Parallelization of Variable Rate Decompression for GPU Acceleration

Lennart Noordsij

CE-MS-2019-08

Abstract

Data movement has been long identified as the biggest challenge facing modern computer systems designers. To tackle this challenge, many novel data compression algorithms have been developed. These compression algorithms can be embedded into bandwidth-bound applications to reduce their memory traffic volume. As a result, data decompression, in many instances, is in the critical path of the application execution, while the compression itself can happen offline or outside of the critical path. Therefore, fast data decompression is of utmost importance. However, most existing parallel decompression schemes adopt a particular parallelization strategy suited for a particular HW platform. Such an approach fails to harness the parallelism found in diverse modern HW architectures. To this end, we propose multiple parallelization strategies for variable rate data decompression. The proposed strategies aim to utilize parallel architectures efficiently. Our strategies are based on generating *extra information* during the encoding phase, and then passing this information in a side-channel to the decoder. After that, the decoder can use that extra information to speed-up the decoding process tremendously. To demonstrate the effectiveness of our strategies, we implement them in a state-of-the-art compression algorithm called ZFP and apply it on a real-life industrial application from ASML. Our implementation is publicly available on GitHub [1]. This application is a feed-forward control model for controlling wafer heat in EUV lithography machines. The application is dominated by matrix-vector multiplication (which is bandwidth-bound) and is executed on GPUs. We show that parallelization strategies suited for multicore CPUs are different from the ones suited for GPUs. On a CPU, we achieve a near-optimal speedup and an overhead size which is consistently less than 0.04% of the compressed data size. On a GPU, we achieve a decoding throughput of more than 130 GiB/s which allows us to execute the ASML application within the given time budget.

Parallelization of Variable Rate Decompression for GPU Acceleration

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Lennart Noordsij
born in Rotterdam, Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Parallelization of Variable Rate Decompression for GPU Acceleration

by Lennart Noordsij

Abstract

Data movement has been long identified as the biggest challenge facing modern computer systems designers. To tackle this challenge, many novel data compression algorithms have been developed. These compression algorithms can be embedded into bandwidth-bound applications to reduce their memory traffic volume. As a result, data decompression, in many instances, is in the critical path of the application execution, while the compression itself can happen offline or outside of the critical path. Therefore, fast data decompression is of utmost importance. However, most existing parallel decompression schemes adopt a particular parallelization strategy suited for a particular HW platform. Such an approach fails to harness the parallelism found in diverse modern HW architectures. To this end, we propose multiple parallelization strategies for variable rate data decompression. The proposed strategies aim to utilize parallel architectures efficiently. Our strategies are based on generating *extra information* during the encoding phase, and then passing this information in a side-channel to the decoder. After that, the decoder can use that extra information to speed-up the decoding process tremendously. To demonstrate the effectiveness of our strategies, we implement them in a state-of-the-art compression algorithm called ZFP and apply it on a real-life industrial application from ASML. Our implementation is publicly available on GitHub [1]. This application is a feed-forward control model for controlling wafer heat in EUV lithography machines. The application is dominated by matrix-vector multiplication (which is bandwidth-bound) and is executed on GPUs. We show that parallelization strategies suited for multicore CPUs are different from the ones suited for GPUs. On a CPU, we achieve a near-optimal speedup and an overhead size which is consistently less than 0.04% of the compressed data size. On a GPU, we achieve a decoding throughput of more than 130 GiB/s which allows us to execute the ASML application within the given time budget.

Laboratory : Computer Engineering
Codenumber : CE-MS-2019-08

Committee Members :

Advisor:

Chairperson: dr. ir. Zaid Al-Ars, QCE, TU Delft

Member: dr. ir. Claudia Hauff, WIS, TU Delft

Member: dr. ir. Mohamed Bamakhrama, Supervisor, Synposys

Member: dr. ir. Peter Lindstrom, Project Leader, LLNL

Dedicated to my parents, girlfriend, family and friends, who have always supported me

Contents

List of Figures

List of Tables

List of Acronyms

COTS Commercial Off-The-Shelf

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DCT Discrete Cosine Transform

EBC Embedded Block Coding

EUV Extreme Ultraviolet

FLOPS Floating-Point Operations

GEMV Generalized Matrix-Vector multiplication

GPU Graphics Processing Unit

HBM High Bandwidth Memory

HW Hardware

HPC High-Performance Computing

IC Integrated Circuit

JPEG Joint Photographic Experts Group

OMP/OpenMP Open Multi-Processing

PCIe Peripheral Component Interconnect Express

(P)VLD (Parallel) Variable Length Decompression

SIMD Single Instruciton Multiple Data

SM Streaming Multiprocessor

WHFF Wafer Heat Feed Forward

List of Notations

C Chunk Size - the number of blocks per chunk

D Offset - the bit position distance from stream start

f Overhead Factor - the size of the side channel information relative to the compressed data

H Hybrid strategy - a strategy that combines the approaches and adcantages of the offsets and lengths strategies

I Overhead Size - the number of bits of the side-channel information

L Block length - the length of a block in number of bits

P Partition Size - the number of blocks per partition

R Compression Ratio - the ratio of compressed data size vs uncompressed data size

$\mathbf{R}_e$ Effective Compression Ratio - the ratio of compressed data size and side channel information overhead vs uncompressed data size

S Data size - the total data size in number of bits

Acknowledgements

There are many people I would like to thank for their help with my work, directly or indirectly. First and foremost Mohamed Bamakhrama for his outstanding supervision. He has helped me develop much faster than I imagined to be possible at the start of this project. His advice on both my thesis project and future options are something I will remember throughout my career. Next Steven van der Vlugt for taking over supervision when needed. His reviews of my work and improvement suggestions have been very valuable and he always proved a great listener for any issues, even when they were not related to this work. Furthermore Zaid Al-Ars for providing me with valuable insights on how to get my message across clearly. Also John Wagensveld for his part in supervising and challenging me and Martijn Berkers for his reviews and help. A special thank you goes to Peter Lindstrom and Matthew Larsen from LLNL. Their support with ZFP and discussion on parallelization strategies proved very valuable throughout my work.

I would like to thank all of my friends and family for their support. They made it possible for me to focus on the work when needed and also refresh my mind when possible. Specifically I would like to thank my mentor Kornel Hoekerd who provided me with valuable advice on difficult topics, which allowed me to stay focused on my work. The same applies to my girlfriend, who was always there to help or listen when I needed it.

Also I would like to thank ASML for providing me with this opportunity, and all of my colleagues. They made my days at ASML a great learning experience and also a great time.

Finally I would like to thank the DOE NNSA ECP project and ECP CODAR project for providing the Scientific Data Reduction benchmarks.

Lennart Noordsij
Delft, The Netherlands
August 12, 2019

1 Introduction

Memory bandwidth and data movement have been long identified as the biggest challenge facing computer systems' designers in the current decade [4]. This challenge becomes even harder in massively data-parallel architectures such as GPUs [5]. In order to overcome this bottleneck, a lot of research has been focusing on novel *online* data compression techniques [6]. In online compression, the data encoding and/or decoding happens at run-time as part of another application. The primary goal of online compression is to reduce the amount of data that the application has to transfer into/from memory at the expense of sacrificing extra compute power to perform the encoding/decoding steps. As the memory bandwidth gap worsens, any savings in data transfer time lead to huge savings in the total application execution time, even with the additional overhead used to encode/decode the data. In contrast to online compression, offline compression focuses solely on reducing, as much as possible, the compressed data size. At first look, both online and offline compression seem to share the same final goal. However, online compression focuses, in addition to reducing the compressed data size, on reducing the overhead of encoding and decoding. To reduce such overhead, many online compression algorithms focus on utilizing modern architectural features such as vector instructions, multicore CPUs, and caches [6].

1.1 The memory wall

Computing applications are often categorized as either *compute-bound* or *bandwidth-bound*. A compute-bound application is one where the majority of the total execution time is spent on calculation operations, such as addition and multiplication. On the contrary, a bandwidth-bound application is one where the majority of the total execution time is spent in memory operations. This classification is hardware platform dependent, as it is based on the time spent on specific operations. Due to the growing gap between memory bandwidth and computation performance, an increasing number of applications is qualified as bandwidth-bound on modern hardware architectures such as GPUs [4][5]. This is often referred to as the "Memory Wall" [7]. Data compression is a solution to speed up these bandwidth-bound applications.

1.2 Data compression

Compression algorithms can be analyzed based on many different metrics. In this work we focus on *compression ratio* and *decoding throughput*. Compression ratio refers to the *reduction ratio* in compressed data size vs. the original data size. For example, if the algorithm compresses a file of 1 MB into 100 KB, then we say that the compression ratio

is 10. Decoding throughput is the amount of data that can be decoded in a given time frame, for example 10 GiB/s.

Compression algorithms can also be classified in many different ways. An algorithm is said to be a *tiled* algorithm if it divides the input data into *tiles* or *blocks* (i.e., groups) of n values and then compresses each tile independently from other tiles. In this work we focus on tiled algorithms, as the independence of tiles allows for tile-level parallelism. A *lossy* compression algorithm is an algorithm where data compression introduces a user controlled error. This is contrary to *lossless* algorithms where the decoded data after decompression is bitwise (exactly) equal to the original data. For lossy compression algorithms the maximum achievable compression ratio is dependent on the error bound, where increasing the error bound allows a higher compression ratio. In this work we focus on lossy algorithms, as they achieve much higher compression ratios than lossless algorithms. *Online* compression means that data encoding and/or decoding happens at run-time as part of another application. The primary goal of online compression is to reduce the amount of data that the application has to transfer into/from memory at the expense of sacrificing extra compute power to perform the encoding/decoding steps. As the memory bandwidth gap worsens, any savings in data transfer time lead to huge savings in the total application execution time, even with the additional overhead used to encode/decode the data. In contrast to online compression, *offline* compression focuses solely on reducing, as much as possible, the compressed data size. At first look, both online and offline compression seem to share the same final goal. However, online compression focuses, in addition to reducing the compressed data size, on reducing the overhead of encoding and decoding. In this work, we focus on online decompression, where only *decoding* is assumed to be in the critical path. This is true for many real-time applications such as the ASML Wafer Heat Feed Forward (WHFF) model. *Encoding* is assumed to be outside of the critical path and therefore not relevant to the total execution time. Furthermore we can classify compression algorithms into: (i) *fixed rate*, and (ii) *variable rate*. In fixed rate compression, each tile in the original dataset is encoded using a *fixed* number of bits. In contrast, with variable rate compression, the number of bits per compressed tile *varies* across the data. In this work we focus on variable rate, as this generally achieves much higher compression ratios at the same error level.

To quantify the impact of applying compression to speed up applications with a data transfer bottleneck, we will present the execution path with and without compression and derive equations. This will show why it is important to have fast decoding and a high compression ratio.

