies them to another heterogeneous CPU-FPGA system based on OpenCL. It is useful to apply the framework to a different, non-commercial platform to demonstrate its generality. The aim is to apply compute-intensive compression at line rate by making use of unused CPU cycles on the software side and spare FPGA resources on the hardware side of the channel. Hence, the contributions of this chapter are:

- An implementation of the framework on an OpenCL platform to demonstrate the generality of the framework.
- An extension of the mathematical model and its application to the framework on a different platform to show its generality.
- Comparison of empirical performance with model-predicted performance.
- Evaluation of the performance of data transmission across a bandwidth-bottlenecked PCI-express channel and a bandwidth-bottlenecked memory channel.
- Evaluation of memory compression performance, showing a decompression bandwidth of up to 7805MB/s for lossless floating point.

Section 5.2 provides a detailed description of the framework described in this chapter. Section 5.5 describes a mathematical model to predict the performance of streaming systems operating under this framework. Section 5.4 discusses the technical details of

how the system is built on the OpenCL platform. Section 5.6 evaluates the performance of the framework on a commercial platform when different codecs were applied to streaming systems.

5.2 Conceptual Framework

So far, the conceptual framework has been tested and shown to work on the Maxeler platform, which is a streaming-based platform. Since Maxeler is a proprietary platform with tightly integrated hardware, drivers and software, it is necessary that the framework must work on other platforms in order for the framework to be general. In this chapter, the framework that was conceptualised in the previous chapters is applied to a new platform that is not based on streaming.

The previous chapters have discussed the necessity of a general framework, and to demonstrate the general framework, it must meet certain requirements. These requirements are:

- Allowing the programmer to apply software and hardware codecs to the data without major changes to the software or hardware.
- Using unused/extra resources in software and hardware.
- The framework is extensible such that other codecs can be applied to a data channel.
- The framework must be applicable across different platforms.

Therefore, in this chapter, the framework is tested on a different platform: Namely, the OpenCL platform. Some of the main differences between the Maxeler and OpenCL platform are:

- OpenCL is a vector based platform unlike Maxeler which is a streaming based platform, such that they are suited for different applications.
- OpenCL is an open platform that is applicable to CPUs, GPUs, FPGAs and SoCs while Maxeler is a proprietary platform with tightly integrated hardware, drivers and software.

- Data transmission in OpenCL is typically done via on-card memory while data transmission in Maxeler is typically done directly between the host and the FPGA kernel.
- OpenCL gives a greater degree of flexibility over on-card memory accesses (though it requires the programmer to design these accesses in an efficient way) while Maxeler has a rigid memory access pattern (and highly efficient for specific applications).
- OpenCL does not have a strict order in the way that data is processed while Maxeler has a strict ordering of data in its data streams.
- Parallelism on OpenCL is achieved using work items, loop unrolling and compute units, while parallelism on Maxeler is achieved using a highly optimised streaming pipeline.

Despite the differences between the two platforms, there are also a number of similarities:

- Both can be used to implement heterogeneous HPC applications.
- Both provide a software API that allows the transmission of large amounts of data between the host processor and the hardware using DMA.
- Both support applications with multiple kernels.
- Both face the challenge of a bandwidth bottleneck.

OpenCL is therefore a useful platform to test the framework on. It is different enough from Maxeler to demonstrate the generality of the framework, while having enough similarities for the framework to still be applicable and useful. The flexibility of the OpenCL platform allows the framework to be tested on not just FPGAs, but also GPUs. While the main focus in this chapter is the application of the framework to a different CPU-FPGA system, it is also applied to a CPU-GPU system.

Since data transmission between the host and the hardware in OpenCL applications is typically done via on-card memory, the focus of this chapter is also simplex transmission rather than duplex transmission. This lessens the need for multithreading in software as it will not be necessary to have simultaneous compression and decompression in software.

However, software pipelining is still used in the form of having simultaneous compression and data transmission. The host accesses the on-card memory to read/write and the FPGA kernel also accesses the on-card memory to read and write. The use of on-card memory on OpenCL is less restrictive than the usage of on-card memory on the Maxeler platform, where the data transmissions have a strict granularity. This gives a greater flexibility to apply compression as the data size can vary significantly depending on the compression algorithm being used and the nature of the data being compressed.

OpenCL gives the programmer a large degree of control over the types of memory that are used by computation kernels. It has a memory hierarchy consisting of four levels: Global; constant; local; private. On an FPGA system, global memory will be the off-chip DDR memory on the board. Constant memory can be used to store read-only data that is used by the kernel, and the data can be stored in an on-chip scratchpad. Local memory is for smaller chunks of data and it is stored on on-chip block RAM. Private memory is data that is stored on registers. The programmer can design the kernel to make use of these types of memory in ways to make the kernel as fast and efficient as possible. OpenCL is often used to design kernels that repeatedly read and write large amounts of memory. Maxeler does allow the use of memory locally on the chip and the use of registers but it does not automatically create and control scratchpads. While OpenCL gives a large degree of control to the programmer, it can be challenging to achieve optimum performance in a system since a large design space needs to be considered and explored.

The OpenCL platform also gives the programmer a number of different ways to parallelise hardware implementations. These include work items, compute units and loop unrolling. Maxeler does not provide such options, however, it provides a single highly optimised pipeline where data is streamed through. While OpenCL gives more flexibility, parallelism through multiple compute units and loop unrolling will result in systems that use a larger amount of resources. In a highly parallel OpenCL system, the bandwidth available to supply the system with data will be a bottleneck.

OpenCL also has a different way of managing the ordering of data segments compared to Maxeler. Due to the nature of work items, they may be performed out of order, while Maxeler has a strict ordering of data in its data streams. Codecs without data dependencies can be parallelised through the use of work items or loop unrolling.

As mentioned in the previous chapters, it is not expected that this framework will be applicable to every bandwidth-bottlenecked heterogeneous HPC application. So far, the investigation has been on applications where the performance is restricted by the bandwidth bottleneck in the channel between the host processor and the hardware. There are however, other applications where data is transmitted from the host to the on-card memory where the on-card memory will then be read/written multiple times by the kernel. In this case, the CPU-FPGA bandwidth will only account for a fixed proportion of the total time – the overall speedup that can be achieved by purely focusing on the CPU-FPGA bandwidth is limited by the proportion of time that is actually spent transmitting data. However, this chapter does also focus on cases where data is written to the on-card memory where the kernel processes the data after it has been written. This is relevant for kernels which repeatedly access large amounts of data on the on-card memory.

5.3 Codecs

The compression codecs to be tested are:

- Lossy double precision – where 64-bit floating point numbers are converted to 32-bit floating point numbers.
- Lossless floating point – where zeroes in sparse floating point data are reduced to a single bit in a bitmask.

Delta compression was not used here, since it will not parallelise well on an OpenCL platform when there are data dependencies between adjacent values. Though it can be parallelised across segments, there is limited scope for parallelisation within segments – therefore, this algorithm is better suited for Maxeler which has a deep pipeline, rather than OpenCL. It was suitable for Maxeler since Maxeler is specially designed for streaming.

5.3.1 Lossy Double Precision

As it did in Chapter 3, Lossy double precision converts from IEEE 64-bit floating point to IEEE 32-bit floating point. The conversion is performed the same way as in Chapter 3. The only difference is the presence of a 32-bit header at the start of the block of encoded data to indicate the number of floating point numbers in the block. Assuming that the size of the block of data is much larger than the header, the compression ratio will be approximately 0.5. Lossy double precision is a fast and efficient codec. Since this can be easily parallelised and pipelined, its speed is limited by how fast memory can be accessed.

5.3.2 Lossless Floating Point

The lossless single precision float algorithm used in the OpenCL platform is very different from that used in Maxeler. The algorithm here is designed to be efficient on an OpenCL platform – it makes use of multiple memory reads and writes and it is internally parallelisable. When the floating point data is compressed, each floating point element of data is checked. If the element is zero, the element is removed and a binary ‘0’ is stored in a bitmask. If the element is non-zero, a binary ‘1’ is stored in the bitmask and the value of that element is kept. The dataflow is illustrated by Fig. 5.1. This allows the decompressor to then check the bitmask to determine whether or not each element is zero or non-zero and use it to reconstruct the original data. The dataflow in decompressor is illustrated by Fig. 5.2. At the start of an encoded block of data, a 32-bit header will indicate the number of floating point values in that block. The best case compression ratio of approximately 0.03125 ($1/32$) will occur if all the numbers are zero, such that all the numbers are removed and only the header and bitmask will be present. This assumes that the size of the block of data being compressed is much larger than the header. The worst case compression ratio of approximately 1.03125 ($33/32$) will occur if none of the numbers are zero, such that all the raw numbers are transmitted along with the bitmask.

It must be noted that while there are existing sparse data compression techniques, they are aimed at highly sparse data, while this algorithm can also be used for data with moderate sparsity.

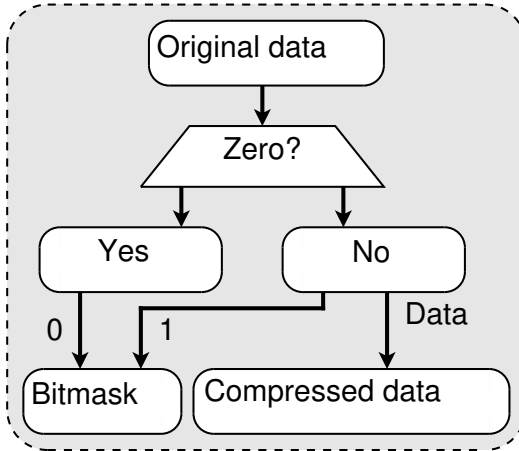


FIGURE 5.1: Block diagram showing dataflow when the lossless floating point compression algorithm is applied to sparse data.

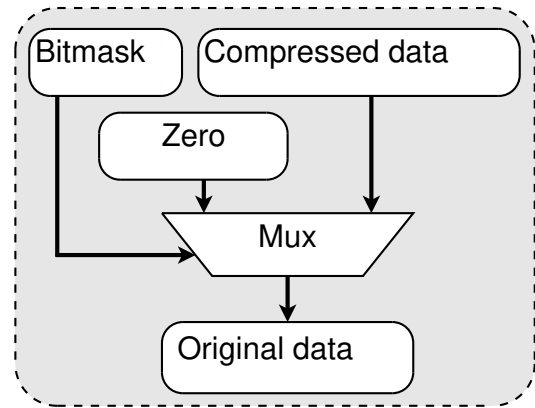


FIGURE 5.2: Block diagram showing dataflow in the decompression of floating point data.

5.4 Implementation

The framework is built using OpenCL on a heterogeneous machine with a multi-core CPU and an FPGA interfaced together via a PCI-express channel. On this system, bandwidth across the PCIe channel is in fact limited by the driver of the FPGA card, which can only handle data transfers up to 3.1GB/s according to the specification. The bandwidth was in fact measured to be 2308MB/s by transmitting a 2GB block of data into the on-card memory via PCIe. The OpenCL software libraries and API allow communication and data transmission between the CPU and FPGA and it allows for high speed transmission of data across the channel. This platform is a suitable match for the requirements of the overall investigation. The data is transmitted to the FPGA via on-card memory, and not directly to the FPGA as was the case with Maxeler. The flexibility of OpenCL allows this to be a practical option, while it was not particularly practical to implement data transmission via the on-card memory on the Maxeler platform for the particular requirements of this investigation. When the framework is applied on an OpenCL platform, compression in software is still done by using otherwise unused CPU cycles and the compression on the hardware side is still done by making use of unused FPGA resources.