Figure 1.1 shows the execution path of an application without using compression. Let T_{transfer} be the time needed to transfer the data over the bandwidth-limited channel, and T_{app} be the time needed by the application to process the data once it arrives. Assume that data transfer and computation are pipelined. Then, it follows that the latency and period of the system *without* compression are given by:

$$\text{Latency} = T_{\text{transfer}} + T_{\text{app}} \tag{1.1}$$

$$\text{Period} = \max\left(T_{\text{transfer}}, T_{\text{app}}\right) \tag{1.2}$$

Now, suppose that we compress the data before sending it with an average compression ratio denoted by R. This case is illustrated in Figure 1.2. It follows that the latency

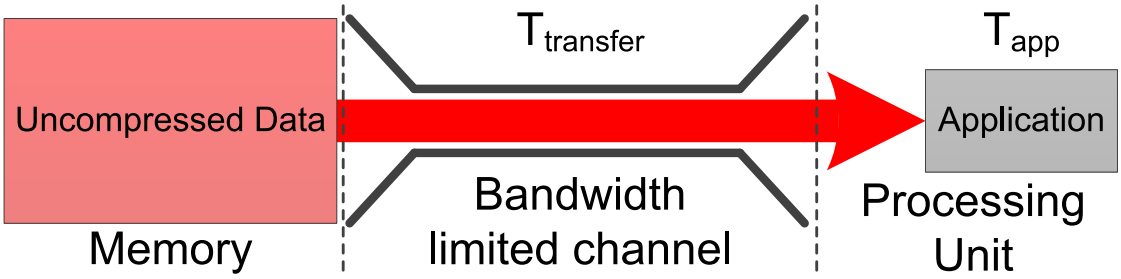


Figure 1.1: Execution path without compression

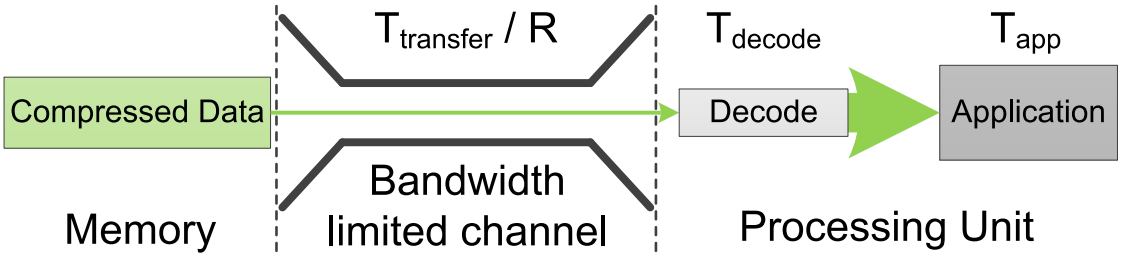


Figure 1.2: Execution path with compression

and period of the system *with* compression are given by:

$$\text{Latency} = \frac{T_{\text{transfer}}}{R} + T_{\text{decode}} + T_{\text{app}} \tag{1.3}$$

$$\text{Period} = \max\left(\frac{T_{\text{transfer}}}{R}, T_{\text{decode}} + T_{\text{app}}\right) \tag{1.4}$$

It is straightforward to see from Equations 1.3 and 1.4 that the data transfer time is reduced by a factor R. However, as shown in Figure 1.2, a decode step is added which has its own latency (denoted by T_{decode}). Therefore, it is important to keep the decoding time as short as possible. In order to speed up decoding on modern multi-core HW platforms, for example GPUs, it should be done in *parallel*.

1.3 An industrial case study: feed forward control in lithography machines

The work described in this thesis was carried out at ASML. ASML is the world's largest manufacturer of photolithography machines. Photolithography is the process of printing an image on silicon to produce an integrated circuit (IC). An important driver for the development of this industry is Moore's law [8]. This is a prediction made by Gordon Moore in 1965 that the number of components on an IC would double every year until 1975, and from then on double every two years. This prediction has proven to be accurate for decades [9].

In lithography, physical and chemical effects due to imperfections of the system and wafer introduce small errors in the printed image. As the dimensions of the features

to be printed are shrinking, these errors are becoming increasingly problematic. To compensate for this, modern lithography machines model these processes and apply corrections. This is called *computational lithography*.

At this moment Extreme Ultraviolet (EUV) Lithography is seen as the key enabler to shrink transistor dimensions over the next decade. EUV lithography has been in development for over two decades and over the last five years the first machines are deployed in production. As the technology is maturing, a major challenge is the rapid increase in complexity of computational lithography [10]. An example of a physical model with high computational requirements is the Wafer Heat Feed Forward (WHFF) model [11] introduced in EUV machines. Bamakhrama et al. recently published a paper [12] on using HW acceleration to run this model within its real-time requirements. In this paper, they state that the WHFF model can benefit greatly from GPU acceleration. When using GPUs to run this model, the bottleneck is the PCIe interface found on all modern Commercial Off The Shelf (COTS) GPUs. In order to alleviate this bottleneck to accelerate the application they propose to use data compression. The algorithm they use to meet the compression ratio requirements while staying within their error bounds is ZFP in variable rate mode. However, there is no support for ZFP variable rate modes on GPU yet. For this application it is essential to develop a strategy that can do PVLD (parallel variable length decompression) on a GPU with high decompression throughput.

1.4 Problem statement

Contrary to parallel fixed rate decompression, parallel variable rate decompression is challenging. We will illustrate this by presenting how fixed rate decompression is parallelized in tiled algorithms and explain why this simple approach can not be applied to variable rate decompression.

We start with an uncompressed dataset and a tiled compression algorithm. When we look at the set of tiled compression algorithms, nearly all of them apply the same three general steps:

1. Divide the data into tiles
2. Apply a transformation (often based on decorrelation)
3. Encode the blocks with either fixed or variable rate

The results of these steps is what we refer to as the *(compressed) bitstream*. It is a series of encoded blocks. Now, we want to access a specific block in this bitstream. Before we can do this, we have to decode it. In order to decode a specific block the decoder needs to know the bit position corresponding to the start of this block in the bitstream. We call this position the *block offset* and define it as follows:

Definition 1.1 *(Block Offset) The* ***block offset*** *of block n, denoted by $D(n)$, is the number of bits between the start of a compressed bitstream and the starting bit of the nth compressed block where $n > 0$. $D(n)$ is given by:*

$$D(n) = \sum_{i=0}^{n-1} L_i \tag{1.5}$$

where L_i is the length of compressed block i in bits.

Recall from Section 1.2 that in fixed rate the blocks in the compressed bitstream have the same compression ratio. This means that a fixed-rate compressed block length is the original block length B divided by the compression ratio R. Substituting this in Equation 1.5 gives:

$$D(n) = \sum_{i=0}^{n-1} L_i = n \cdot \frac{B}{R} \tag{1.6}$$

This means that the offset of any block in a fixed-rate compressed bitstream can be computed easily if the compression ratio R is known. As these parameters are constant and known at encode- and decode-time, fixed rate decompression is easy to parallelize. On the other hand, with variable rate compression, the compressed block length is not constant. This means that the expression from Equation 1.5 does not simplify like in Equation 1.6. It remains a summation where the offset of the current block depends on the accumulated length of all previous blocks. Hence, the challenge in PVLD is to efficiently access or compute the block offset for every block in the compressed bitstream.

1.5 Research contributions

In this thesis we address the following research questions:

1. **Can we develop a solution for PVLD which**
   - is generic and can be applied to multiple compression algorithms?
   - can be used efficiently on multiple HW platforms?
   - allows the end user to make a trade-off between compression ratio and decoding throughput?

2. **Can we make an implementation that is able to run the WHFF model within its requirements?**

To answer the questions we develop a solution for PVLD. Our solution is based on generating extra information during encoding that can be used to parallelize the decoding. We call this extra information *side-channel information* and we propose a set of three strategies to encode it. We also often refer to it as *overhead*, since it is extra data which does not contain information about the original data but is required to allow parallel decoding. Figure 1.3 illustrates the concept by depicting the execution path with compression and the addition of the side channel information.

We aim to make our solution applicable to different algorithms by basing our strategies on generic properties that are shared between algorithms. Our side channel information encoding strategies have different properties in terms of overhead size and decoding method. This will allow us to choose the optimal strategy based on the HW platform used. It also provides the end user with a trade-off between compression ratio and decoding throughput based on their application. We will implement our strategies and choose a configuration that allows us to run the WHFF model on a GPU.

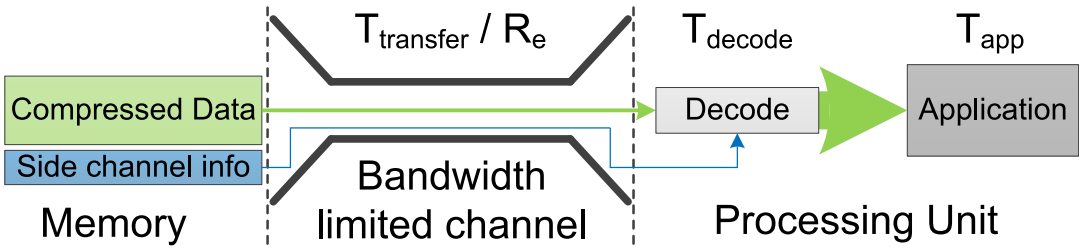


Figure 1.3: Execution path with compression including side channel information

1.6 Thesis structure

The thesis is structured as follows: In Chapter 2 we provide background information on parallel computing, on compression algorithms and on ZFP. In Chapter 3 we show related work in the field of parallel decompression on CPUs and GPUs and compare our solution to other existing and self proposed solutions. In Chapter 4 we introduce our general solution for parallelizing tiled variable rate decompression. We also propose a ZFP specific optimization to increase the decoding throughput on GPU. In Chapter 5 we evaluate the performance of our strategies applied to ZFP on a multicore CPU and a GPU. We also show that our proposed solution is able to meet and surpass the WHFF requirements. Finally, in Chapter 6 we conclude the results of our strategies. We also give recommendations on how to use them effectively and what is important in parallel decompression on different HW platforms.

2 Background

In this chapter we give background information on parallel computing. Then, we compare different floating-point compression algorithms. Finally we show why ZFP was chosen for the WHFF model and we give an explanation of how ZFP works.

2.1 Parallel computing

Historically, computing performance has been growing with Moore's Law for years, meaning that a new CPU would offer a significant performance increase for older software, as the chip would be much faster. However this scaling has ended more than a decade ago and forced chip makers and software application designers to go towards parallel computing [14]. This trend can be seen in both CPUs and GPUs.
A widely used method to classify computer architectures is Flynn's taxonomy [15]. Flynn proposed a model in which computer architectures are classified based on the number of concurrent instructions issued and data-streams used. In this model there are 4 different architecture types: SISD, SIMD, MISD and MIMD. The I and D stand for Instruction and Data, the S and M for Single and Multiple. For example an SISD architecture is one where one instruction is issued to work on one single data stream at a time. Alternatively, an SIMD architecture issues one instruction which works on multiple datastreams concurrently. Modern high end CPUs, which are multi-core and superscalar are defined as MIMD architectures. GPUs are typically SIMD devices.

2.1.1 CPU

CPUs are computer architectures designed to efficiently handle all applications. Because of this the focus in the design process is often on single core performance, since not every application can make good use of multiple cores. Recently the number of cores is increasing significantly in CPUs designed for specific applications, mainly CPUs aimed at the server market. An example of this are Intel Xeon CPUs, which go up to 56 cores each executing 2 threads simultaneously, allowing 112 parallel threads in total [16]. This also holds for AMD Epyc CPUs, where the next generation is announced to have 64 cores with 2 simultaneous threads per core, so 128 threads per CPU in total [17]. These devices are also suited for HPC applications, as these applications are typically designed to exploit parallelism.
In order to make efficient use of the available cores, OpenMP [18] has been developed. OpenMP is an API that allows the user to define parallel regions in their program. It is compatible with C, C++ and Fortran code and supported by nearly every instruction set architecture and operating system.
An example of a possible parallel region is a matrix vector multiplication where each

output element of the resulting vector can be computed independently of the other elements. A possible implementation is to have each thread compute an output element by multiplying a row of the matrix with the input vector and summing all the multiplication results.
A good use of OpenMP is to implement your parallel region in a loop with fixed loop bounds and defining this loop as a parallel region with the pragma `omp parallel for`. Using this, the loop iterations are divided over a number of threads which is either user-specified or the maximum number of threads available on the CPU. This allows the user to control the load balancing. Furthermore, OpenMP has three scheduling policies: static, dynamic and guided. The most simple policy is static, where the loop iterations are divided at compile time into equally sized groups [19]. If the number of groups is not specified by the user, it will be set to the number of threads that can be executed in parallel on the available HW. This is a simple and effective way to divide the work over the available cores, given that the loop iterations are approximately equal in terms of execution time. With dynamic scheduling the loop iterations are divided over threads. These threads are distributed to the cores at runtime based on availability. This can be beneficial when the loop iterations have a varying execution time, as it allows cores that execute the shorter threads to run multiple of them during execution of a long thread on a different core. Guided is similar to dynamic, with the difference that the number of loop iterations per thread is variable. The goal is to allow a fine-grained distribution with low overhead, as the majority of the iterations are divided equally over the number of threads and the remaining iterations are scheduled in a finer grained manner comparable to the dynamic policy.

2.1.2 GPU

Since GPUs have historically been developed for image rendering, the focus has always been on high throughput and floating-point computation power. Since image rendering is a massively parallel application, the number of cores on GPUs is currently orders of magnitude larger than on a CPU. Where high end parallel CPUs have tens of cores and threads, GPUs have up to thousands. An example is the NVidia Tesla V100 which has 5.120 CUDA Cores [20]. Due to this high number of cores, the computational power of a high-end GPU is much higher than any CPU, which makes them ideal candidates for HPC applications [21].
Because the computational performance of GPUs is so high the memory bandwidth is often a bottleneck. This is especially problematic for HPC applications, as these often require large amounts of data. Therefore, GPU manufacturers are using HBM (High Bandwidth Memory) in modern GPUs designed for HPC applications. An example of this is the NVidia Tesla GPU series, where the two most recent high end devices (P100 and V100) are both equipped with HBM2. This trend is expected to continue as both NVidia and AMD are announcing even higher memory bandwidth for their next generation of HPC GPUs.
In order to efficiently use a GPU in software applications, we have to look at the architecture of the device. In this thesis we focus on NVidia GPUs, which execute threads in groups called *warps*. A warp is a group of threads which are executed in parallel on

one streaming multiprocessor (SM). The warp size is the number of threads in a warp and therefore also the number of cores in an SM. Each cycle, all threads in a warp have to execute the same instruction. When each thread wants to issue the same instructions this is a good model which leads to efficient execution with massive parallelism. However when threads want to execute different instructions, for example when control decisions differ between the threads, this leads to *warp divergence*. When warp divergence occurs, the different instructions are executed sequentially. This means that while one thread is executing its instructions, the other threads are waiting. This continues until all threads reach the same instruction, which is is called synchronization. Warp divergence is a loss of parallelism and can cause significant slowdown in GPUs. Therefore it is important to keep threads synchronized on warp level.
Another architectural feature of GPUs is the availability of shared memory. NVidia GPUs have a pool of low latency memory per SM which can be accessed by every thread in a warp executing on this SM. This means that data sharing between threads in a warp is very fast. However, when threads in different warps want to exchange data, it has to go through main memory. Limiting or eliminating data dependencies between warps is essential in developing high performance CUDA applications, as the latency of shared memory is much lower than the latency of main memory, even when using HBM.

2.2 Compression algorithms

Compression is an active field of research and development is still happening for existing and new algorithms. For many algorithms the focus is online (de)compression, as encoding and decoding time are becoming critical factors in an increasing amount of applications [22][23]. Both research institutions and commercial organizations are working on this. For example, Facebook and Google both recently decided to open-source a compression algorithm, Zstandard [24] and Snappy [25]. These algorithms are similar in the sense that both of them share the same goal: To (de)compress any kind of data lossless with high (de)compression speed and compression ratio. For Snappy, the focus is more towards the speed rather than the compression ratio while Zstandard allows the user to make a trade-off between them. Furthermore there are algorithms where the compression ratio and/or encoding/decoding throughput is heavily dependent on the input data. These are algorithms that exploit a known property about the input data, such as local correlation or repeating patterns. For example, a transform based algorithm achieves the best results with highly correlated data, and an algorithm based on Huffman Encoding achieves the best compression ratios for data with frequently repeating bit patterns. Three of these algorithms are GFC [26], ZFP [27] and SZ [28].

Table 2.1 shows some properties of GFC, ZFP, SZ, Zstandard and Snappy. The first observation is that all of these algorithms use a variable rate mode. This is because variable rate modes generally achieve much higher compression ratios than fixed rate mode. For lossy algorithms it should be noted that this means a higher compression ratio at the same error bound. The reason for this is simple: in a large dataset, the information density is nearly always varying across the data. With lossless algorithms it has been shown the compression ratio is bound by the Shannon Entropy of the dataset [29], where the optimal policy is to encode symbols with a different number of bits based on how

Algorithm	Fixed rate	Variable rate	Tiled	Lossless or Lossy	Data types	Platforms	License	Languages	Source
GFC	×	✓	✓	Lossless	FP-DP	GPU	BSD Custom	CUDA	[26]
ZFP	✓	✓	✓	Lossy	Int, FP-SP, FP-DP	CPU, GPU	BSD	C, C++	[30]
SZ	×	✓	×	Lossy	FP-SP, FP-DP	CPU	Open Source	C, Fortran, Java	[28]
Zstandard	×	✓	×	Lossless	Up to 64-bit	CPU	BSD, GPLv2	C, others	[24]
Snappy	×	✓	×	Lossless	Up to 64-bit	CPU	BSD	C++	[25]

Table 2.1: Properties of Floating-Point compression algorithms
FP-SP stands for Floating-Point Single Precision and FP-DP stands for Floating-Point Double Precision, as defined in IEEE-754 [3]

frequent they occur. As the number of bits per encoded symbol is variable, this is considered variable rate encoding. Lossy algorithms allow for some loss of information to increase the maximum achievable compression ratio. To understand why variable rate is favored over fixed rate, consider the following. When we compress a sequence with high information density and one with low information density using the same error bound, the resulting compression ratio will be higher for the sequence with low information density. If we have a dataset consisting of 10 tiles with varying levels of information density and we apply fixed rate compression, we have to encode every tile with the same number of bits. This leads to high errors in the high density tiles, unnecessarily low compression ratio for the low density tiles, or both. With variable rate the number of bits used in encoding can vary between tiles, which means we can spend bits on high density tiles and save bits on low density tiles. This leads to higher overall compression ratio at the same error level.

Out of all the algorithms in Table 2.1, ZFP is the only one that has a fixed rate mode. The reason for this is twofold: it simplifies memory management and random access. Additionally, it also simplifies parallelization of decoding.

For many applications, including the WHFF model, random access is not a requirement. Here compression is applied solely to reduce strain on a bandwidth limited channel and the array is decompressed as soon as possible after arrival. As the error bound of an application is user defined, the goal is to achieve maximum compression ratio and decompression throughput at this error bound. In those cases it is optimal to use a variable rate mode with an efficient PVLD strategy. This conclusion is also reached in the paper about the WHFF model [12], where the authors call for a 'GPU-friendly' decompression scheme, for example an efficient GPU implementation of ZFP variable rate modes.

An algorithm that is designed specifically for GPUs is GFC. However, it can not be used for the WHFF model as it is a lossless algorithm and therefore it is unable to achieve the required compression ratio. Furthermore, GFC only allows inputs in double precision. As the WHFF dataset is single precision it would require extra conversion steps which increase the size of the dataset. Finally GFC is not as well documented and maintained as ZFP which makes it more difficult to deploy in a business context.

Snappy and Zstandard are both lossless algorithms which are also unable to achieve the required compression ratio for the WHFF dataset. For this application the algorithms under consideration are SZ and ZFP.

SZ is a prediction based algorithm, which takes a sequence of input values and encodes every value as either a prediction based on the previous values, or 'not predictable'. This algorithm performs well if there is predictability in the data, meaning that subsets of the data are (near) constant or follow a linear trend or quadratic trend. As the WHFF dataset contains correlated data, this algorithm performs well. However, at the same error bound ZFP achieves a higher compression ratio on the WHFF dataset. Furthermore, SZ has no GPU implementation yet, meaning that the implementation effort is considerably higher than with ZFP. Finally, we aim to develop a general solution for parallelization that can be applied to multiple algorithms. With SZ this is difficult, as the parallelization strategy that is currently implemented in their parallel CPU implementation is based on algorithm specific properties rather than a property that many algorithms share, such as tiling. Therefore we choose to use ZFP. The same choice was

made by Bamakhrama et al. [12] for the WHFF model.

2.3 ZFP

ZFP is an algorithm developed for high speed (de)compression with high compression ratio. Its development is supported by large research project for to develop next generation HPC platforms such as the US Department on Energy's Exascale Computing Project [31][32]. It is tiled and transform based, which means the compression is based on local correlation within tiles where a high degree of correlation will lead to a high compression ratio. ZFP compression can be described by the three step process introduced in section 1.4. In ZFP, tiles are referred to as *blocks*, which is the term we will use for the rest of this thesis. A block is a set of 4^d values, where d is the dimensionality of the block from one (1D) to four (4D).

ZFP is a lossy algorithm which has three modes of operation: fixed rate, fixed precision and fixed accuracy. In fixed rate mode every block is encoded using a fixed number of bits, as introduced in section 1.2. Fixed precision and fixed accuracy are both variable rate modes. In fixed *precision* mode the user specifies a *relative* error bound, meaning that the error introduced by compression can not be larger than for example 1% of the original value. In fixed *accuracy* mode the user specifies an *absolute* error bound, meaning that the error introduced by compression can not be larger than for example 1e-3. The proof that ZFP satisfies these user defined error bounds, along with a more extensive description of the algorithm, can be found in [33].

In the next chapter, we will introduce our solution for parallel variable rate decompression.

Related Work and Alternative Solutions

3

Parallel variable rate compression has received a lot of attention in the past few years [34, 26, 35, 36, 13]. Most of the existing work is motivated by the need to alleviate the bandwidth limitation between memory and processing units. In this chapter we will evaluate the existing solutions and show why they are not suitable for our problem.

3.1 Speculative execution

In [13], the authors proposed a new approach to parallel decoding based on *speculative execution*. Speculative execution parallelizes decoding based on prediction of the block boundaries. Their implementation splits decoding into three pipelined stages: Scanner, Decompresser and Merger.

The *Scanner* stage scans the compressed bitstream and predicts where block boundaries are. Based on these predictions, the *Decompresser* stage will decompress blocks starting from the predicted block boundaries. The correctness of the predictions can be checked when previous blocks are done decompressing, since the end of a block is the boundary with the subsequent block. For correct predictions, the decompressed block is correct and the *Merger* stage will add it to the output. If the prediction was incorrect, the incorrectly decompressed block is discarded and the correct block will be decompressed from the now known block start. This is repeated until the whole compressed bitstream has been decompressed and merged successfully.

The authors have applied their strategy to three variable rate compression algorithms: zlib, bzip2 and H.264. Their approach achieves a speedup of 1.2 to 8.53 on a 36 core platform and their results show that the optimal speedup is achieved at 3, 14 and 18 cores respectively for their chosen compression algorithms. In their results it can be seen that the speedup is not linear with the number of cores used. When increasing the number of cores used above the mentioned optimal numbers, the speedup saturates or even decreases. This shows that the maximum speedup that can be achieved with their strategy is limited and does not scale well with an increasing number of cores.

In order to evaluate the performance of this method it is important to look at:

1. The prediction accuracy (percentage of predictions that are correct)
2. The mis-prediction penalty (time lost when the prediction is incorrect)
3. Prediction overhead (time spent on predicting block boundaries)

The limited scalability of speedup is a property of speculative execution due to the following three reasons. Firstly, they mention that a mis-prediction causes the detection recovery overhead to propagate through the following predictions. This recovery is a

penalty that does not reduce, but stays constant or even increases with an increasing number of cores. Since it is not possible to completely eliminate prediction errors, this issue will always limit the speedup for a large number of cores. Furthermore, speculative execution does not account for load balancing, which is a major factor in the speedup difference between algorithms reported in the paper. This will lead to scalability issues, as load imbalance is a major limiting factor for scaling parallel efficiency. Finally, the overhead of their scanner step is dependent on the compression algorithm their strategy is applied to. Detecting a block boundary is dependent on the encoding strategy applied to the blocks. This will limit the maximum achievable speedup in algorithms where it is difficult to detect block boundaries.

We conclude that this approach is not suitable for our purpose, as our primary objective is fast decoding. With this approach the speedup saturates after using a low number of cores, while we need to keep scaling this into thousands of cores for GPU acceleration.

3.2 JPEG compression

JPEG is an image compression algorithm which is mostly comparable to ZFP. It is also lossy, uses tiling and is transform based. Since JPEG is widely used and a lot of research has been done on efficient encoding and decoding it is a good reference point to compare our work to.

Firstly, there is research on parallelizing individual steps of the JPEG algorithm. In [37] the authors parallelize the most complicated step of JPEG, the Embedded Block Coding (EBC). The authors report that a Verilog implementation of their proposed method can achieve up to 6 times speedup and a factor 6 reduction in memory bandwidth compared to other sequential implementation. Although the authors have devised a scalable method to parallelize the EBC, it is not applicable for us for two reasons. Firstly, the authors focus on a specific step of JPEG, which means that their approach can only directly be applied to JPEG and needs reworking for other algorithms. Also, when parallelizing a single step of an algorithm, the maximum achievable speedup is bound by Amdahl's Law. All other steps of JPEG decompression are executed serially in their approach, which significantly limits speedup scalability.

An approach more similar to ours is [38] in which the authors propose a way to parallelize Motion JPEG XR decoding on a GPU. They propose to parallelize JPEG decoding on the block level, similar to our method. However they mention that for JPEG there are dependencies between blocks, which makes it difficult to parallelize on the block level. Their method to limit these dependencies is to increase the size of the blocks, which in turn limits the number of blocks that can be processed in parallel. The authors manage to achieve a decoding speed of 3652 frames per second for 352x288 images, to 46 frames per second for 7680x4320 images on an NVidia GeForce GTX 480 GPU. The use of a GPU shows that their approach of parallelizing decompression on the block level is scalable to a high number of cores.