The hardware implementations used in OpenCL implementations must not have many loop dependencies as this complicates parallelism and often reduces the effectiveness of parallelism. While work items are generally processed in order, it cannot be assumed

to be the case and we must design the compression kernels in such a way that they still function correctly, even if the work items are processed out of order. Work items and loop unrolling are the primary ways which parallelism is used in this framework. The coarse segmentation will be similar to that used in Maxeler, which only used coarse segmentation.

In the case of sequential compression, the entire block of data to be transmitted is compressed prior to transmission to or from the FPGA. In the case of threaded compression, the block of data is divided into segments, and segments are transmitted as they are compressed. The way the segments are processed in the OpenCL hardware implementation is different from the way the segments are processed in the Maxeler hardware. There are two versions of the threaded implementation: one where data is transmitted to the FPGA card as it is compressed by the CPU; the second where the data is transmitted to the FPGA card and the FPGA kernel is executed as the data is compressed by the CPU.

During implementation, local memory was tested but not used for the final implementation of the OpenCL codecs. The hardware was found to be very fast using global memory due to the memory access patterns. Global memory accesses were found to be very efficient, since many of the accesses are sequential rather than random such that a scratchpad formed of local memory would not be particularly necessary.

Compute units (an OpenCL construct to replicate the hardware) were also tested and found not to have any major impact on performance but their performance data have been included in the evaluation section for completeness.

For implementation on the GPU, the data sizes used were limited by the memory available to the GPU. The simplex bandwidth of the PCI-express channel between the CPU and GPU was measured to be 2990MB/s.

5.5 Mathematical Model

The mathematical model used in this chapter is identical to the modified model developed in Chapter 4 and it is now applied when data is transmitted between the CPU and the hardware on a different heterogeneous system.

When the PCIe channel is the bottleneck, the model is applicable whether the data is being transmitted from the CPU directly to the FPGA kernel or whether the data is being transmitted from the CPU to the on-card memory of the hardware accelerator

5.6 Evaluation

Some of the experiments for this chapter were performed on a machine with two Intel Core i3 CPU cores with hyperthreading and a Nallatech card with an Altera Stratix V FPGA. A machine with two Intel Core i3 CPU cores with hyperthreading and an Nvidia Quadro K4000 GPU was also used for some of the experiments.

The metrics that are used to assess performance are simplex transmission time, effective bandwidth, speedup and compression ratio. Due to the nature in which OpenCL is used in many applications plus the fact that we work with a machine that has only 2 physical CPU cores, this chapter also focuses on simplex transmission, so there will not be a huge emphasis on software multithreading. It is important to note that many of the measured results in this section involving software do not show speedups because they were measured using a single compression thread due to the limited availability of CPU resources. However, we use the model to predict the performance in a situation where there are a larger number of CPU cores available.

While the performance is measured by applying different types of compression, the effect of varying certain parameters are also observed. OpenCL lets the user control the parameters of the system in a different way to that of Maxeler. The parameters to be varied are:

- Segment size.
- Total data size.
- Work size
- SIMD work items
- Loop unrolling
- Data sparsity (for lossless floating point compression)

Software pipelining will still be used to perform line rate compression for the communication between the CPU and the FPGA board via PCI-express. As observed in Chapter 3, even without a large number of compression threads, data transmission across the PCI-express channel can sometimes be faster when software-pipelined segmented compression was used.

Better improvements in performance are expected for larger total data sizes since the data transmission time will be dominant in comparison to the fixed overheads involved in data transmission. The majority of time should be spent on transmitting data rather than handling overheads.

The performance when varying parameters is tested on lossless floating point compression and lossy double to single conversion. The codecs chosen all have high degree of parallelism that can be exploited using OpenCL.

Where the lossless compression codec is used, the total data size is 1GB. Where the lossy double precision codec is used, the total data size is 2GB for the FPGA and 1GB for the GPU.

5.6.1 Comparison with Model and Model Prediction



FIGURE 5.3: The model is tested on line-rate compression where data is simultaneously compressed and transmitted across the PCIe channel.

Fig. 5.3 shows the phases of the data processing which this model is tested on – online compression as data is transmitted from the software to the hardware.

Fig. 5.4 compares the measured input time to transmit 1GB of data from the CPU to the onboard memory of the FPGA board for lossless floating point compression against the time predicted by the mathematical model. Since the machine has only 2 physical CPU cores, it is only possible to test the model on the OpenCL platform using 2 threads. This is not ideal since two threads are not enough to achieve line rate in software, but the model has been shown to work with a larger number of threads when tested on the Maxeler platform – so it can be used to predict performance with more threads. The maximum discrepancy within the region of interest is 8.79% observed at a segment size

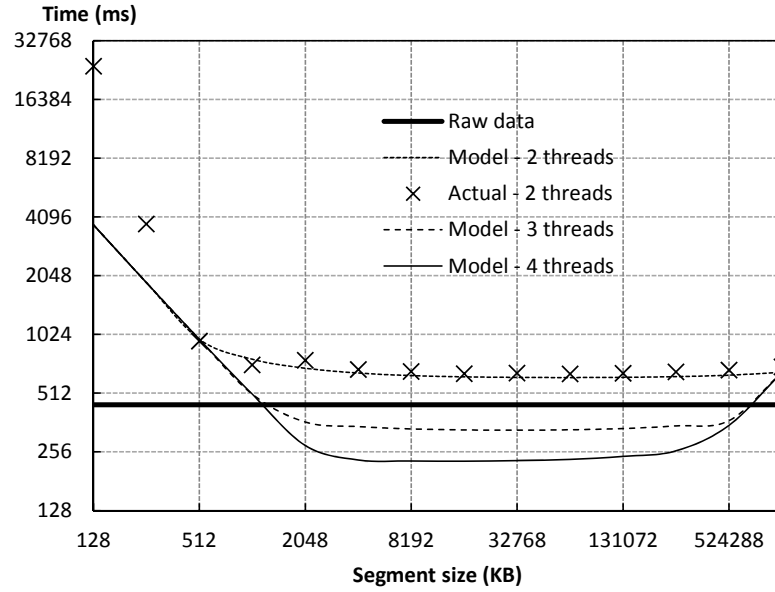


FIGURE 5.4: Comparison of the time taken between measured values and predicted values for lossless floating point compression.

of 2MB and the maximum discrepancy overall is 84.6%, observed at a segment size of 128kB. From this, it would appear that the hardware used in this machine has much larger overheads when processing a large number of small segments.

So the model must not be used for smaller segment sizes since it cannot accurately predict the performance at those segment sizes. Realistically, an application would operate near the centre where the discrepancy is much lower.

The mathematical model used in this chapter is identical to the model used in Chapter 4 (though slightly different from Chapter 3), and the data shows that the model is flexible and applicable to different platforms. Therefore, it is reasonable to use the model to predict the performance of other systems.

Since the model has been shown to be relatively accurate for making estimates of the performance on both the Maxeler platform and an OpenCL platform, it can be used for making predictions on performance where the necessary hardware is not available. Since the machine used in this experiment is not suitable for software multi-threading, the model can be used to estimate what the performance would be on a different machine that supports a larger number of CPU threads, where the specifications would otherwise be the same except for the number of threads that the CPU can support. Fig. 5.4 also shows the predictions of the time taken to transmit 1GB of data for lossless floating point compression.

The predictions from the model show that using 3 threads (2 compression and 1 transmission) or 4 threads (3 compression and 1 transmission), can achieve a faster performance than raw data for moderate data sizes. This is a reasonable assumption since the previous chapters have already shown the model to be reliable. Measured data from the previous chapter also showed that using 3 or 4 software threads can achieve a faster performance than transmitting raw data (since the software codecs will achieve line rate with this many threads), and the performance predicted by the model then was shown to be accurate in the region of interest (moderate segment sizes).

5.6.2 Varying the Data Sparsity in Lossless Floating Point Compression

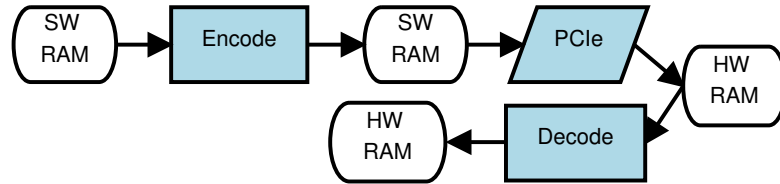


FIGURE 5.5: The performance of compression, transmission across PCIe and decompression are investigated separately in this section.

Fig. 5.5 shows the phases of the data processing where different levels of data sparsity are tested.

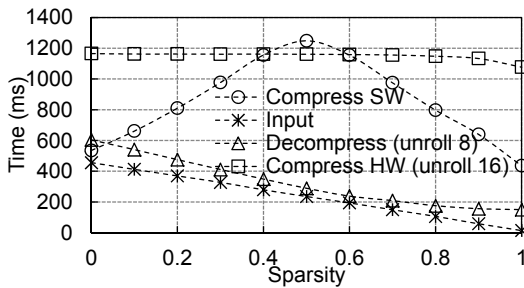


FIGURE 5.6: Graph showing how the sparsity of data affects the transmission time between the CPU and on-card memory, and the decompression time on FPGA hardware for a lossless floating point compression algorithm.

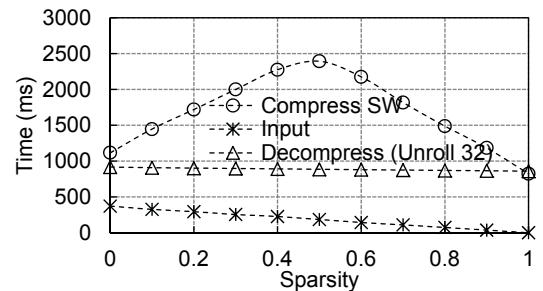


FIGURE 5.7: Graph showing how the sparsity of data affects the compression time, transmission time between the CPU and on-card memory, and the decompression time on GPU hardware for a lossless floating point compression algorithm.

Figs. 5.6 and 5.7 are included to show a comparison between a heterogeneous CPU-FPGA system and a heterogeneous CPU-GPU system. It was initially done as a preliminary experiment to make some initial observations. A fixed total data size of 1GB was

used. This is an appropriate size since the framework is designed for applications where large amounts of data are processed. While 2GB was used in the previous experiments, technical limitations restricted the size to 1GB in this experiment. The lack of memory on the host machine prevents 2GB of data being used for this particular codec, and the GPU card has insufficient memory to support 2GB of data. The compress time is measured using a single thread in software.

The peaky shape of the software compression graph is due to the nature of the floating point compression algorithm. At high sparsities, very little data has to be written to RAM when compressing. At low sparsities, a large amount of data is written, but in a somewhat consistent pattern. At medium sparsities, the nature of the data is unpredictable and the RAM accesses are inefficient. Even though the compression ratio is not bad, the worst case performance is disproportionately slow as a result of random global memory accesses.

Judging by the significantly faster performance of the hardware compression in comparison to the software, this would be particularly well-suited to systems where data from the hardware global memory is read by the FPGA multiple times. It should be noted how unroll 8 was used in the FPGA and unroll 32 was used in the GPU – the two devices are fundamentally different architectures, so the implementations had a number of differences. The FPGA implementation has a work size of 1, and an unrolling factor of 8 is applied to the fine-grained inner loop that iterates along bits of the bitmask and the elements that each bit of the bitmask corresponds to (the outer loop is coarse-grained). Loop unrolling on an FPGA is extremely resource intensive, so unrolling it to 32 is not a sensible option, even though it will give even better performance – there is a trade-off between resource usage and performance, and there is more discussion further ahead. The GPU implementation has a work size of 32, and this is used to parallelise along the bitmask – there is no inner loop that iterates on the bitmask. The loop unrolling is applied to the coarse-grained outer loop in the GPU. The GPU used in this experiment used in this experiment is quite old and therefore cannot perform this particular algorithm at line rate, but from its performance, it is clear that newer models will be able to operate at line rate.