Patent [39] describes an approach to parallelize JPEG very similar to our offsets strategy. It is based on placing restart markers in the compressed stream, which creates blocks that can be decompressed independently. Then the bit location of these restart

markers with respect to the start of the bitstream, the offsets, are stored and used to parallelize the decoding. The difference with our approach is that this strategy requires insertion of restart marker into the compressed bitstream, which means the parallelization strategy and algorithm are not completely decoupled. This is needed because in JPEG the blocks are coupled, whilst in ZFP they are independent. Furthermore, we provide different methods of encoding the information besides simply storing the offsets. This means that our proposed framework is more flexible, as it allows the user to trade off compression ratio for decoding speed.

3.3 Side-channel information approaches

In Section 1.4 we stated that in order to parallelize decoding on block level we need to know the block boundaries. We have concluded that fixed rate is not a feasible solution for our problem as it often achieves low compression ratios when compared to variable rate. Furthermore, in Section 3.1 we show that predicting block boundaries is also not a feasible solution for large scale parallelism as it does not scale well. Therefore the most suitable approach is to transfer extra information which helps identify the block boundaries. This is an often used approach and can be implemented in many ways.

3.3.1 Block length encoding

In [34], the author implemented a parallel variable length encoding (PVLE) scheme suited for integer data sets. The scheme is based on saving the length of each *codeword* and then performing a parallel *prefix sum* [40] on the array of lengths. The author reports a speedup of 35x to 50x when comparing this method on an NVidia GeForce GTX280 GPU to a serial CPU method.

The lengths strategy is a reasonable approach and is one of the strategies in our proposed framework. However, we will explore different schemes for generating the extra information needed to parallelize the decompression process and show that the lengths based approach used in [34] is not the most suitable one for modern GPUs. Also, this work is not directly applicable to our case as the parallelization strategy has been applied to a Huffman encoder which is not able to achieve the compression ratio that is needed for our application. Furthermore in this work the strategy has only been applied to GPU. It does not mention a parallel CPU implementation or the expected benefits of parallelizing CPU execution. Since our work shows that the optimal strategy is dependent on the HW platform used, it is important to compare execution on both. Finally, the paper only mentions encoding, while we focus on decoding.

3.3.2 GFC

In [26], the authors proposed an algorithm, called GFC, for parallel compression and decompression on GPUs. It achieves decoding throughputs of over 95 Gb/s (12 GB/s) for many datasets on an NVidia GeForce GTX285 GPU. GFC shares many similarities with the ZFP algorithm used in our work. Both algorithms are tiled compression algorithms and support floating-point data. However, GFC is a lossless algorithm which means

that the achievable compression ratio is very limited, between 1.013 and 3.528 for their listed datasets. Furthermore, GFC is specifically designed to map well to GPUs. As GPUs and CPUs are inherently different architectures it is expected that GFC will not perform well on CPUs without adjusting the algorithm. In this work we propose to apply a parallelization framework on top of an algorithm (in our case ZFP) where the parallelization strategy can be based on the used HW platform without changing the underlying compression algorithm.

3.4 Block length quantization

A solution to reduce the challenge of finding the block boundaries is to apply quantization to the block lengths. While fixed rate allows only a single length for a compressed block and variable rate modes allow any length, we propose a quantized length approach where only a number of block lengths is allowed. For example, in a dataset where the original size of a block is 512 bits, a fixed rate mode with rate 4 would only allow compressed blocks with a length of 128 bits and a variable rate mode would allow any length between 1 and 512 bits. A quantized mode on the other hand could allow lengths of for example 128, 256, 384 and 512 bits. These lengths do not have to be distributed uniformly, as long as the longest possible block length is included in order to guarantee that the error bounds are satisfied. An advantage with this approach is that the block boundaries are much easier to predict, since there are a limited number of possibilities much smaller than in the original variable rate modes. Another advantage is the limitation in bits required to store the length. If we allow only 4 lengths instead of 512, the number of bits required to store side-channel information which indicates the block lengths is reduced from 9 to 2.

However, this approach also has a number of disadvantages. First and foremost, the algorithm should be adjusted in order to only allow the specified block lengths and not any arbitrary value between 1 and the maximum length. This means that with this approach we have to change the algorithm itself, not only the parallelization strategy, which goes against our intention of developing a parallelization method that is independent of the algorithm. Furthermore in order to satisfy the error bounds we have to use the same number of bits or more than in the 'optimal' variable rate modes, where we use the minimum number of bits required to meet the error bounds. This means that compared to the original variable rate modes all block lengths will be rounded up to the nearest allowed length. Increasing the number of allowed lengths will decrease the number of bits added by this 'rounding', but increase the number of bits required to store the side-channel information per block. This trade-off is input data dependent and therefore to do this efficiently one would need to analyze the data to determine the optimal number and values of quantization intervals.

This method is not the most suitable for our purposes since this will make the parallelization strategy dependent on the algorithm, as mentioned before. Also some early tests have shown us that the rounding overhead will often be more than the side-channel information size in the case where we do not apply quantization at all, therefore increasing the size overhead introduced to allow parallelization.

	Speculative exeuction	Step-level parallelism	Side-channel information	Block length quantization
Decoding throughput	+/-	+	++	++
Scalability	- -	- -	++	++
Data overhead	++	+	-	- -
Algortihm independence	-	- -	+	-

Table 3.1: Properties of the different parallel decoding solutions
++ indicates positive, - - indicates negative

3.5 Solution comparison

Table 3.1 summarizes the properties of each strategy discussed in this chapter. Since decoding throughput and scalability are the most important properties for our case, it is clear that speculative execution and step-level parallelism are not suitable for our case. Speculative execution has limited decompression throughput at a low number of cores due to the extra scanner and merger steps that are introduced, and the throughput does not scale well into a high number of cores. Although step level parallelism may give good speedup for a low number of cores, its scalability is limited by the steps of the compression algorithm that remain serial. Also, it is very algorithm dependent which means a strategy for one algorithm can not simply be applied to a different one.

The side-channel information and block-length quantization approaches are similar in terms of decoding throughput and scalability. Block length quantization can even be faster considering that the possible lengths are restricted, meaning a decoder can be optimized (e.g. unrolling) to efficiently decode specific lengths. However, placing this restriction on the algorithm itself modifies the compressed bitstream and can break compatibility with the original implementation. The advantage of side-channel information is that it generates extra information based on the properties of the encoded bitstream, rather than adding restrictions to the format of the bitstream. This, combined with the fact that the side-channel information overhead is often much smaller than the introduced overhead by padding compressed blocks to fit a specific length, leads us to conclude that our side-channel information approach is the optimal method for our use-case.

4 Proposed Solution

In this section we introduce our solution, which can be applied to any tiled algorithm. We propose three parallelization strategies and discuss their advantages and disadvantages. Then, we look at practical implementation constraints to fit our strategies, applied to ZFP, on modern hardware. Finally we look at some ZFP specific optimizations which are needed to speed up ZFP decompression for our example application, the WHFF model.

4.1 Side-channel information

As mentioned in Section 1.4, in order to decode blocks of encoded data in parallel, the block offsets have to be known before decoding starts. These block offsets can be stored directly or can be computed with the sum in Equation 1.5. If one has only the compressed bitstream, neither the block offsets nor the block lengths needed for the sum are known at the decoder. Therefore our solution is to generate *extra side-channel information* during encoding and transfer it to the decoder. The side-channel information can be transferred over the same channel used to send the compressed data. This extra information allows us to compute the start of each compressed block in the bitstream in parallel. In the absence of the side-channel information, the decoder falls back to sequential decoding.

Figure 1.3 illustrates the concept of our solution. We use the extra information to massively speed up the decoding. Therefore, as mentioned in Section 1.5, there is a trade-off between the decoder speedup and the introduced side-channel size overhead. The side-channel information can be compressed itself to reduce its size further, for example using Binary Interpolative Coding [41]. However, this introduces a side-channel decoding step before the data decoding can start. For the sake of simplicity, in the rest of this thesis we assume that the side-channel information is sent as is.

4.2 Definitions

We start introducing our solution by defining its *overhead factor* as shown in Definition 4.1.

Definition 4.1 *(Overhead Factor) Let S be the total size of the original uncompressed data in bits, R be the average compression ratio and I be the total number of bits required to encode the side-channel information. The* ***overhead factor****, denoted by f, is given by:*

$$f = \frac{I}{S/R} \tag{4.1}$$

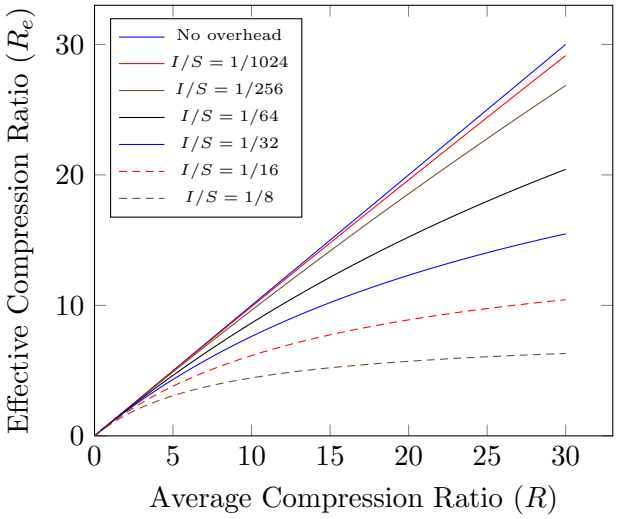


Figure 4.1: The effective compression ratio R_e versus the average compression ratio R for different ratios of I / S

Sending extra information, over the same communication channels as the compressed data reduces the effectiveness of compression. This reduction results in a new *effective compression ratio* defined as follows:

Definition 4.2 *(Effective Compression Ratio) The* ***effective compression ratio****, denoted by R_e, is the original uncompressed data size divided by the total amount of data transferred over the communication channel. R_e is given by*

$$R_e = \frac{S}{S/R + I} = \frac{R}{1 + f} \tag{4.2}$$

It can be seen from Equation 4.2 that the effective compression ratio depends merely on two factors: (i) the average compression ratio R, and (ii) the overhead factor introduced in Definition 4.1. To minimize the impact of the extra information on the transfer time, one would like to minimize the side-channel information size I with respect to the original data size S, as this in turn minimizes f. The impact of I on R_e can be visualized as shown in Figure 4.1.

To develop strategies that minimize I and still allow for a large decoding speedup, we will introduce a number of definitions. Given a block with length L, we define the number of bits required to encode L as I_L and it is given by:

$$I_L = \lceil \log_2(L) \rceil \tag{4.3}$$

Similarly, given a block with block offset $D(n)$, then the number of bits required to encode $D(n)$ is denoted by $I_{D(n)}$ for $n > 0$ and it is given by:

$$I_{D(n)} = \left\lceil \log_2 \left(\sum_{i=0}^{n-1} L_i \right) \right\rceil \tag{4.4}$$

Finally, we define I_D to be the maximum number of bits required to encode an offset for any block in a compressed bitstream of N blocks with an average data compression ratio of R. As an offset is a cumulative sum of lengths, the largest offset is the one of the last compressed block in the bitstream. Then I_D is given by:

$$I_D = \max\left(I_{D(n)}\right) = \left\lceil \log_2\left(\sum_{i=0}^{N-1} L_i\right)\right\rceil \leq \left\lceil \log_2\left(\frac{S}{R}\right)\right\rceil \tag{4.5}$$

Now, we are ready to detail **what** and **how much** extra information we send to compute the block offsets before we start decoding. In this work, we propose three strategies for sending extra information which we explain in the following sections.

4.3 Strategy 1: offset encoding

The first and simplest strategy is to transfer the block offsets directly. This means that the information can be used without an extra processing step during decoding. However, from Equation 4.5, it follows that the number of bits required to encode offsets grows with the original data size S. This means that encoding an offset using a fixed number of bits gives an upper bound for the maximum compressed file size. Hence, the transfer size overhead of this strategy can be significant. To reduce this, we introduce the concept of *chunks*.

Definition 4.3 *(Chunk) A **chunk** is a set of C consecutive compressed blocks to be decoded by a single thread.*

To reduce the transfer size overhead, we group the blocks in chunks of size C and encode one offset per chunk rather than one offset per block. As a result, during decompression, every thread decodes a chunk instead of a block. This reduces the extra information size by a factor C. However, the increase in blocks per thread introduces the following two issues:

1. A decoding dependency between blocks in a chunk, which introduces a control loop
2. The number of available threads is reduced by a factor C, which can cause load imbalance on many-core systems

The impact of these issues differs per HW architecture and will be explored later.

Figure 4.2 gives a visual impression of blocks and chunks in an example stream of 12 variable-rate compressed blocks with a chunk size of 2. The example stream is a selection of 12 compressed blocks from the Brain dataset [2], compressed using ZFP with a fixed accuracy of 0.05. The figure shows the offsets that should be encoded for this stream. For the first chunk this starts at 0 and it accumulates for subsequent chunks.

4.4 Strategy 2: length encoding

From Equation 1.5, it can be seen that it is also possible to compute any offset as a cumulative sum of the lengths of the previous blocks. Recall that for an uncompressed

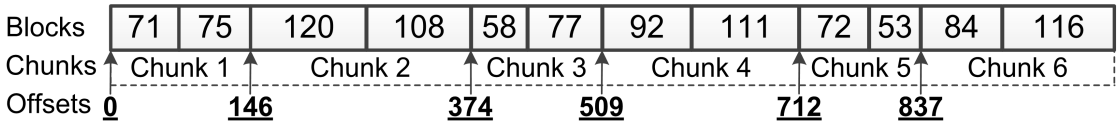


Figure 4.2: Side-channel information for 12 compressed blocks of the Brain dataset [2] using Strategy 1 with a chunk size of 2. The bold underlined values are encoded offsets and the values inside the blocks are the block lengths in bits

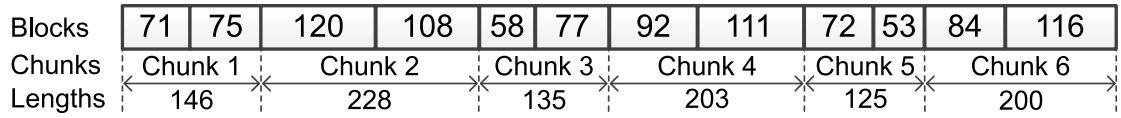


Figure 4.3: Side-channel information for 12 compressed blocks of the Brain dataset [2] using Strategy 2 with a chunk size of 2. The values under the chunks are the encoded lengths and the values in the blocks are block lengths in bits

ZFP block, L is constant and is equal to $4^d \cdot 2^p$. Assume that we do not have expansion, which means the length of any compressed block is smaller than or equal to the length of an uncompressed block, so $R \geq 1$ for each block. Then, from Equation 4.3 it follows that for compressed ZFP blocks the number of bits required is given by:

$$I_{L,\text{ZFP}} = \left\lceil \log_2 \left((4^d \cdot 2^p)/R \right) \right\rceil \leq 2d + p \tag{4.6}$$

Equation 4.6 shows that, contrary to offset encoding, the number of bits required to encode the length of a ZFP block does not scale with array size and has an upper bound based on the data precision and dimensionality. Therefore, instead of transferring block offsets, we propose to transfer block lengths and introduce a *preprocessing* step at the decoder. The preprocessing step computes the required block offsets through a cumulative sum of all the block lengths. Such a sum introduces a dependency between blocks, where the block offset of block n is dependent on the block lengths of the previous $n-1$ blocks. This cumulative sum is also known as Exclusive Scan or Prefix Sum [40]. In a naive implementation, the solution is N sequential additions, where N is the total number of blocks in the compressed stream. This means that the naive implementation can be solved in $\mathcal{O}(N)$ operations with $\mathcal{O}(N)$ sequential steps. However, there are work-efficient parallel algorithms to perform Prefix Sum in $\mathcal{O}(N)$ operations in $\mathcal{O}\left(\log_2(N)\right)$ parallel steps [42].

Similar to Strategy 1, we can create chunks of blocks and encode the length of a chunk rather than a block. This reduces the number of elements in the Prefix Sum by a factor C. However the number of bits needed to encode a chunk length is larger than the number of bits required to encode a block length, as the maximum length of a chunk of C consecutive blocks in bits is C times larger than the length of a single block. This effect can be quantified by substituting L with $L \cdot C$ in Equation 4.3 to see that it increases the required number of overhead bits per chunk by $\log_2(C)$.

Figure 4.3 shows the chunk lengths that would be encoded with a chunk size of 2 on the example bitstream shown in Figure 4.2. Here, it can be seen that contrary to chunk

offsets, the chunk lengths do not accumulate, hence the number of bits needed to store them does not scale with the compressed data size.

If we compare Strategy 1 to Strategy 2, we see that Strategy 1 has an advantage in the sense that all of the required information is transferred as is. This means that in Strategy 1, there is no preprocessing step needed to reconstruct the chunk offsets as is the case with Strategy 2. However, the overhead size for Strategy 1 is larger than the overhead size of Strategy 2. The main issue with the preprocessing step in Strategy 2 is that even in a work efficient parallel implementation, its duration scales with the number of blocks. In order to address the issues associated with both strategies, we propose a third *hybrid* strategy. The third strategy combines the advantages of strategies 1 and 2 in order to minimize both size and computation overhead. Before we introduce the third strategy, we define the concept of *partitions* as follows:

Definition 4.4 *(Partition) A **partition** is a set of P consecutive chunks, hence a partition contains $P \cdot C$ consecutive blocks.*

4.5 Strategy 3: hybrid encoding

The idea of this strategy is to divide the decoding work over partitions, where each partition is independent of other partitions. This means that the complexity of any preprocessing step is bounded by the partition size. The partitions are encoded efficiently to ensure minimum size, while also limiting the time consumed by preprocessing. For a partition of cardinality P, the side-channel information is encoded as follows:

- The first element will be encoded as a single chunk offset
- The remaining $P - 1$ elements will be encoded as chunk lengths

Such an encoding scheme provides the following advantages:

- An offset as first element decouples the partition from other partitions
- The partition size P decouples the complexity of the preprocessing step from the array size

As a result, the preprocessing step execution time does not scale with the array size for Strategy 3. Figure 4.4 shows Strategy 3 applied to the example from Figure 4.2. The chunk size C is set to 2 and the partition size P is set to 3. With such parameters, the encoded information starts with one chunk offset, followed by two chunk lengths per partition. The chunk lengths within a partition are used to compute the chunk offsets using Prefix Sum. The overhead size for large partitions is comparable to that of Strategy 2 as for every P blocks we store $P - 1$ chunk lengths and only a single chunk offset. Thus, the overhead per partition is defined by:

$$I_H = I_D + (P - 1) \cdot I_L \tag{4.7}$$

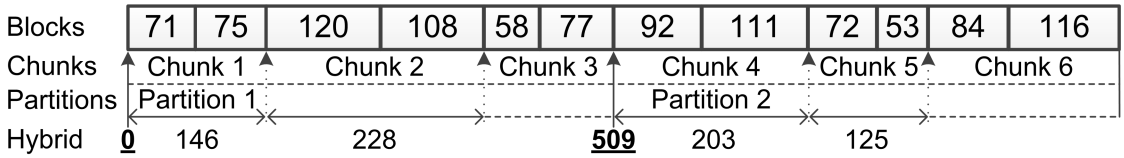


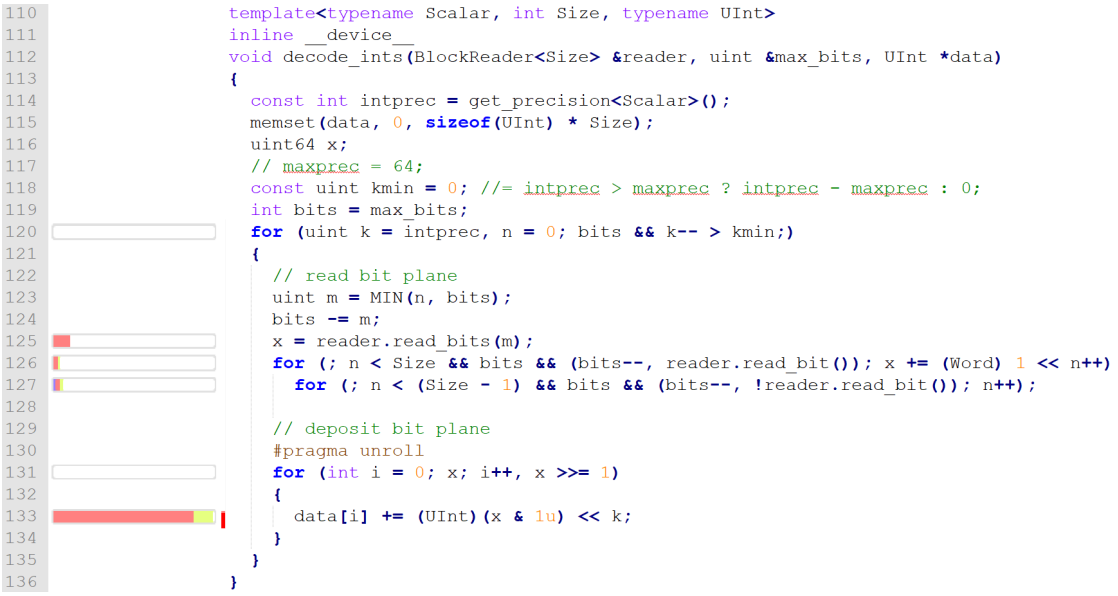
Figure 4.4: Side-channel information for 12 compressed blocks of the Brain dataset [2] using Strategy 3 with a chunk size of 2 and a partition size of 3. Values in the blocks are block lengths in bits. The bold underlined values are chunk offsets, the others are chunk lengths

Strategy	f
Strategy 1	$R \cdot \left(\lceil \log_2\left(\frac{S}{R}\right)\rceil\right)/(C \cdot L)$
Strategy 2	$R \cdot (\lceil \log_2(C \cdot L)\rceil)/(C \cdot L)$
Strategy 3	$R \cdot \left(\lceil \log_2\left(\frac{S}{R}\right)\rceil + (P-1) \cdot (\lceil \log_2(C \cdot L)\rceil)\right)/(P \cdot C \cdot L)$

Table 4.1: Overhead factor f for the side-channel strategies

4.6 Overhead analysis and implementation considerations

To compare the overhead size of the different strategies we use Table 4.1. The values for f are computed by substituting the expression of I in Equation 4.1 for every strategy. This means that these values are the maximum achievable f, as they assume a size-optimal implementation, meaning that the information is stored in the absolute minimum number of bits required. However, there are several practical and hardware considerations that prevent us from using the minimum number of bits. First, most modern computer architectures work with data types that are multiples of bytes. An implementation with for example 7 or 33 bits per value would increase complexity and may limit performance. Therefore, we choose to implement schemes where lengths and offsets are encoded using standard data types, so $I = 2^{k+3}$ with $k \in \mathbb{N}$. Second, in the case of ZFP, the number of bits required to encode chunk lengths in Strategy 2 and 3 is dependent on the dimensionality of the array. For example, a block length for a 1D single precision array requires 7 bits to encode, while a block length for a 3D double precision array requires 12 bits. In order to make one general solution that is applicable to all modes, one has to "over-size" the type used for storing the chunk lengths. Hence, such a type will not be optimal in terms of overhead size for all dimensions and precision modes. Third, the number of bits used to encode chunk offsets places an upper bound on the compressed file size. In this work, we have a design requirement to support compressed files with sizes up to 1 TiB. Such file size means that the chunk offset size must be encoded with at least 43 bits. In order to fit a 43 bit value within standard data types, we choose to encode chunk offsets as 64 bit integers. In turn, this allows us to address a compressed file of up to 2 EiB.



```
110  template<typename Scalar, int Size, typename UInt>
111  inline __device__
112  void decode_ints(BlockReader<Size> &reader, uint &max_bits, UInt *data)
113  {
114    const int intprec = get_precision<Scalar>();
115    memset(data, 0, sizeof(UInt) * Size);
116    uint64 x;
117    // maxprec = 64;
118    const uint kmin = 0; //= intprec > maxprec ? intprec - maxprec : 0;
119    int bits = max_bits;
120    for (uint k = intprec, n = 0; bits && k-- > kmin;)
121    {
122      // read bit plane
123      uint m = MIN(n, bits);
124      bits -= m;
125      x = reader.read_bits(m);
126      for (; n < Size && bits && (bits--, reader.read_bit()); x += (Word) 1 << n++)
127        for (; n < (Size - 1) && bits && (bits--, !reader.read_bit()); n++);
128
129      // deposit bit plane
130      #pragma unroll
131      for (int i = 0; x; i++, x >>= 1)
132      {
133        data[i] += (UInt)(x & 1u) << k;
134      }
135    }
136  }
```

Figure 4.5: Stalls caused per line of code

4.7 ZFP specific GPU optimizations

Besides the general approach of presenting a framework for parallel variable rate decompression for tiled algorithms, we also look specifically at ZFP, release version 0.5.4. The goal is to improve the current CUDA fixed rate implementation, and carry any changes over to the variable rate version that will be implemented. Such improvements are useful for ZFP in general and are required to meet the WHFF model time budget.