The software compression is slow because only one CPU is used to compress the data.

The effectiveness of parallel software compression has been shown in the previous chapters so compression would be more effective on a system with a larger number of CPU cores. Also, for reasons that are unknown, the software compression on the GPU machine is significantly slower than the FPGA machine, although both machines have the same specification.

Furthermore, the results indicated that at a sparsity higher than 0.5, the decompression bandwidth in the FPGA implementation is higher than the bandwidth of the PCI-express channel. Most sparse data compression algorithms require highly sparse data, while this framework could allow compression to be applied to data that is less sparse than what typical sparse data compression algorithms would deal with. For example, the sparsity of data in algorithms such as sparse matrix multiplication is much higher than the sparsities that this experiment deals with. Hence the compression here could potentially accelerate algorithms such as dense matrix multiplication in cases where data is not completely dense.

The hardware compression is slow in the FPGA, though the decompression is much faster. The hardware implementation of this lossless floating point compression algorithm does not parallelise well in the FPGA. It involves writing data to random locations in memory as the data is compressed – and the memory address where each individual element of data depends on the nature of the data being compressed. An unfortunate disadvantage of OpenCL is that it is not effective at pipelining writes to unknown memory locations at compile time. As a result of this, much of this section will focus on the hardware decompression.

5.6.3 Varying Segment Size



FIGURE 5.8: Compression being applied as data is transmitted from software to the onboard memory of the hardware (the same as in the model)

It is important to note that there is only one software compression thread in this experiment (due to the limitations of the host machine), which is not sufficient to achieve a speedup using the lossless floating point compression algorithm. It is still worth observing the performance under different parameters, with the knowledge that it will be

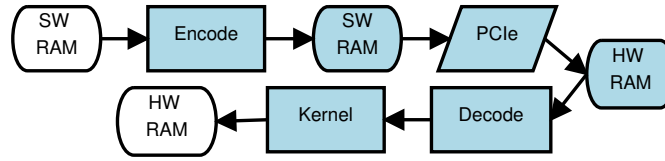


FIGURE 5.9: Compression being applied as data is transmitted from software to the onboard memory of the hardware and then decompression being applied before it is transmitted through an FPGA kernel.

faster with a larger number of threads (as shown by the estimation from the mathematical model).

Figs. 5.8 and 5.9 show the two cases which are tested by varying the segment size, on both lossless and lossy compression. The former is the same case that is handled by the model – the data is simultaneously compressed and transmitted to the onboard memory of the hardware accelerator. The latter is where the data transmitted to the FPGA is decompressed and transmitted through an FPGA kernel. For the purpose of this experiment, a floating point multiplier is used.

5.6.3.1 Lossless Floating Point

These experiments are performed on 1GB of data. Figs. 5.10 and 5.11 show the speed ratio observed from using different segment sizes for lossless floating point compression. The software-only version Fig. 5.10 shows the case where data is simply transmitted into the on-board memory while Fig. 5.11 shows the case where the FPGA kernel is also executed as the data is transmitted onto the FPGA board. Both the experiments use data with a sparsity of 0.9 (90%). The performance of the software-only version appears to be particularly erratic – the exact reason for this has not been established, though we suspect it is related to CPU cache performance.

Fig. 5.12 shows how changing the unroll factor on the hardware affects software-pipelined data transmission from the CPU into the FPGA hardware.

5.6.3.2 Lossy Double Precision

The experiments here are performed on 2GB of data.

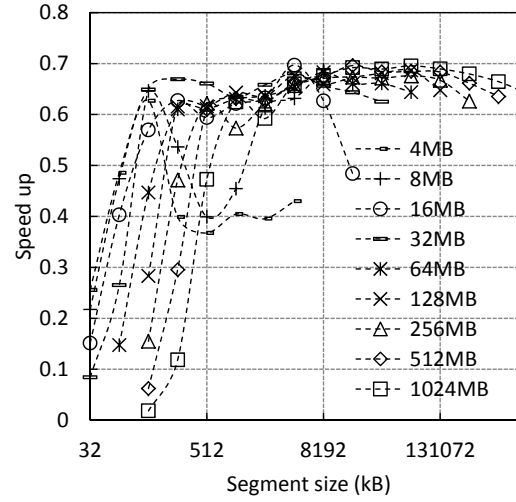


FIGURE 5.10: Speed ratio against the segment size for lossless compression of sparse floating point data for different total data sizes, where software-pipelined data is transmitted into the on-board memory of the FPGA board.

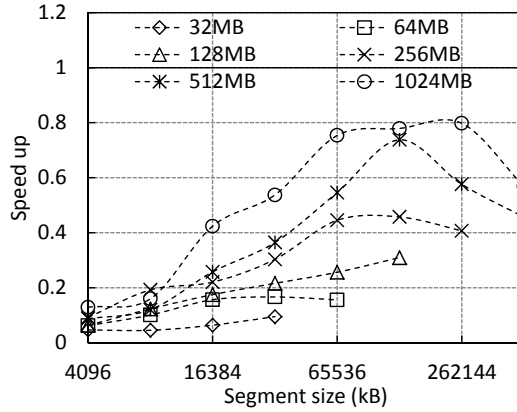


FIGURE 5.11: Speed ratio against segment size for lossless compression of sparse floating point data for different total data sizes, where software-pipelined data is transmitted into the on-board memory of the FPGA board and the FPGA kernel is executed (an unroll factor of 8 was used here).

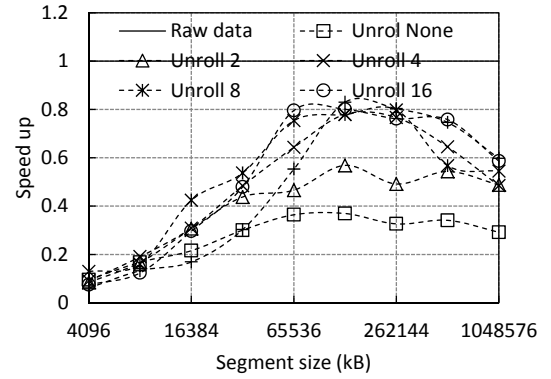


FIGURE 5.12: Speed ratio against the segment size for lossless compression of sparse floating point data for different unroll factors in hardware. The FPGA kernel is executed in this instance.

Figs. 5.13 and 5.14 show the speed ratio observed from using different segment sizes for lossy double precision conversion. The software-only version Fig. 5.13 shows the case where data is simply transmitted into the on-board memory, and a maximum speedup of 1.63x is achieved. Fig. 5.14 shows the case where the FPGA kernel is also executed as the data is transmitted onto the FPGA board.

An interesting observation to note here is that the speedup is faster than 2x (more specifically, 2.09x) in the instance where the kernel is executed as the data is transmitted

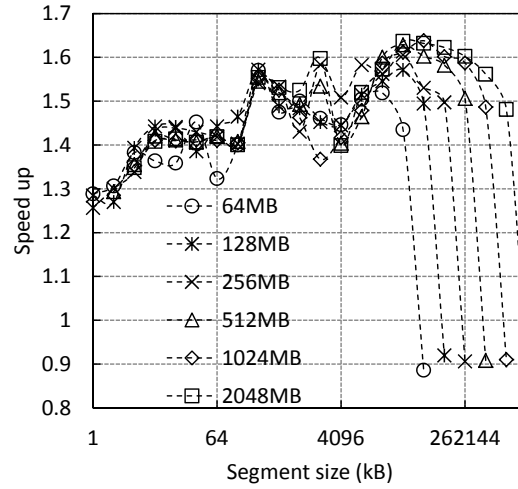


FIGURE 5.13: Speed ratio against the segment size for lossy compression of floating point data for different total data sizes, where software-pipelined data is transmitted into the on-board memory of the FPGA board.

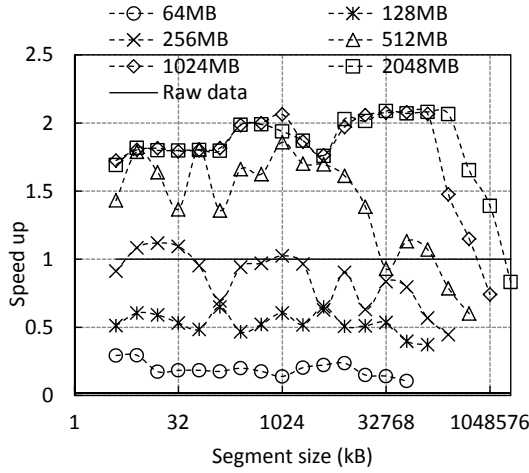


FIGURE 5.14: Speed ratio against the segment size for lossy compression of floating point data for different total data sizes, where software-pipelined data is transmitted into the on-board memory of the FPGA board and the FPGA kernel is executed (a SIMD factor of 8 was used here).

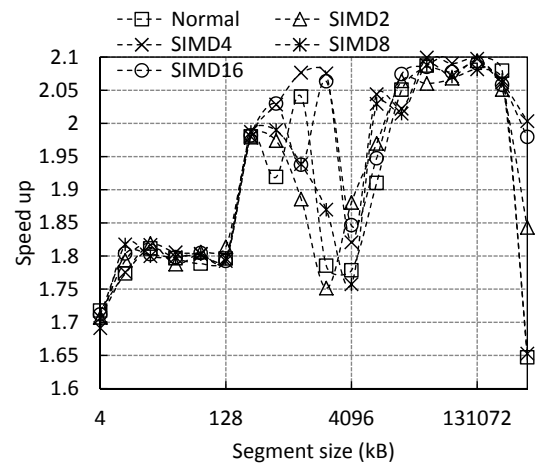


FIGURE 5.15: Speed ratio against the segment size for lossy compression of floating point data for different SIMD factors, where software-pipelined data is transmitted into the on-board memory of the FPGA board and the FPGA kernel is executed).

to the board, when we would expect the theoretical maximum speedup to be equal to $2x$ (due to the compression ratio of the lossy floating point compression being 0.5). The speedup is achieved at a loss of precision. It would not be possible to achieve that speedup from using software multi-threading alone. It appears that the speedup happens as a result of the compressed data being read on the hardware in a more efficient manner by the hardware decoder – being read by the hardware decoder at more than twice the

speed that the raw data would be read from the memory. This results in an overall speedup of more than 2x across the software and hardware.

Fig. 5.15 shows how changing the SIMD factor on the hardware affects software-pipelined data transmission from the CPU into the FPGA hardware. Again, we note the speedup being more than 2x, for the reasons explained above.

These graphs show the effect that applying compression in the same manner as was done on the Maxeler platform will have on a non-Maxeler based platform. Additionally, the data across the three chapters have shown that the framework can function on both simplex and duplex transmission.

5.6.3.3 Summary of Performance for Varying Segment Size

Since there was only one software compression thread used, many of the experiments did not have speedups. However, there was a discussion about using the model to predict the performance with more software threads, since the model has previously been shown to be relatively accurate. The experiments that did not have speedups are still included for completeness, with the knowledge that there are likely to be speedups if a multi-core CPU platform were available to run them.

The results show that good speedups can be achieved under certain circumstances. It was discussed earlier how Figs. 5.13 and 5.14 showed a notable improvement in performance for lossy double precision conversion, even with a single compression thread, since it is a very fast codec. The improvement in performance is achieved at the loss of precision. These results show that compression is viable in these circumstances. The software-only version (Fig. 5.13) shows the case where compressed data is transmitted into on-board memory and it shows a 1.63x speedup over transmitting raw data. The one with software transmission and hardware execution showed a 2.09x speedup, which is faster than the theoretical 2x speedup. It would appear that this is because of the memory accesses to the compressed data in hardware by the decompressor are more efficient than the memory accesses to raw data by the kernel.