To accelerate the ZFP GPU implementation, we analyze the execution time breakdown in order to identify the biggest contributors. For this we use NVidia Visual Profiler [43]. This tool allows us to profile the execution time of individual functions and analyze up to assembly level where stalls, warp divergence and other performance decreasing effects are occurring. From there we can decide which (sub)functions need to be re-evaluated.

Figure 4.5 shows the code of a core function of ZFP decoding, decode_ints. It can be seen that a large number of stalls are occurring in line 133. Figure 4.6, which shows the distribution of the causes of pipeline stalls in the application. As decode_ints is the only function where stalls due to memory dependencies occur, it is clear that this line is the most significant cause of stalls and therefore significantly slows down the application. In order to reduce or remove the stalls, we have to analyze the implementation of this function.

Line 133 is an implementation of the transposed storage of a decoded bit plane. What this means is that the bit plane, which is currently stored in a 64-bit unsigned integer, needs to be stored as a single bit (e.g. the second bit) of 64 subsequent values. This is illustrated in Figure 4.7. The current implementation stores the last bit and right-shifts the bit plane, until all leftover bits to be stored are zeros. This is possible because the memory where the values have to be stored is initialized as all zeros, meaning

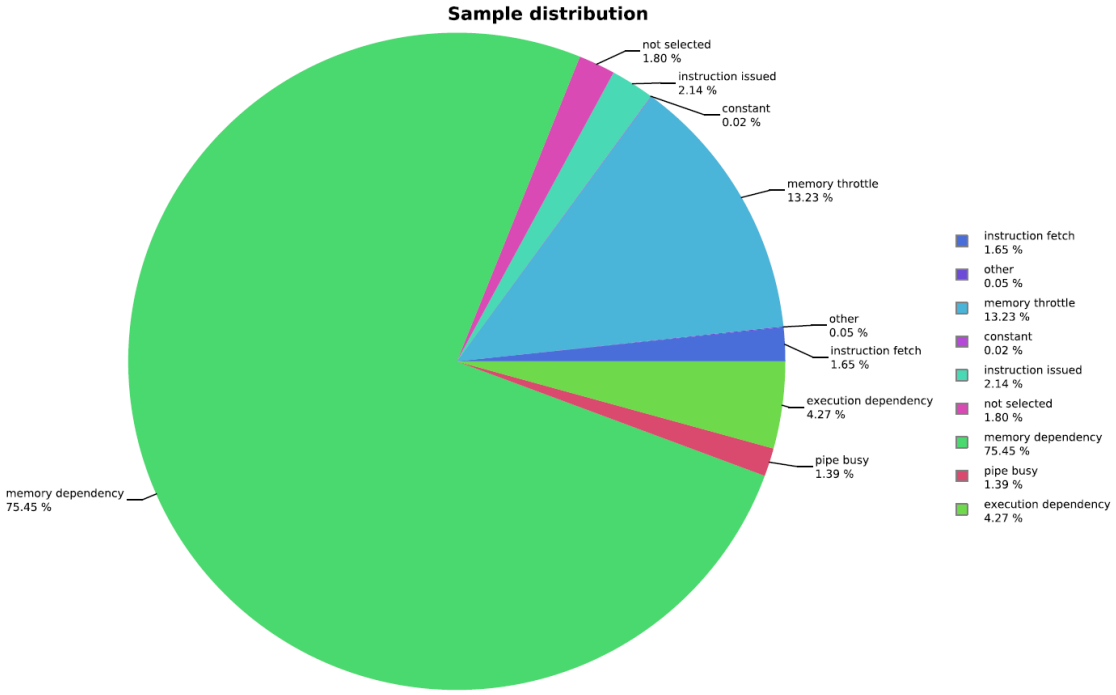


Figure 4.6: Distribution of causes of stalls

that storing a zero bit is essentially a redundant operation. The implementation is work-efficient as it minimizes the amount of store operations and can terminate the loop earlier than the maximum 64 iterations. However on GPU, this early termination causes warp divergence, since not every bit-plane has the same number of non-zero bits to store. This warp divergence leads to stalls, and false memory dependencies for the load and store operations in the divergent code. We propose to set this loop bound to the maximum theoretical value, the 64 iterations. This increases the total amount of work to be done. However, it eliminates warp divergence and therefore the false memory dependencies. It also allows us to unroll the loop with a factor of 64, which eliminates control dependencies between iterations. We expect this optimization to completely eliminate the memory dependency stalls and warp divergence on line 133, which will cause significant decoding speedup. The speedup will be most significant at low compression ratios, as this line is called once per decoded bit plane and the number of encoded bit planes is inversely proportional to the compression ratio.

Another cause of stalls are lines 125, 126 and 127. Line 125 reads a variable number of bits based on the state of the decoder. Lines 126 and 127 together form the implementation of a decision tree, where the decoder reconstructs the original bit planes based on the encoded bitstream. All of these 3 lines implement processes based on variable input data, which is the encoded bitstream. Therefore it is very difficult to eliminate dependencies or warp divergence. We decide that trying to optimize these lines is currently not worth the effort for our application, as the potential speedup is expected to be minimal.

In the next chapter we will evaluate the results of implementing the fixed loop bound

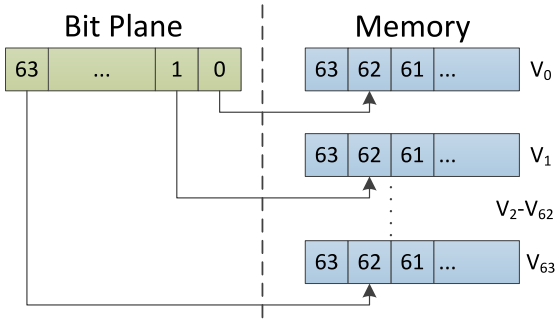


Figure 4.7: Transposed storage of a bit plane of 64 bits

optimization, as well as our proposed strategies for parallel variable rate decoding.

5 Evaluation and Results

In this Chapter, we present the results of evaluating our three proposed strategies in Chapter 4. First, we describe the experimental setup. Then, we measure the performance with a range of different datasets. We analyze both the CPU and GPU results. This is done to get an insight in the properties of our strategies, and to find the optimal strategy for each HW platform. Finally, we evaluate the compression ratio and GPU throughput of ZFP using the ASML dataset.

5.1 Experimental setup

We evaluate the proposed strategies on both a CPU and a GPU. The parallel variable rate decoding is implemented in the following frameworks:

- **OpenMP**: targeting CPUs
- **CUDA**: targeting NVIDIA GPUs

We use a dedicated test server to evaluate ZFP specific optimizations, our strategies and the results of applying ZFP to the WHFF model. The configuration of our test server is outlined in Table 5.1.

We implement our proposed strategies on top of ZFP release version 0.5.4. First, we apply the fixed loop bound improvement as was mentioned in Section 4.7. Then, we compare the performance of the optimized CUDA fixed rate implementation on NVidia Tesla P100 and V100. Following this, we extend the CUDA fixed rate implementation to also support our strategies for variable rate decoding.

In the CUDA fixed rate implementation, the block offsets are computed by multiplying the block number by the compressed block size, as denoted in Equation 1.6. To support variable rate, we first introduce a parameter that contains information on the

Table 5.1: The setup used for evaluating the strategies

Item	Value
CPU	Intel Xeon Bronze 3106 (dual socket, 6 cores/socket)
GPU	NVIDIA Tesla V100 32 GiB (PCIe Gen3 16 lanes)
RAM	256 GiB DDR4 2666 EEC
Storage	1.92 TiB SSD SATA disk
OS	CentOS 7 (64-bit)
Compiler	GNU GCC 4.8.5
CUDA	CUDA version 10.0.130

execution policy. Then, if we are in a variable rate mode, we allocate and transfer the side channel information to the GPU along with the compressed data. We also introduce the concept of chunks and a chunk size parameter, which is ignored in fixed rate mode, but based on the side channel information in variable rate modes. The decoding in fixed rate is unchanged, it still creates one thread for each block and computes the block offset based on the block number. However, for variable rate, we create one thread per chunk which decodes C blocks sequentially. The chunk offset is read or computed from the side channel information, based on the strategy applied as explained in Chapter 4. ZFP release v0.5.4 does not support OpenMP decoding at all. However, it does support OpenMP encoding as well as serial decoding. In order to implement parallel OpenMP decoding, we use the same method as described for the CUDA version. We implement both fixed and variable rate modes and use static OpenMP scheduling. The full code of ZFP and parallel variable rate decompression, which includes the CUDA and OpenMP decoding, can be found on Github, branch zfp_parallel [1] and in Appendix A.

We choose to use open source datasets used in research on compression algorithms in order to get reproducible results. For initial ZFP benchmarking we use a single precision version of the Brain [2] dataset. This is because it is 2D single precision, similar to the ASML dataset, and it provides comparable decoding throughput when compressed with the same compression factor as the ASML dataset. For evaluation of the strategies we use multiple datasets with different properties (1/2/3D single/double), which are described in Table 5.2. Finally for the WHFF model we generate results using the ASML dataset and then provide results using the Brain dataset to allow verification. For all datasets on a CPU, we use the evaluation methodology outlined in Figure 5.1. The decoding throughput is defined as the uncompressed array size divided by the decoding time, so S/T_{decode}. In ZFP, the decoding throughput is highly dependent on array dimensionality, data type and compression ratio. We use multiple datasets which have different dimensionalities and data types. For every dataset we use 5 ZFP precision values and we compute the mean of the compression ratios and execution times. The reason to average over these parameters is that the goal is to analyze the impact of our proposed parallelization strategies, rather than the maximum ZFP performance using optimal settings. On a GPU we use the same methodology with chunk sizes 1, 2, 4, 8 and 16.

5.2 ZFP specific GPU optimization results

Figure 5.2 shows the decoding throughput before and after implementing the fixed loop bound and unrolling optimization. It can be seen that the throughput has increased with a factor of at least 1.4, up to 4.4 for low compression ratio. Figures 5.3 and 5.4 show the stalls per line of code and per cause of the optimized version. In comparison to Figures 4.5 and 4.6 it can be seen that the number of stalls on line 133 have decreased massively. Also, there are no more memory dependency stalls due to this line. The impact of the optimization can be seen clearly when comparing the percentage of total stalls caused by memory dependencies, which is 75.45% in the original implementation and only 11.64% after applying the optimization. This optimization has been accepted as a contribution to the official ZFP repository. It can be found in pull request #35 [44]

Table 5.2: The used datasets. More extensive descriptions can be found in the referenced sources

Name	Description
NYX	Cosmological hydrodynamics simulation
ISABEL	Hurricane simulation
CESM-ATM	Climate simulation
BRAIN	Brain impact simulation
BROWN	Synthetic Brown data
PLASMA	Plasma temperature simulation

Name	Dimensions	Precision	Size (MiB)	R	Source
NYX	3D: 512x512x512 (6 fields)	Single	3072.0	4.94	[45]
ISABEL	3D: 100x500x500 (13 fields)	Single	1239.8	3.22	[45]
CESM-ATM	2D: 3600x1800 (79 fields)	Single	1952.9	5.60	[45]
BRAIN	2D: 17730 x 1000	Double	135.3	7.45	[2]
BROWN	1D: 8388609	Double	64.0	5.75	[45]
PLASMA	1D: 4386200	Single	16.7	1.88	[2]

Require: A dataset to be evaluated
1: $\mathbf{R} = \emptyset$ {List of mean compression ratios}
2: $\mathbf{T} = \emptyset$ {List of mean execution times}
3: $\mathbf{I} = \emptyset$ {List of extra information sizes}
4: **for** $i = 0$ **to** 12 **do**
5: $C = 4^i$
6: $\Gamma = \emptyset$
7: $\Lambda = \emptyset$
8: **for** $j = 8$ **to** 24 (incremented by 4) **do**
9: Invoke ZFP on the dataset using precision given by j
10: Add the resulting compression ratio to Γ
11: Store the execution time in Λ
12: **end for**
13: Compute the harmonic mean of Γ and add it to $\mathbf{R}$
14: Compute the standard mean of Λ and add it to $\mathbf{T}$
15: Compute the extra information size I and add it to $\mathbf{I}$
16: **end for**
17: **return** Mean compression ratio, overhead, and execution time of the dataset under every chunk size C

Figure 5.1: Evaluation methodology for the CPU implementation

and in Appendix B.