FIGURE 5.16: Decoding compressed data in the onboard memory of the FPGA card

5.6.4 Memory Compression

While the machine being used in this experiment is not suitable for software multi-threading, it would still be useful for FPGA applications where data is written into the onboard RAM once and then processed multiple times by the FPGA kernel. The onboard memory of the FPGA card has two memory banks, each with a bandwidth of 25600MB/s. In the case of a single kernel accessing the memory, there would not be any bandwidth bottleneck, but the presence of multiple kernels on the FPGA can bottleneck the memory bandwidth, and so it is useful to investigate the how the hardware throughput of compression is affected by various optimisations.

Fig. 5.16 shows the data transmission phase that is investigated in this Memory Compression section. The hardware implementation of this lossless floating point compression algorithm does not parallelise well in the OpenCL implementation due to random memory writes when compressing, so it does not produce very useful results. Therefore, this section focuses on using memory decompression to access compressed data in memory.

When implementing the compression on hardware using OpenCL, a large design space had to be explored with various ways to increase or decrease the level of parallelism, such as work items, loop unrolling and compute units. The following subsections contain information about how the hardware was optimised to achieve a high throughput. Brute-forcing all combinations of parameters would be extremely time consuming and so it is not a practical option.

5.6.4.1 Lossless Floating Point

One of the main factors to investigate is the effect that loop unrolling has on the decompression algorithm. This was tested on 1GB of floating point data with different levels of sparsity. A work size of 1 was used as it was found to have the best performance (more detail of how this decision was made is included further ahead).

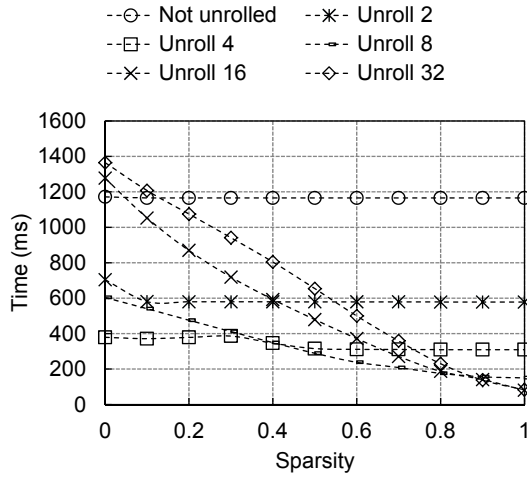


FIGURE 5.17: Graph showing decompress FPGA kernel time for lossless floating point as the data sparsity increases (for different unroll factors and a work size of 1)

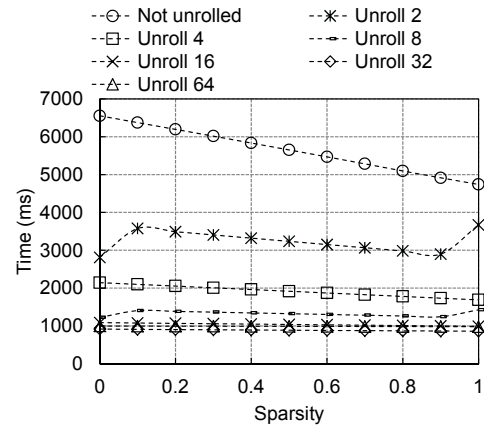


FIGURE 5.18: Graph showing decompress GPU time for lossless floating point as the data sparsity increases (for different unroll factors and a work size of 1)

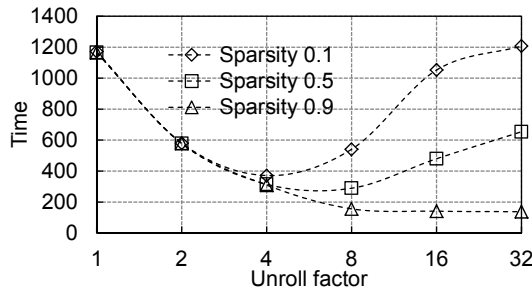


FIGURE 5.19: Graph showing decompress FPGA kernel time for lossless floating point (for low, medium and high sparsity) as unroll factor increases (Work size: 1)

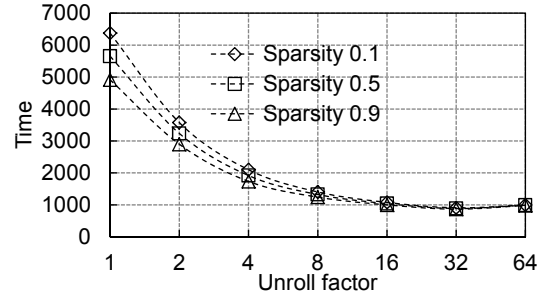


FIGURE 5.20: Graph showing decompress GPU time for lossless floating point (for low, medium and high sparsity) as unroll factor increases (Work size: 1)

Fig 5.17 and Fig. 5.19 show the measurements from two different perspectives: Increasing sparsity and increasing unroll factor. It is interesting to note that highly unrolled implementations perform very poorly on dense data. Profiling data seems to indicate that scratchpad performance of reading input data is efficient where loop unrolling is not used, but scratchpad performance is very poor on dense data where loop unrolling is used. This is not surprising since there will be large amounts of random accesses to memory in a highly unrolled implementation, so that data in scratchpads will likely be replaced before they are accessed again. For sparse data, performance is generally good at any level of loop unrolling, and profiling data seems to suggest that scratchpad performance is efficient for sparse data, regardless of the level of loop unrolling.

The same experiment was then performed on a GPU, and the results are shown in Fig

5.18 and Fig. 5.20. On GPU, loop unrolling shows a clear improvement in performance all across levels of data sparsity.

While loop unrolling was found to provide the fastest performance, other optimisations were also investigated. The lossless floating point algorithm uses a bitmask where each bit specifies whether a certain element of data is zero or not. Due to the bitmask, it makes sense to attempt to parallelise the algorithm along the bitmask, using a work size of more than 1. In this case, a work size of 32 was chosen since the bitmask was interpreted as an array of 32 bit integers (it can be interpreted as 16 bit or 8 bit integers too, and it would make no difference). Using SIMD work items were then attempted, but it was found not to produce any useful improvement in performance. Therefore, this was not investigated further for this codec and its results have not been included here.

Lossless floating point was able to achieve a faster performance when 1 single work item was used as opposed to using 32 work items. The read memory accesses in lossless floating point are large random but the writes are sequential since the exact write location for each element of data is known while the exact read location is initially unknown. It would appear that as the read memory accesses become more random, parallelising memory accesses through the use of work items is not efficient. However, parallelising by the use of loop unrolling seems highly efficient possibly because that could cause data to be read from and written contiguous chunks of memory as the kernel works its way through the entire set of data. The use of local memory was attempted but it did not result in a major improvement in performance. This is discussed later.

5.6.4.2 Lossy Double Precision

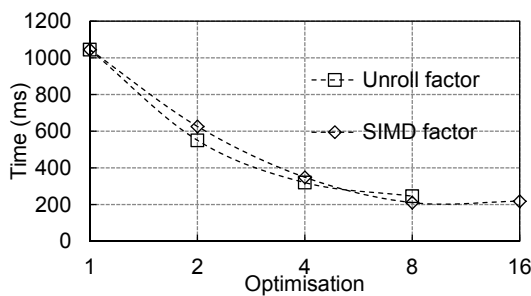


FIGURE 5.21: Graph showing the kernel time of a lossy double precision codec on an FPGA as the unroll and SIMD factors increase (Work size: 1024)

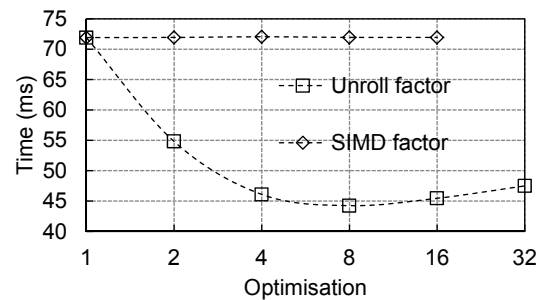


FIGURE 5.22: Graph showing the kernel time of a lossy double precision codec on a GPU as the unroll and SIMD factors increase (Work size: 1024)

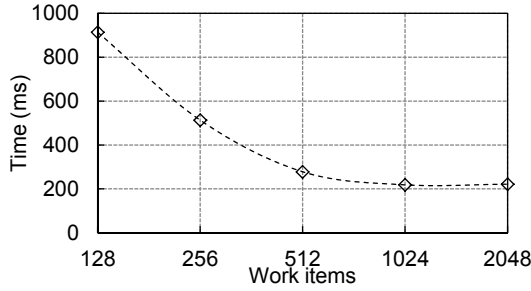


FIGURE 5.23: Graph showing the kernel time of a lossy double precision codec on an FPGA as the work size increases (Work size: 1024)

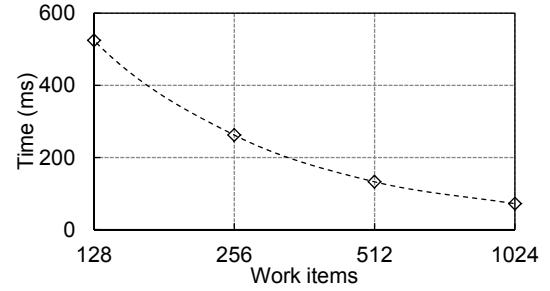


FIGURE 5.24: Graph showing the kernel time of a lossy double precision codec on GPU as the work size increases (Work size: 1024)

The same experiments were then performed on a lossy double precision conversion codec, albeit there is no data sparsity factor involved here. Figs 5.21 and 5.22 show the performance of lossy double precision for 2GB of data on an FPGA and 1GB on a GPU as the loop unroll factor and SIMD factor increases. Due to the limited on-card memory available to the GPU, the data size had to be limited to 1GB. The inconsistency in the data size for the two different versions resulted because the decision to perform experiments on a GPU was taken after the experiments were already performed on the FPGA with 2GB of data.

The work size is fixed at 1024 as this was observed to have the best performance. For both the FPGA and the GPU, the time taken decreases as the unroll factor increases; when the unroll factor becomes too high, the time begins to increase again – which is expected since the onboard memory will not have a high enough bandwidth to supply the data to a kernel with such a high level of parallelism from unrolling. It is also notable that the GPU is actually faster than the FPGA in this particular case.

Varying the SIMD work items did result in a slightly better improvement than changing the unroll factor on the FPGA, unlike for the lossless floating point compression algorithm where using SIMD work items had little positive impact. There appears to be no improvement when SIMD work items are used on the GPU.

The next optimisation to be discussed is the work size. The total data size is 2GB for the FPGA and 1GB for the GPU; and we do not use any loop unrolling or set a SIMD work items factor. Both the FPGA and GPU show an improvement in the time taken to run the decoder kernel in Figs 5.23 and 5.24. The GPU only supports up to 1024 work items which is why the graph for the GPU stops at 1024.

5.6.5 Resource Usage

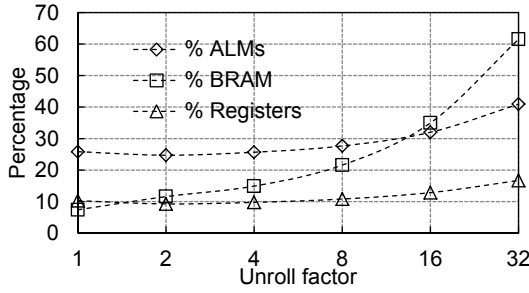


FIGURE 5.25: Graph showing percentage of FPGA resources used as the unroll factor increases for lossless floating point (Work size: 1)

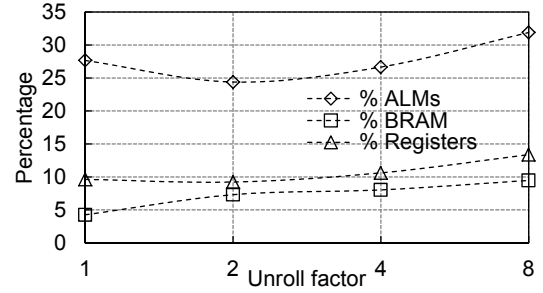


FIGURE 5.26: Graph showing the resource usage as the unroll factor increases for the lossy double precision codec (Work size: 1024)

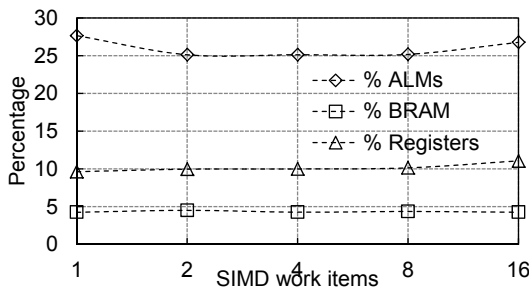


FIGURE 5.27: Graph showing the resource usage of a lossy double precision codec as the SIMD factor increases (Work size: 1024)

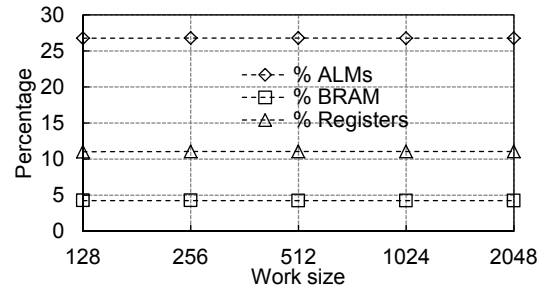


FIGURE 5.28: Graph showing the resource usage of a lossy double precision codec as the work size increases with no unrolling and a SIMD factor of 16.

While loop unrolling will drastically improve the performance of an FPGA hardware implementation, it is also resource intensive. Since one of the main arguments to support this framework is that it uses spare resources to enhance performance, we cannot have a codec that uses up excessive chip area, since there will need to be space for the existing computation kernel on the FPGA.

Fig 5.25 shows the resource usage for different amounts of loop unrolling. For example, it shows that at an unroll factor of 32, around 60% of the BRAM on the chip is consumed. If an unroll factor of 32 were to be used, it would only be appropriate to apply the framework to an existing FPGA kernel which does not use a large amount of block RAM, unless we are using a much larger FPGA with a larger amount of resources.

The graphs suggest that resource usage is slightly higher when there is no unrolling, than when there is an unroll factor of 2. It would appear that the compiler performs various optimisations when unrolling the loop.

As was done in the previous experiment, it is necessary to investigate how resource usage is affected when the codec is parallelised through loop unrolling. Fig 5.26 shows the change in resource usage as the unroll factor increases. As was observed in the lossless floating point case, ALM and register usage are higher when there is no unrolling compared to when there is an unroll factor of 2. Besides that, the resource usage increases, albeit quite slowly. Fig. 5.26 clearly shows that the percentage of ALMs and BRAM used increase by only small percentage when the loop is unrolled.

Fig 5.27 shows a minimal change in resource usage as the SIMD factor increases compared to the unroll factor increasing.

Fig 5.28 shows that the increase in resource usage is negligible as the work size increases. If this is combined with a specified SIMD factor as was done earlier, very fast computation can be achieved.

5.6.5.1 Memory Bandwidth vs Resource Usage: Lossless Floating Point

We can make some further observations about the relationship between resource usage and decompression bandwidth on Figs. 5.29, 5.30 and 5.31. These are scatter plots where both variables are independent of each other and dependent on the unroll factor. Bandwidths measured using a sparsity of 0.1, 0.5 and 0.9 are included. All three graphs show that at high data sparsities (which the compression algorithm is designed to process), the bandwidth is high when there are a large amount of FPGA resources being used. However, we cannot allow the framework to consume all the resources on the FPGA as there needs to be room available for the existing computation kernel. Hence there is an important trade-off between resource usage and bandwidth.

5.6.5.2 Memory Bandwidth vs Resource Usage: Lossy Double Precision

As was done in the previous experiments, we make further observations about the relationship between the resource usage and the on-board memory bandwidth. This is illustrated by Figs 5.32, 5.33, and 5.34. The general trend observed is that the bandwidth increases with resource usage.

We can make some further observations about the relationship between the resource usage and decode bandwidth as the number of SIMD work items change. The scatter

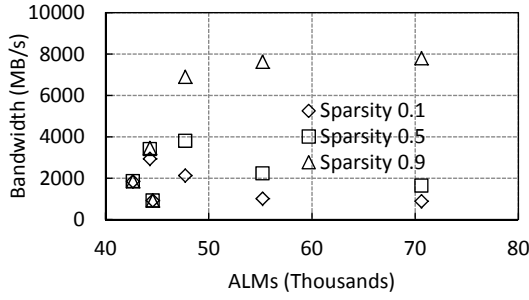


FIGURE 5.29: Graph showing the variation in bandwidth with ALM usage for lossless floating point (for different unroll factors and a work size of 1)

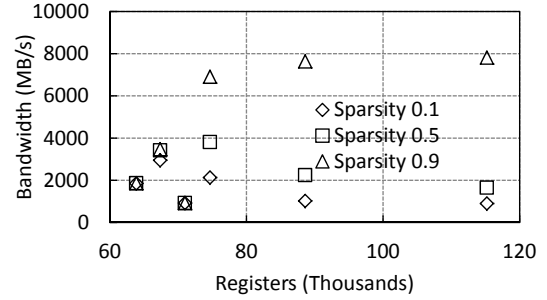


FIGURE 5.30: Graph showing the variation in bandwidth with register usage for lossless floating point (for different unroll factors and a work size of 1)

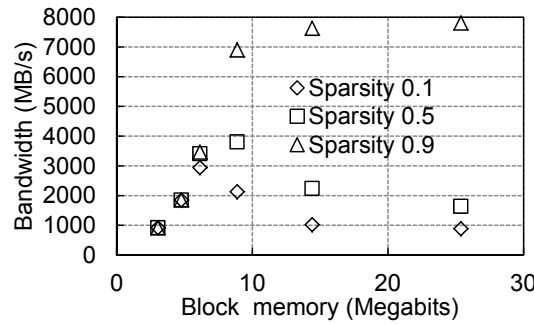


FIGURE 5.31: Graph showing the variation in bandwidth against the usage of block memory bits for lossless floating point (for different unroll factors and a work size of 1)

plots appear to show a steeper increase in bandwidth as the resource usage increases when the SIMD factor is varied.

The pattern between the bandwidth and the resource usage is peculiar compared to the other graphs we have seen so far. Figs 5.32 and 5.33 show that an improvement in performance can be made without using additional ALMS or registers if the work size is increased.

5.6.6 Summary of Parameters

Table 5.1 shows how lossless floating can be sped up by the use of loop unrolling, on both FPGA and GPU. The table contains some of the data that were shown in the graphs above. The values were measured for a total of 1GB data, with a sparsity of 0.9, since these are the type of conditions where it would be suitable to apply compression.

When determining the best parameters, the first step was to establish the best work size to use. Using a work size of 1 was found to have better performance than using

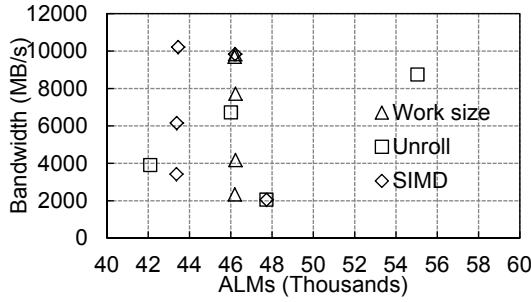


FIGURE 5.32: Graph showing the memory bandwidth against ALM usage for the lossy double precision codec. The work size is varied for a fixed SIMD factor of 16, and the unroll and SIMD factors are varied for a fixed work size of 1024

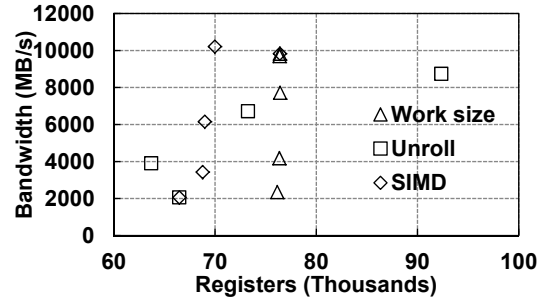


FIGURE 5.33: Graph showing the memory bandwidth against register usage for the lossy double precision codec. The work size is varied for a fixed SIMD factor of 16, and the unroll and SIMD factors are varied for a fixed work size of 1024

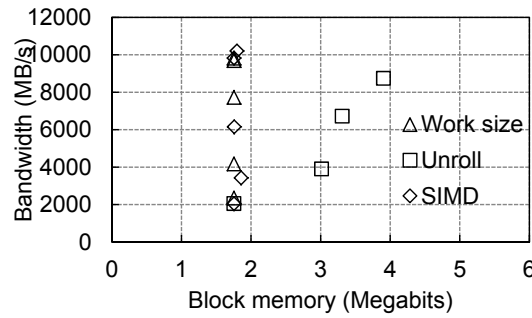


FIGURE 5.34: Graph showing the memory bandwidth against the usage of block memory bits for the lossy double precision codec. The work size is varied for a fixed SIMD factor of 16, and the unroll and SIMD factors are varied for a fixed work size of 1024

multiple work items on the FPGA. With a work size of 1, SIMD cannot be used on the FPGA, and loop unrolling was used to increase the speed. From the table, it is clear that unrolling up to a factor of 8 has a significant improvement in speed, without a significant increase in resource usage. Unrolling any further results in relatively smaller increases in speed, but relatively larger increases in resource usage.

On the GPU, a work size of 32 was observed to have the best performance. Loop unrolling and SIMD were both tested on the GPU, but only loop unrolling showed an increase in the speed.

The bandwidth is the speed at which decompressed data is written out to global memory from the decompressor. A decompression bandwidth of 7805MB/s was achieved at an unroll factor of 32.

Table 5.2 shows some of the cases where the best performance is achieved. Initially, the work size was gradually increased to determine the optimum work size for this algorithm.

TABLE 5.1: Table showing a summary of the parameters for lossless floating point.

HW	Time (ms)	BW (MB/s)	% ALMs	Regs	% BRAM
FPGA Original	1166	922	25.8	70995	7.38
Unroll 2	578	1857	24.7	63888	11.6
Unroll 4	309	3470	25.7	67357	14.9
Unroll 8	156	6904	27.7	74661	21.6
Unroll 16	141	7629	40.0	88608	35.0
Unroll 32	138	7805	40.9	115221	61.5
GPU	4914	226.6	-	-	-
Unroll 2	2897	218.5	-	-	-
Unroll 4	1734	370.7	-	-	-
Unroll 8	1242	619.3	-	-	-
Unroll 16	988.8	864.6	-	-	-
Unroll 32	865.3	1241	-	-	-

TABLE 5.2: Table showing a summary of the parameters for lossy double precision.