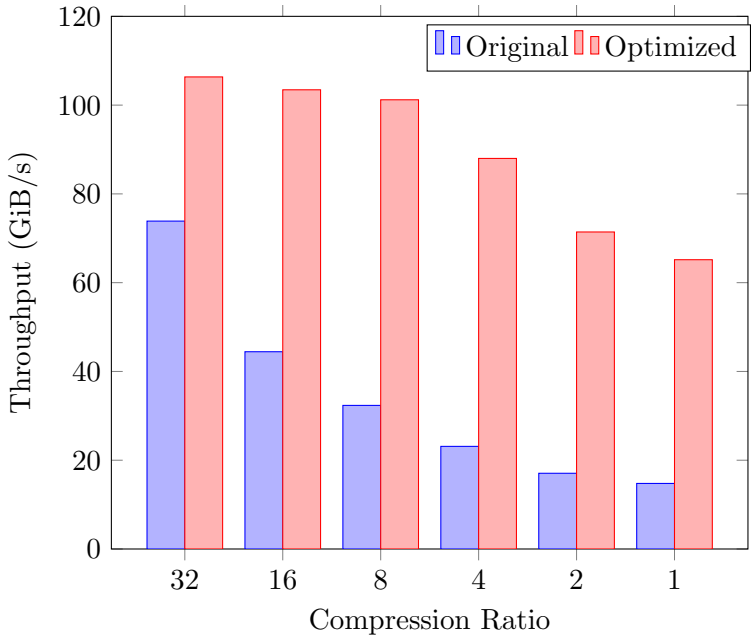


Figure 5.2: Decoding throughput of the original and the optimized CUDA fixed rate decoder on NVidia Tesla V100 using the Brain dataset [2]

```
110 template<typename Scalar, int Size, typename UInt>
111 inline __device__
112 void decode_ints(BlockReader<Size> &reader, uint &max_bits, UInt *data)
113 {
114   const int intprec = get_precision<Scalar>();
115   memset(data, 0, sizeof(UInt) * Size);
116   uint64 x;
117   // maxprec = 64;
118   const uint kmin = 0; //= intprec > maxprec ? intprec - maxprec : 0;
119   int bits = max_bits;
120   for (uint k = intprec, n = 0; bits && k-- > kmin;)
121   {
122     // read bit plane
123     uint m = MIN(n, bits);
124     bits -= m;
125     x = reader.read_bits(m);
126     for (; n < Size && bits && (bits--, reader.read_bit()); x += (Word) 1 << n++)
127       for (; n < (Size - 1) && bits && (bits--, !reader.read_bit()); n++);
128
129     // deposit bit plane
130     #pragma unroll Size
131     for (int i = 0; i < Size; i++, x >>= 1)
132     {
133       data[i] += (UInt)(x & 1u) << k;
134     }
135   }
136 }
```

Figure 5.3: Stalls caused per line of code after applying the fixed loop bound optimization

5.3 NVidia Tesla P100 vs V100 comparison

Our test server contains an NVidia Tesla V100 GPU. However, we want to compare ZFP performance on different GPUs, specifically with NVidia Tesla P100, as both seem suitable for running the WHFF model. To compare we use a node on the ASML High

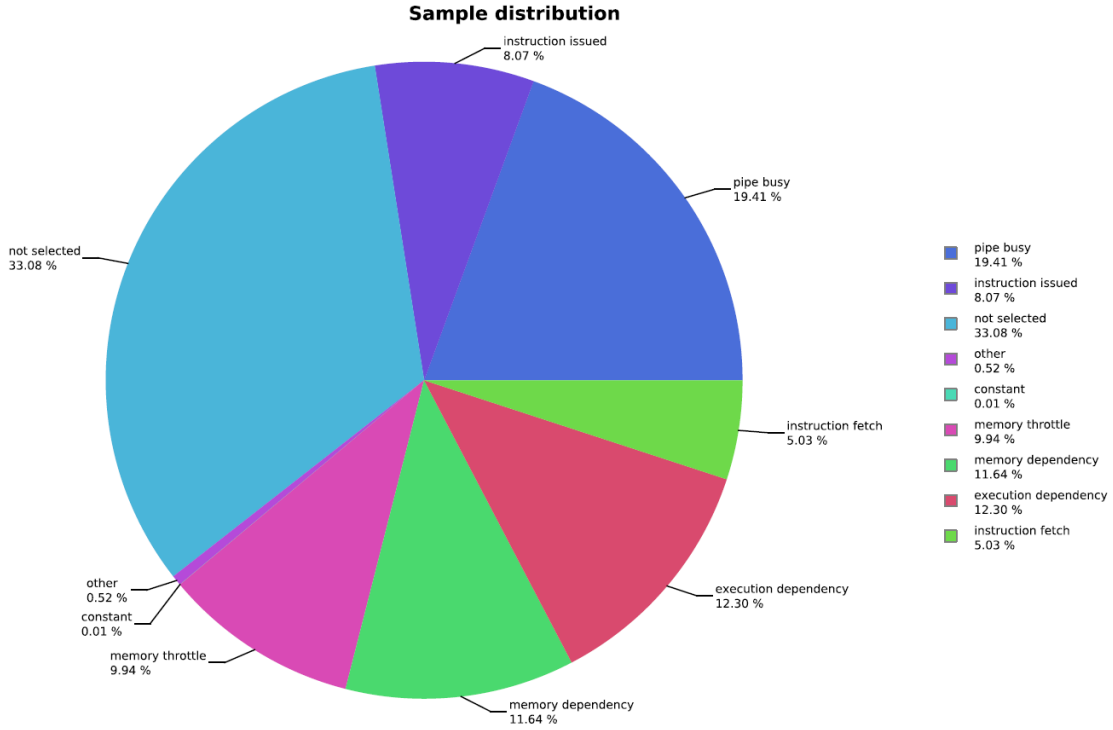


Figure 5.4: Distribution of causes of stalls after applying the fixed loop bound optimization

Table 5.3: The Tesla P100 node on the HPC

Item	Value
CPU	Intel Xeon E5-2660 v4 (single socket, 14 cores/socket)
GPU	NVIDIA Tesla P100 32 GiB (PCIe Gen3 16 lanes)
RAM	4 GiB
Storage	4 GiB
OS	RedHat Enterprise 7.2 (64-bit)
Compiler	GNU GCC 4.9.3
CUDA	CUDA version 8.0.44

Performance Cluster (HPC) which contains an NVidia Tesla P100. The configuration of this node is listed in Table 5.3. We use the optimized fixed rate implementation from Section 5.2 and the Brain dataset [2].

Figure 5.5 shows ZFP decoding throughput on the P100 and the V100. The speedup of the V100 compared to the P100 ranges from a factor 2 to a factor 3. In explaining this large speedup, we first evaluate the differences between the configuration of the HPC P100 node compared to the V100 server. As we are looking at the execution time of a single GPU kernel, we expect that any configuration differences other than the GPU itself have minimal impact on the execution time. The speedup can be attributed to two

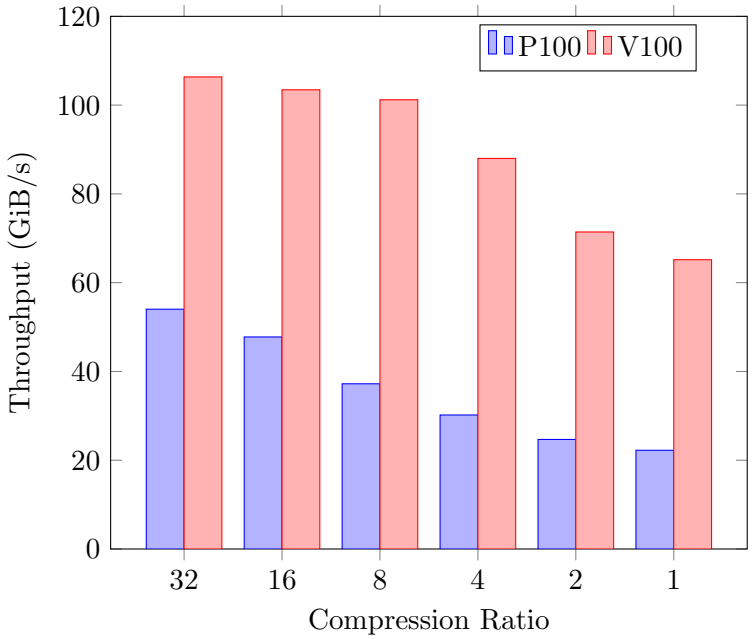


Figure 5.5: Decoding throughput on an NVidia Tesla V100 and P100 with the optimized ZFP fixed rate implementation using the Brain dataset [2]

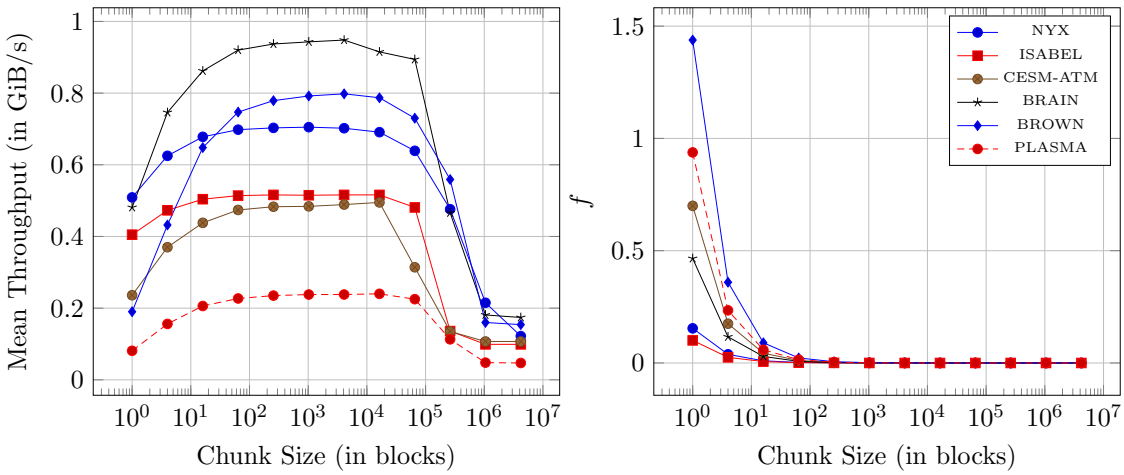
GPU	CUDA Cores	Single Precision TeraFLOPS	Memory Bandwidth (GB/s)	Datasheet
V100	5120	14	900	[20]
P100	3584	9.3	732	[46]

Table 5.4: Specifications of the NVidia Tesla V100 and P100

other factors: general device performance and specific architectural optimizations.

Table 5.4 shows the specifications of the NVidia Tesla P100 and V100. Compared to the P100, the theoretical peak single-precision FLOPS of the V100 is approximately 50.5% higher and the theoretical memory bandwidth is also 23% higher. The speedup can partially be attributed to these differences.

Furthermore, there are some architecture specific differences between the V100 and the P100. The Volta architecture of the V100 is the first NVidia GPU to support independent thread scheduling [47]. This means that in the case of warp divergence, threads can still execute and reconverge on sub-warp granularity, leading to more efficient execution of divergent branches. In Section 4.7 we explain that the core decoding function is an input data based decision tree which causes warp divergence. This function is executed more efficiently on the V100. This can be seen in Visual Profiler, where it is indicated that the number of memory dependency stalls is much higher in the P100 than the V100. This is caused specifically by memory operations before sync operations in the disassembly, as in the P100 these cause the whole warp to stall until the memory operation is completed, rather than scheduling this during execution of another thread



(a) Decoding throughput as a function of C (b) Overhead factor f as a function of C

Figure 5.6: Strategy 1 using OpenMP on Xeon Bronze 3106

as is done in the V100.