HW	Time (ms)	BW (MB/s)	% ALMs	Regs	% BRAM
FPGA Original	1045	2055	27.7	66467	4.25
Unroll 2	550.5	3901	24.4	63697	7.31
Unroll 4	319.7	6717	26.7	73265	8.03
Unroll 8	245.9	8733	31.9	92353	9.47
SIMD 2	626.3	3429	25.1	43365	4.50
SIMD 4	349.0	6154	25.1	43374	4.27
SIMD 8	210.4	10208	25.2	43453	4.36
SIMD 16	218.4	9832	26.8	46204	4.26
GPU	71.92	14930	-	-	-
Unroll 2	54.81	19591	-	-	-
Unroll 4	46.09	23295	-	-	-
Unroll 8	44.24	24270	-	-	-
SIMD 2	71.94	14925	-	-	-

The best performance from the FPGA was achieved at a work size of 1024; the GPU only supports a maximum work size of 1024, and it also had the best performance. After that, loop unrolling and the SIMD factor were both varied. The FPGA showed a better speed when the SIMD factor was changed, and it also used lower resources compared to loop unrolling. The GPU had a better speed with loop unrolling as opposed to the SIMD factor.

As mentioned previously, the GPU had only 1GB of data due to a lack of memory available on the graphics card. The bandwidth is the speed at which decompressed data is written out to global memory from the decompressor.

Lossy double precision is a highly parallel algorithm, where data is read from and written to known memory addresses, because the encoded and decoded sizes are known beforehand. Hence the data accesses are sequential rather than random. Parallelising along the lines of work items (as intended by OpenCL) works very efficiently, and more so than loop unrolling. It achieves faster performance without using additional FPGA resources. It would appear that the work item based computations on global memory are highly efficient for sequential memory accesses on the lossy double precision codec. Due to the high throughput of the kernel, along with the observation that the use of local memory did not benefit lossless floating point, the use of local memory was not attempted for this. Since both the reads and writes to and from memory are done on known, sequential addresses, plus the fact that the same address does not have to be accessed multiple times, it is the main reason why there is not a strong case for using local memory for this codec.

5.6.7 Miscellaneous

On-chip memory and additional compute units were tested on lossless floating point compression and it has been discussed here for completeness. It was tested on one of the implementations with the highest performance – with a work size of 1 and a loop unroll factor of 8. Tables 5.3 and 5.4 show the observed results, and the data obtained from loop unrolling is included for comparison. The ‘original’ data is the measured performance without any loop unrolling or OpenCL constructs that optimise performance.

Additional compute units only had a very small impact on performance and so they were not used for most of the experiments.

Since the memory reads during the hardware decompression of floating point data are read from unknown locations, it can be very random depending on whether or not the data is compressed (the memory read location has to be calculated using the value of the bitmask). Reading from unknown locations in global memory is inefficient, therefore, it was worth testing on-chip memory to see if it would be faster than directly accessing global memory for this particular codec. This involved periodically prefetching a block of compressed data from global (off-chip) memory onto an on-chip buffer, and decompressing the data from the buffer, instead of directly accessing the compressed data in

TABLE 5.3: Table showing how compute units and local memory affect the decompression kernel time in hardware. All times are in milliseconds.

Implementation	Sparsity 0.1	Sparsity 0.5	Sparsity 0.9
Original	1166	1166	1166
2 compute units	1121	1121	1120
4 compute units	1149	1148	1147
Local memory	595.5	595.5	595.6
Unroll 8	540.5	289.2	156.2
Unroll 16	1053	479.5	140.9

TABLE 5.4: Table showing how compute units and local memory affect the global memory write bandwidth in hardware. The write bandwidth is the speed at which decompressed data is transmitted out of the decompressor.

Implementation	Sparsity 0.1	Sparsity 0.5	Sparsity 0.9
Original	921.3	921.4	921.5
2 compute units	958.5	958.6	958.7
4 compute units	935.4	935.8	936.0
Local memory	1803	1803	1803
Unroll 8	1987	3715	6898
Unroll 16	1020	2240	7629

the global memory to decompress it. The data on the buffer is accessed 32 times before it is replaced, once for each bit in the bitmask.

While on-chip memory did improve performance compared to the original version, the improvement was not enough to be of any practical use – although it may be possible to achieve a better performance through a broader and deeper exploration of the design space. Since the speed is the same for both high and low sparsity, the transfer of data through the buffer is the bottleneck. It may be possible to fine tune this implementation to improve its performance further. Whether or not it could outperform loop unrolling remains to be seen.

On-chip was not tested for lossy double precision since the global memory accesses are sequential and are performed on known memory addresses – this allows the compiler to create hardware that performs efficient global memory accesses. Using local memory is likely to increase resource usage while providing little benefit – plus the improvement that may be achieved may be insignificant compared to the time it would take to explore the design space to achieve this improvement. The memory addresses that are read from the lossless floating point algorithm are unknown before they are calculated based on

the value of the bitmask – this reduces the efficiency of the memory accesses, hence the performance is very poor at high levels of sparsity.

5.7 Conclusion

The framework has been extended to the OpenCL platform and its generality has been demonstrated by applying a lossy and a lossless compression codec. The framework was shown to improve both the transmission time across the PCIe channel, and also improve memory read times when compressed data is read from memory as opposed to reading raw data. These improvements were achieved by exploring the broad design space of OpenCL, including experimenting with optimisations such as loop unrolling, SIMD, and work size.

Additionally, improvements were observed for heterogeneous CPU-GPU applications as well as FPGA applications. OpenCL implementations are fundamentally different from Maxeler, since they are heterogeneous, but not streaming.

The results have also demonstrated the generality of the model, since it is able to calculate the data transmission time relatively accurately on a platform other than Maxeler.

It was not possible to do a deeper investigation of software multithreading due to the limitations of the machine (only two physical CPU cores), so there is scope for further investigation. Many of the speedups observed are below 1 as a result of there not being enough software threads – therefore, the software compression on a single thread cannot operate at line rate. The data was included because it is still useful to show that framework functions on OpenCL – and the model showed that with more threads, there would be a speedup. For lossless compression, line rate can be achieved by using 3 or more software threads – which was demonstrated not only by the model, but also by the measured results in Chapters 3 and 4.

To compare how well this framework works with Maxeler and OpenCL, both have their relative advantages and disadvantages. Maxeler’s Low Latency interface operates in an inherently streaming manner, therefore it is well suited to apply line-rate compression to data as it is being transmitted. Much of the high-level primitives offered by Maxeler to implement streaming kernels are optimised for a high throughput. The major drawback

is the relatively limited flexibility that it offers when it comes to flow control and bus widths. Additionally, when Maxeler performs operations on onboard memory, there is a granularity of 384 bytes, which would make it difficult to apply compression to memory operations. OpenCL for the most part, does not have inherent streaming functionality, since OpenCL applications typically operate on data in external memory of the hardware accelerator. It has limited streaming functionality in the form of pipes and more recently, host pipes. However, it offers significantly flexibility for kernels that perform operations to data in memory, and this allows compression and decompression to be optimised to achieve a high throughput. With further development on the streaming functionality of OpenCL, it could become a more suitable platform for the framework than Maxeler.

Chapter 6

Publications

Using High-Level Knowledge to Enhance Data Channels in FPGA Streaming Systems [64] – Marlon Wijeyasinghe and David Thomas – DOI: 10.1109/FPL.2014.6927381 – PhD forum paper in the proceedings of the International Conference on Field Programmable Logic and Applications 2014. This was the original proposal at the start of this PhD project, and so the entire thesis is about this fundamental idea, though a lot has changed through the progression of the project.

A Multi-codec Framework to Enhance Data Channels in FPGA Streaming Systems [45] – Marlon Wijeyasinghe and David Thomas – DOI: 10.1007/978-3-319-30481-6_17 – Best Paper Award at the International Symposium on Applied Reconfigurable Computing 2016. The contents of this paper are discussed in Chapter 3 of the thesis.

Combining hardware and software codecs to enhance data channels in FPGA streaming systems [65] – Marlon Wijeyasinghe and David Thomas – DOI: 10.1016/j.micpro.2017.05.003 – Journal Paper in a special issue of Microprocessors and Microsystems. This paper is an extended version of the original paper published in the proceedings of the International Symposium on Applied Reconfigurable Computing 2016. The contents of this paper are discussed in Chapter 3 of the thesis.

Chapter 7

Conclusion

This thesis has examined techniques to improve the performance of bandwidth-bottlenecked heterogeneous systems by applying compression to the data channels between the software processor and the hardware accelerator. Additionally, it has aimed to simplify the challenge of simultaneously obtaining line rate performance on compression algorithms on both software and hardware to achieve benefits in performance to streaming applications. In this chapter, the main contributions of the thesis are discussed and proposals for future work that could expand on the work presented in this thesis are discussed in section 7.1.

The contributions of this thesis have been discussed in detail in the previous chapters. To reiterate the contributions of this thesis, they are as follows:

- Chapter 2 provided a background on existing implementations of many different compression algorithms on various platforms, to different applications. The motivations for the framework, along with how it addresses some of the issues that are not addressed by existing work are discussed.
- Chapter 3 proposed a multi-threaded framework to apply compute-intensive codecs to a segmented data stream by using extra resources in software and hardware for high-bandwidth applications. The framework was implemented on a commercial platform using C code on the CPU side and OpenSPL code on the FPGA side. A model was proposed for predicting the end-to-end transmission latency when the framework is applied to a streaming system. The accuracy of the model was

compared against measured values when the performance of the concrete implementation was tested by applying compression codecs to a heterogeneous system with a bandwidth-limited PCI-express channel. The framework was tested on a streaming FFT application.

- Chapter 4 proposed a modified and adaptive version of the above framework. By adaptive, we mean that the codec being used to transmit data changes at run time depending on the system performance. The results showed that switching the compression technique at run-time can be beneficial in cases where the nature of the data can change, especially at times when the data is incompressible.
- Chapter 5 extended the framework where it was then implemented on an OpenCL platform and its performance when transmitting data across a bandwidth-bottlenecked PCI-express channel was measured. A variation of the mathematical model developed in Chapter 3 was created to predict the performance of the framework on this new platform and its accuracy compared against measured values.

Throughout the chapters, the results have shown that there can be a notable performance improvement by adding compression to the data channels between the software and the hardware accelerator, by using extra resources in software and hardware. The results have shown that compression can be made to perform at line rate in software using multithreading and in hardware using pipelining. The potential of the framework was demonstrated through the use of a range of different codecs, and it was also tested on a real life application. Furthermore, its generality was demonstrated by applying the framework to two very different platforms. The work showed that using state-of-the-art compression algorithms is not always what is necessary, but simple and fast compression algorithms can achieve decent speedups without significant effort to design sophisticated compression algorithms. By experimenting with the different parameters available, it was shown how the compression can be optimised depending on the type of data being compressed and the size of the data before being compressed.

While the physical bandwidth of data channels are improving (which may appear to make compression less useful), computational capability is increasing at a much faster rate, so the work in this thesis would be useful for future applications as well as present

applications. As the FPGAs become larger, compression will become increasingly necessary for many applications to make use of the full computational capabilities of newer technologies.

By investigating two platforms, the relative advantages and disadvantages of each platform became clear. If the work were to be done afresh, the main point that would be done differently is to investigate multiple platforms at the beginning. When the project was originally done, several platforms were considered and one was selected (Maxeler), and OpenCL then used as an extension to the work. Due to the proprietary nature of Maxeler, specific optimisations to flow control are not always possible, though the platform offers a very high throughput. Handling pipeline stalls and the fixed granularity of Maxeler is a major challenge. OpenCL gives greater control over optimisations. It is also worth investigating the OpenCL platform in more depth, since it offers a large amount of flexibility, and the full extent of this flexibility is worth exploring.

While this project has a number of good achievements, there are also a number of limitations. The main limitation of this work would be the power consumption of such a framework. The larger power consumption of multiple software threads was not considered in this investigation. This may mean that in applications where power supply is limited, the method discussed in this thesis may not be suitable. Moreover, there are questions about the extra heat produced by all this computation, and how much energy will be required to cool a data centre. Furthermore, while there are many streaming applications where this framework would have useful benefits, the benefits offered by this framework would be reduced in applications where data transmission is not happening all the time – for example, where large amounts of data are transmitted to the hardware and multiple iterations are performed on the data (and during this point, data is not being transmitted). In many applications, the spare resources available are likely to be limited, and the availability of resources is an important factor in determining whether the framework can be used. In such cases, the full capability of the framework may not be utilised if there are insufficient resources available for multiple codecs.

There is indeed, room for improvement. The impact of this work could be increased further if a wider range of codecs are tested and if the framework had implementations for more different platforms, as this would thoroughly demonstrate its generality. Some of the work in this thesis, that has not yet been published in conferences and journals

will be submitted for publication. If the framework were to be implemented on other platforms, each platform is likely to present its own unique set of challenges.

Despite the limitations of the work discussed in this thesis, the work presented here has demonstrated that there are indeed viable benefits in streaming applications, and this work has made progress in tackling the bandwidth bottleneck that many computing applications that use hardware accelerators face. Hopefully, the work done in this thesis will contribute towards a general improvement in the communication between the computational elements of heterogeneous systems, and it may further be applied in general networking applications. The work discussed in this thesis could be of interest to: FPGA vendors; HPC programmers using heterogeneous platforms; providers of low-level IP services, developers of streaming compilers.

7.1 Future Work

The framework has been tested on two different systems and performance improvements have been observed on both – one of the systems being a commercial platform that has been intentionally designed for heterogeneous streaming applications and the other being an open platform that is intended for general heterogeneous applications. Having shown positive results on two unrelated systems, it is worth investigating further to see how applications on other heterogeneous platforms will be affected by using the framework described in this thesis.

With the addition of host pipes to OpenCL, there is scope for further investigation into OpenCL being used in a streaming manner. Comparisons could be made about which platform is optimum for streaming, and which platform is the best for using the framework discussed in this report.

7.1.1 Power

During this project, performance of heterogeneous systems were measured using various metrics, but power was not of the metrics that were investigated. The project will benefit from an investigation on the power overhead of the framework, especially when using multiple software threads to perform compression. The power overhead of individual

and multiple codecs on the FPGA could also be investigated, though this is expected to have a much lower impact than the use of multiple software threads.

The higher power consumption from a larger number of software threads will indeed produce more heat. Then there will be further power required to cool the data centre where the servers will be located. It is worth investigating the the total increase in power consumption against the speed up achieved, and compare it against other methods to increase the speed and their power consumption.

In Chapter 4, multiple compression algorithms operated in parallel. Since applying multiple compression techniques in software to every segment would be very power intensive, this would not be suitable for a commercial implementation. In a commercial implementation, the compression ratio of different codecs, could be tested on every 128, or perhaps every 1024 segments, instead of testing on every segment. This would reduce the power consumption of running multiple software threads. If the compression ratio is only checked once in a while, then partial reconfiguration in hardware could also be used, if multiple codecs are to be switched on and off as required, instead of them all occupying resources on the hardware even if only one of the codecs are active at any particular time. This would reduce the amount of resources used by the adaptive framework and allow it to be used for other purposes, such as the existing computation kernel on the FPGA.

7.1.2 Codecs and Applications

While a number of specific codecs have been investigated in this project, there is scope for further investigation into other codecs.

For example, run-length encoding is a general-purpose codec that can potentially be performed fast enough to operate at line rate. Other codecs such as YUV image compression could also be investigated. These would be suitable codecs since their simplicity could potentially allow them to operate at line rate and they are unlikely to occupy a large amount of FPGA resources.

The adaptive compression discussed in Chapter 4 has room for further improvement – currently the type of compression is decided based on the compression ratio, but the adaptivity could be enhanced to consider the amount of data in the queue to be

transmitted, to determine which type of compression to use. This would have similarities with PACM [38], which was discussed in the Background chapter. It was also mentioned in the conclusion of Chapter 4, how testing the compression ratio on every segment is not realistic in a commercial implementation – instead, the compression ratio should be checked only occasionally to reduce power consumption. Additionally, adaptivity could be further extended to work for lossy compression of floating point and/or fixed point data. In such a case, precision may be altered depending on the error tolerance of the application. If this were to be tested, Monte Carlo simulations would be a suitable application to test, since the algorithm is designed to filter out elements of data with greater error.

Besides codecs, there is also scope for investigation into more real-life applications that could benefit from this framework. The framework was applied to an FFT application, but the effect of this framework on other applications that handle large quantities of data could be investigated. For example, dense matrix multiplication is an application that may benefit from compression due to the large amounts of data it often handles. Machine learning applications could benefit from the framework since they often process and transmit large amounts of data.

Compression does not have to be the only type of codec that is applied through such a framework. It is worth investigating whether encryption codecs can be applied at line rate using the ideas proposed in this thesis to improve the security of sensitive data. This may be particularly useful on a data centre network such as Catapult C, where data is transmitted between servers via network switches. This would require investigation of the current difficulty or ease of intercepting data that is transmitted within a data centre network.

7.1.3 Automatic Optimisation of Parameters

When using segmented data transmission with the codecs, there are a large number of parameters involved and determining the optimum parameters manually is tedious and time consuming. Many of the experiments have shown how changes to the parameters can have major impacts on the performance of system running under the framework discussed in this thesis. The experiments have determined what the optimum parameters

are, and showed that the framework can still provide a speed-up as long as the parameters are close to the optimum.

An automated exploration of the design space could explore different parameters and combinations of parameters in greater depth. This could be used to fine tune the performance if other codecs were to be investigated, instead of tuning them manually.

The current implementation of the framework requires effort from the programmer to include the codecs. A potential improvement will be whether the compiler could intelligently determine where it would be appropriate to insert compression codecs, and then apply them automatically, without effort on the programmer's side. There is already some degree of automation where the type of codec can be switched at run-time.

7.1.4 Other Streaming and Non-Streaming Interfaces

In this thesis, CPU-FPGA, CPU-GPU and FPGA-RAM interfaces were investigated. The focus of this framework was on heterogeneous systems, but the framework could easily be applicable for homogeneous systems too.

It is worth applying this framework to homogeneous systems and compare its performance against other frameworks that are specifically designed for homogeneous systems. We could determine the cases where the framework discussed in this thesis could be useful for homogeneous applications. An example would be the Latency-insensitive Environment for Application Programming (LEAP) [66], which is an operating system that allows designs to be mapped to various FPGA platforms and it focuses on latency-insensitive data transmission.

Additional heterogeneous systems, such as Intel's HARP would be very useful to investigate. Since this is a system with shared memory between the host processor and the hardware accelerator, memory compression could have improvements in performance.

In the background section, networking applications such as Catapult C was discussed. Since such a data centre network involves transmitting large amounts of data between servers, there is a strong case for assessing the impact that the compression techniques discussed here could be used to accelerate data transmission between servers.

Bibliography

- [1] F. Douglass. On the Role of Compression in Distributed Systems. *SIGOPS Oper. Syst. Rev.*, 27(2):88–93, April 1993. ISSN 0163-5980. doi: 10.1145/155848.155864. URL <http://doi.acm.org/10.1145/155848.155864>.
- [2] Maxeler Technologies. MaxCompiler White Paper. Technical report, 2011.
- [3] Maxeler Technologies. Dataflow Programming for Networking. Technical report, 2013.
- [4] Nallatech. Nallatech 385 Altera Stratix V FPGA Accelerator Card. Technical report.
- [5] K. Sano, K. Katahira, and S. Yamamoto. Segment-Parallel Predictor for FPGA-Based Hardware Compressor and Decompressor of Floating-Point Data Streams to Enhance Memory I/O Bandwidth. In *Data Compression Conference (DCC), 2010*, pages 416–425, Los Alamitos, CA, USA, mar 2010. IEEE Computer Society. doi: 10.1109/DCC.2010.44. URL <https://doi.ieeecomputersociety.org/10.1109/DCC.2010.44>.
- [6] K. Katahira, K. Sano, and S. Yamamoto. FPGA-based Lossless Compressors of Floating-point Data Streams to Enhance Memory Bandwidth. In *21st IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2010*, pages 246–253, July 2010. doi: 10.1109/ASAP.2010.5540973.
- [7] P. Grigoras, P. Burovskiy, E. Hung, and W. Luk. Accelerating SpMV on FPGAs by Compressing Nonzero Values. In *23rd IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2015*, Sept 2015.

- [8] Z. Wang, T. Zhang, L. Yan, and C. Gong. A High Performance Fully Pipelined Architecture for Lossless Compression of Satellite Image. In *International Conference on Multimedia Technology (ICMT)*, 2010, pages 1–4, Oct 2010. doi: 10.1109/ICMULT.2010.5631429.
- [9] T. Damak, S. Bilavarn, and N. Masmoudi. HLS Based Design of a Mixed Architecture for H.264/AVC CAVLD. In *IEEE 12th International Multi-Conference on Systems, Signals Devices (SSD15)*, 2015, pages 1–4, March 2015. doi: 10.1109/SSD.2015.7348095.
- [10] D.C. Zaretsky, G. Mittal, and P. Banerjee. Streaming Implementation of the ZLIB Decoder Algorithm on an FPGA. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2009, pages 2329–2332, May 2009. doi: 10.1109/ISCAS.2009.5118266.
- [11] M. A. A. El Ghany, A. E. Salama, and A. H. Khalil. Design and Implementation of FPGA-based Systolic Array for LZ Data Compression. In *IEEE International Symposium on Circuits and Systems*, 2007, pages 3691–3695, May 2007. doi: 10.1109/ISCAS.2007.378644.
- [12] S. Naqvi, R. Naqvi, R. A. Riaz, and F. Siddiqui. Optimized RTL Design and Implementation of LZW Algorithm for High Bandwidth Applications. *Przegld Elektrotechniczny*, R. 87, nr 4:279–285, 2011.
- [13] X. Zhou, Y. Ito, and K. Nakano. An Efficient Implementation of LZW Decompression in the FPGA. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 599–607, May 2016. doi: 10.1109/IPDPSW.2016.33.
- [14] B. Li, L. Zhang, Z. Shang, and Q. Dong. Implementation of LZMA compression algorithm on FPGA. *Electronics Letters*, 50(21):1522–1524, October 2014. ISSN 0013-5194. doi: 10.1049/el.2014.1734.
- [15] X. Zhao and B. Li. Implementation of the LZMA compression algorithm on FPGA. In *2017 International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, pages 1–2, Oct 2017. doi: 10.1109/EDSSC.2017.8126506.
- [16] L. P. Deutsch. DEFLATE Compressed Data Format Specification Version 1.3. 1996. doi: 10.17487/RFC1951. URL <https://www.rfc-editor.org/info/rfc1951>.

- [17] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, and K. Sheng. FPGA Implementation of GZIP Compression and Decompression for IDC Services. In *International Conference on Field-Programmable Technology (FPT)*, 2010, pages 265–268, Dec 2010. doi: 10.1109/FPT.2010.5681489.
- [18] O. Plugariu, A. D. Gegiu, and L. Petrica. Fpga systolic array gzip compressor. In *2017 9th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pages 1–6, June 2017. doi: 10.1109/ECAI.2017.8166387.
- [19] Xillybus. www.xillybus.com. Accessed: 2018-10-26.
- [20] K. Papadopoulos and I. Papaefstathiou. Titan-R: A Reconfigurable Hardware Implementation of a High-Speed Compressor. In *16th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2008, pages 216–225, April 2008. doi: 10.1109/FCCM.2008.14.
- [21] R. Mehboob, S. A. Khan, Z. Ahmed, H. Jamal, and M. Shahbaz. Multigig Lossless Data Compression Device. *IEEE Transactions on Consumer Electronics*, 56(3): 1927–1932, Aug 2010. ISSN 0098-3063. doi: 10.1109/TCE.2010.5606348.
- [22] J. L. Nunez and S. Jones. The X-MatchPRO 100 Mbytes/second FPGA-based Lossless Data Compressor. In *Design, Automation and Test in Europe (DATE)*, 2000, pages 139 – 142, March 2000.
- [23] J. L. Nunez, S. Jones, and S. Bateman. X-MatchPRO: A High Performance Full-Duplex Lossless Data Compressor on a ProASIC FPGA. In *International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, 2001, pages 56–60, 2001. doi: 10.1109/IDAACS.2001.941979.
- [24] J. L. Nunez and S. Jones. Gbit/s Lossless Data Compression Hardware. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(3):499–510, June 2003. ISSN 1063-8210. doi: 10.1109/TVLSI.2003.812288.
- [25] I. Shcherbakov, C. Weis, and N. Wehn. A High-Performance FPGA-Based Implementation of the LZSS Compression Algorithm. In *IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, 2012, pages 449–453, May 2012. doi: 10.1109/IPDPSW.2012.58.

- [26] B. Sukhwani, B. Abali, B. Brezzo, and S. Asaad. High-Throughput, Lossless Data Compression on FPGAs. In *IEEE 19th International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2011*, pages 113–116, May 2011. doi: 10.1109/FCCM.2011.56.
- [27] Y. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei. A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms. In *Proceedings of the 53rd Annual Design Automation Conference, DAC '16*, pages 109:1–109:6, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4236-0. doi: 10.1145/2897937.2897972. URL <http://doi.acm.org/10.1145/2897937.2897972>.
- [28] W. Qiao, J. Du, Z. Fang, M. Lo, M. F. Chang, and J. Cong. High-Throughput Lossless Compression on Tightly Coupled CPU-FPGA Platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 37–44, April 2018. doi: 10.1109/FCCM.2018.00015.
- [29] J. Fowers, J. Kim, D. Burger, and S. Hauck. A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM '15*, pages 52–59, 2015. doi: 10.1109/FCCM.2015.46.
- [30] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong. Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, SoCC '16*, pages 456–469, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4525-5. doi: 10.1145/2987550.2987569. URL <http://doi.acm.org/10.1145/2987550.2987569>.
- [31] J. Y. Kim, S. Hauck, and D. Burger. A Scalable Multi-engine Xpress9 Compressor with Asynchronous Data Transfer. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 161–164, May 2014. doi: 10.1109/FCCM.2014.49.
- [32] MS-XCA: Xpress Compression Algorithm. <http://msdn.microsoft.com/enus/library/hh554002.aspx>. Accessed: 2019-10-08.
- [33] Y. Thoma, A. Dassatti, and D. Molla. FPGA2: An Open Source Framework for FPGA-GPU PCIe Communication. In *International Conference on Reconfigurable*

- Computing and FPGAs (ReConFig)*, 2013, pages 1–6, Dec 2013. doi: 10.1109/ReConFig.2013.6732296.
- [34] T. Bicer, J. Yin, D. Chiu, G. Agrawal, and K. Schuchardt. Integrating Online Compression to Accelerate Large-Scale Data Analytics Applications. In *IEEE 27th International Symposium on Parallel and Distributed Processing, 2013*, pages 1205–1216, May 2013. doi: 10.1109/IPDPS.2013.81.
- [35] T. Bicer, J. Yin, and G. Agrawal. Improving I/O Throughput of Scientific Applications Using Transparent Parallel Compression. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2014*, pages 1–10, May 2014. doi: 10.1109/CCGrid.2014.112.
- [36] A. Yang, H. Mukka, F. Hesaaraki, and M. Burtscher. MPC: A Massively Parallel Compression Algorithm for Scientific Data. In *IEEE International Conference on Cluster Computing, 2015*, pages 381–389, Sept 2015. doi: 10.1109/CLUSTER.2015.59.
- [37] P. Christie and D. Stroobandt. The interpretation and application of Rent’s rule. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(6):639–648, Dec 2000. doi: 10.1109/92.902258.
- [38] R. Wang, C. Wang, and L. Zha. PACM: A Prediction-Based Auto-Adaptive Compression Model for HDFS. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016*, pages 1617–1626, May 2016. doi: 10.1109/IPDPSW.2016.100.
- [39] E. R. Schendel, Y. Jin, N. Shah, J. Chen, C. S. Chang, S. H. Ku, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova. ISOBAR Preconditioner for Effective and High-throughput Lossless Data Compression. In *IEEE 28th International Conference on Data Engineering, 2012*, pages 138–149, April 2012. doi: 10.1109/ICDE.2012.114.
- [40] C. Krintz and S. Sucu. Adaptive on-the-fly Compression. *IEEE Transactions on Parallel and Distributed Systems*, 17(1):15–24, Jan 2006. ISSN 1045-9219. doi: 10.1109/TPDS.2006.3.

- [41] I. Shcherbakov and N. Wehn. A Parallel Adaptive Range Coding Compressor: Algorithm, FPGA Prototype, Evaluation. In *Data Compression Conference, 2012*, pages 119–128, April 2012. doi: 10.1109/DCC.2012.20.
- [42] X. Cai and J. S. Lim. Algorithms for Transform Selection in multiple-transform Video Compression. In *19th IEEE International Conference on Image Processing, 2012*, pages 2481–2484, Sept 2012. doi: 10.1109/ICIP.2012.6467401.
- [43] L. Nissenbaum. Globally Optimal Algorithms for Multiple-transform Signal Compression. Master’s thesis, Massachussets Institute of Technology, 2015.
- [44] P. A. H. Peterson and P. L. Reiher. Datacomp: Locally Independent Adaptive Compression for Real-World Systems. In *IEEE 36th International Conference on Distributed Computing Systems (ICDCS), 2016*, pages 211–220, June 2016. doi: 10.1109/ICDCS.2016.106.
- [45] M. Wijeyasinghe and D. Thomas. A Multi-codec Framework to Enhance Data Channels in FPGA Streaming Systems. In *12th International Symposium on Applied Reconfigurable Computing (ARC), 2016*, pages 207–219, March 2016. doi: 10.1007/978-3-319-30481-6_17.
- [46] S. Stojanovic, D. Boji, M. Bojovi, M. Valero, and V. Milutinovi. An Overview of Selected Hybrid and Reconfigurable Architectures. In *IEEE International Conference on Industrial Technology, 2012*, pages 444–449, March 2012. doi: 10.1109/ICIT.2012.6209978.
- [47] M. S. Abdelfattah, A. Hagiescu, and D. Singh. Gzip on a Chip: High Performance Lossless Data Compression on FPGAs Using OpenCL. In *Proceedings of the International Workshop on OpenCL 2013 & #38; 2014, IWOCL ’14*, pages 4:1–4:9, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3007-7. doi: 10.1145/2664666.2664670. URL <http://doi.acm.org/10.1145/2664666.2664670>.
- [48] C. Kim and H.l Shin. A Performance-driven Logic Emulation System: FPGA Network Design and Performance-driven Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(5):560–568, May 1996. ISSN 0278-0070. doi: 10.1109/43.506143.

- [49] S. Muhlbach and A. Koch. A Scalable Multi-FPGA Platform for Complex Networking Applications. In *IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2011, pages 81–84, May 2011. doi: 10.1109/FCCM.2011.26.
- [50] J. Pendlum, M. Leaser, and K. Chowdhury. Reducing Processing Latency with a Heterogeneous FPGA-Processor Framework. In *IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2014, pages 17–20, May 2014. doi: 10.1109/FCCM.2014.13.
- [51] H. Shimura, H. Noda, and H. Amano. C4: An FPGA-based Compression Algorithm for ExpEther. In *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, pages 356–362, Nov 2018. doi: 10.1109/CANDARW.2018.00072.
- [52] Project Catapult. <https://www.microsoft.com/en-us/research/project/project-catapult/>. Accessed: 2019-08-23.
- [53] D. Koch, C. Beckhoff, and J. Teich. Bitstream Decompression for High Speed FPGA Configuration from Slow Memories. In *2007 International Conference on Field-Programmable Technology*, pages 161–168, Dec 2007. doi: 10.1109/FPT.2007.4439245.
- [54] D. Koch, C. Beckhoff, and J. Teich. Hardware Decompression Techniques for FPGA-Based Embedded Systems. *ACM Trans. Reconfigurable Technol. Syst.*, 2(2):9:1–9:23, June 2009. ISSN 1936-7406. doi: 10.1145/1534916.1534919. URL <http://doi.acm.org/10.1145/1534916.1534919>.
- [55] J. Yan, J. Yuan, P. H. W. Leong, W. Luk, and L. Wang. Lossless Compression Decoders for Bitstreams and Software Binaries Based on High-Level Synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2842–2855, Oct 2017. doi: 10.1109/TVLSI.2017.2713527.
- [56] L. Daoud, F. Hussein, and N. Rafla. Real-time Bitstream Decompression Scheme for FPGAs Reconfiguration. In *2018 IEEE 61st International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1082–1085, Aug 2018. doi: 10.1109/MWSCAS.2018.8624003.

- [57] Y. Gao, H. Ye, J. Wang, and J. Lai. FPGA bitstream compression and de-compression based on LZ77 algorithm and BMC technique. In *2015 IEEE 11th International Conference on ASIC (ASICON)*, pages 1–4, Nov 2015. doi: 10.1109/ASICON.2015.7517171.
- [58] D. O’Loughlin, A. Coffey, F. Callaly, D. Lyons, and F. Morgan. Xilinx Vivado High Level Synthesis: Case Studies. In *Irish Signals Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014). 25th IET*, pages 352–356, June 2014. doi: 10.1049/cp.2014.0713.
- [59] K. Rupnow, Y. Liang, Y. Li, and D. Chen. A Study of High-Level Synthesis: Promises and Challenges. In *IEEE 9th International Conference on ASIC (ASICON), 2011*, pages 1102–1105, Oct 2011. doi: 10.1109/ASICON.2011.6157401.
- [60] M. A. Zainol and J. L. Nunez-Yanez. CPCIE: A Compression-enabled PCIe Core for Rnergy and Performance Optimization. In *IEEE Nordic Circuits and Systems Conference (NORCAS), 2016*, pages 1–6, Nov 2016. doi: 10.1109/NORCHIP.2016.7792892.
- [61] B. Zainol, M. Amiruddin, and J. Nunez-Yanez. Extending the PCIe Interface with Parallel Compression/Decompression Hardware for Energy and Performance Optimization. *International Journal on Computer Science & Communication Engineering (IJFRSCE)*, 4(2):405 – 419, 2 2018. ISSN 2454-4248.
- [62] H. Yang, K. Fleming, M. Adler, and J. Emer. Optimizing Under Abstraction: Using Prefetching to Improve FPGA Performance. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–8, Sep. 2013. doi: 10.1109/FPL.2013.6645522.
- [63] Maxeler Technologies. Maxeler AppGallery. appgallery.maxeler.com.
- [64] M. Wijeyasinghe and D. Thomas. Using High-Level Knowledge to Enhance Data Channels in FPGA Streaming Systems. In *24th International Conference on Field Programmable Logic and Applications (FPL), 2014*, pages 1–2, Sept 2014. doi: 10.1109/FPL.2014.6927381.

-
- [65] M. Wijeyasinghe and D. Thomas. Combining Hardware and Software Codecs to Enhance Data Channels in FPGA Streaming Systems. *Microprocessors and Microsystems*, 51:275 – 288, 2017. ISSN 0141-9331. doi: <https://doi.org/10.1016/j.micpro.2017.05.003>. URL <http://www.sciencedirect.com/science/article/pii/S0141933117302351>.
- [66] K. Fleming, H. Yang, M. Adler, and J. Emer. The LEAP FPGA Operating System. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sep. 2014. doi: 10.1109/FPL.2014.6927488.