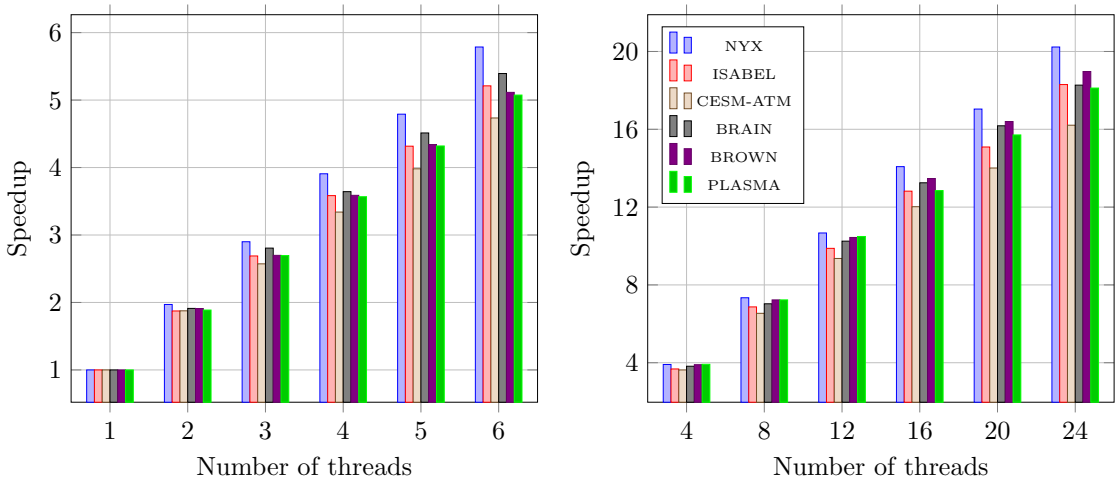
5.4 Strategy 1 results

To evaluate the strategies, we start with Strategy 1. First we show the impact of chunk size C on the throughput and storage overhead. Then, we show the speedup obtained using this strategy. We use the V100 server to evaluate all of our proposed strategies.

5.4.1 CPU results

Figure 5.6a shows the throughput of the OpenMP implementation as a function of the chunk size. We observe that the throughput increases as we increase the chunk size until a certain threshold. This threshold is different for the datasets, but is between 10^4 and 10^6 for most of them. Beyond the aforementioned threshold, the throughput for Strategy 1 starts to degrade. This is caused by insufficient load balancing. Since the chunks are distributed over the threads, the maximum number of threads is inversely proportional to the chunk size. Increasing the chunk size leads to more coarse-grain parallelism, as we parallelize per chunk of C blocks rather than per block. If the number of chunks is smaller than the number of cores, the available parallelism is not fully used. In the worst case, if the chunk size is equal to or larger than the number of blocks, all the blocks are decoded by a single thread. This can be seen in the PLASMA dataset, where increasing the chunk size from 10^6 to 10^7 does not impact the throughput anymore. For the considered datasets, one can conclude that a chunk size in the range $[10^3, 10^4]$ would correspond to near-optimal throughput.

Figure 5.6b shows the storage overhead introduced by Strategy 1 as a function of the chunk size. The y-axis shows the extra information size divided over the compressed data size. Hence, a value of 1 in the y-axis indicates that the overhead is as large as



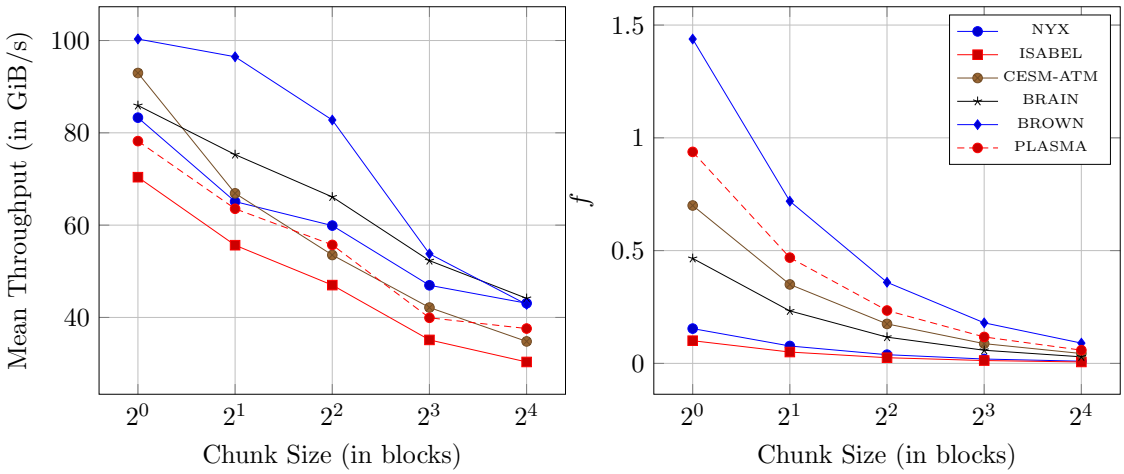
(a) Xeon Bronze 3106 with $C = 4096$ (b) Xeon Gold 6126 with $C = 1024$

Figure 5.7: OpenMP decoding speedup under Strategy 1

the compressed data itself. We observe that Strategy 1 suffers from very large storage overhead for small chunk sizes. However, as the chunk size increases, the overhead drops dramatically and becomes less than 0.0004 for chunk size $C = 4096$. This corresponds to an overhead size of 400 KiB for a compressed size of 1 GiB. Therefore it can be concluded that the storage overhead is negligible to the extent that one does not need to look into further size reduction strategies for OpenMP.

Figure 5.7a shows the speedup achieved by the OpenMP implementation on a 6-core CPU for chunk size $C = 4096$. We see clearly that the implementation has a near-optimal speedup, as it is linear with the number of threads. If we combine the findings from Figures 5.6a, 5.6b, and 5.7a, then one can conclude that Strategy 1 is a very good choice for CPUs as it provides near-optimal speedup for negligible overhead. One can also conclude that there is an optimal choice for chunk size, as the throughput has a clear maximum at a set chunk size. Increasing the chunk size beyond this point has a severe negative impact on throughput, with a minimal gain in overhead as this is already $<1\%$ for chunk sizes above 200.

To investigate the speedup for larger core counts, we evaluated Strategy 1 on an Intel Xeon Gold 6126, which has 24 cores. Figure 5.7b shows the speedup as a function of the number of threads. We used $C = 1024$ instead of the $C = 4096$ we used on Xeon Bronze, since the number of threads is four times higher. At 24 threads, we achieve, on average, a speedup of 18 which corresponds to approximately 75% efficiency. We see that the efficiency varies between the datasets. This is caused by load balancing. Since we use static OpenMP scheduling, the loop iterations are distributed equally over the threads. However, some chunks require more decoding time than others, which means that the time of a loop iteration is not constant, but variable over the whole dataset. A mitigation could be to change from a static to a dynamic scheduling policy, which allows a more fine-grained work distribution. However using a different policy affects the scheduling overhead. The impact of this change is data dependent and can lead to



(a) Decoding throughput as a function of C (b) Overhead factor f as a function of C

Figure 5.8: Strategy 1 using CUDA on Nvidia Tesla V100

a speedup in decoding for one dataset, while it leads to a slow down in decoding for another dataset. To allow a fair comparison between the results of strategies, we only use static scheduling.

5.4.2 GPU results

Figure 5.8a shows the throughput of the CUDA decoder under Strategy 1. If we compare Figure 5.8a to Figure 5.6a we observe that CUDA throughput decreases if we increase the chunk size. This is in contrast to OpenMP, where increasing chunk size would increase throughput up to a threshold. The reasons for this decrease are the load balancing and control loop issues introduced in Chapter 4.3. Since a GPU has many more cores than a CPU, the load balancing issues appear at a much lower chunk size than on the CPU. Furthermore the introduction of a control loop has a larger impact on a GPU than on a CPU. This is due to the fact that after each iteration all threads in a warp need to synchronize. Finally the overhead associated with launching many threads is smaller on a GPU.

Figure 5.8b shows the overhead factor with the CUDA implementation of Strategy 1. Due to the large decrease in throughput, it is not feasible to increase the chunk size to the same extent as in the OpenMP implementation. Therefore, when comparing to Figure 5.6b, we see that the CUDA implementation suffers from a much larger overhead factor. The aim of strategies 2 and 3 is to reduce this overhead factor for the CUDA implementation, without significant loss of decoding throughput.

5.5 Strategy 2 results

5.5.1 CPU considerations

In evaluating Strategy 1, we concluded that the size overhead at chunk sizes that are suitable for OpenMP is minimal. Recall from Chapter 4 that the advantage of Strategy 2 is a reduction of the storage overhead, at the cost of throughput due to the preprocessing step. In order to encode the length of a chunk of 4096 ZFP blocks, one would need at least 24 bits. In order to fit this in a standard data type it would have to be encoded in 32 bits, which is only half of the bits used for the offsets in Strategy 1. Therefore, we conclude that introducing a preprocessing step to halve the already minimal overhead is not worth the implementation effort and, as a result, we do not implement Strategy 2 using OpenMP.

5.5.2 GPU results

On a GPU, we implemented this strategy in an attempt to reduce the overhead. We encoded the lengths with 16 bits per value which ensures correctness for chunks of ZFP blocks with a chunk size up to 2^4. However, the extra time needed to perform the Prefix Sum for large arrays turned out to be the dominant factor in the decoder execution time, even when using a work efficient parallel implementation such as [42]. This confirms earlier observations [34] where other authors also used Prefix Sum for parallel decompression on GPUs and reached the same conclusion. A dominant factor in the Prefix Sum execution time for large arrays is inter-warp dependencies. As a result, Strategy 2 results in a throughput which is lower by a factor of more than two for the same chunk size as Strategy 1. Since our primary objective is maximizing the decoder throughput, we decided that Strategy 2 is not promising for a GPU and we move on to Strategy 3 where we eliminate the inter-warp dependencies to speed up the preprocessing step.

5.6 Strategy 3 results

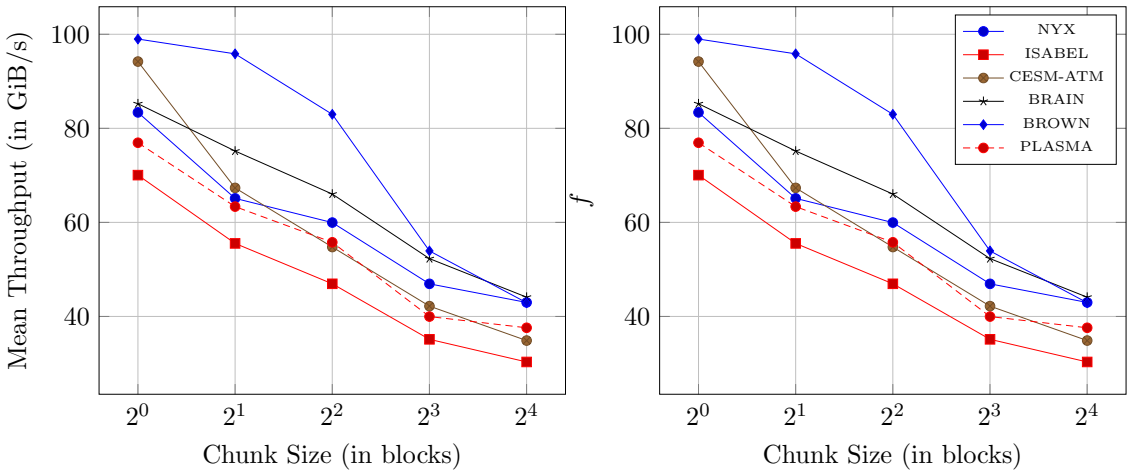
5.6.1 CPU considerations

Similar to Strategy 2 we do not implement this strategy on a CPU as the overhead gain from this strategy is very low while leading to lower throughput.

5.6.2 GPU results

The major disadvantage of Strategy 2 is the time required by the Prefix Sum for large arrays, which is caused by communication between warps. In order to eliminate inter-warp dependencies, we propose a partition size of 32, which is the warp size for modern NVIDIA GPUs [48]. This minimizes both the preprocessing time needed per block and the overhead size.

Figure 5.9a shows the throughput of CUDA under Strategy 3. It can be seen that the results are very similar to Strategy 1. We observe that the impact of the required preprocessing step is minimal, as the difference in throughput is $< 3\%$ for every measurement.



(a) Decoding throughput as a function of C (b) Overhead factor f as a function of C

Figure 5.9: Strategy 3 using CUDA on NVidia Tesla V100

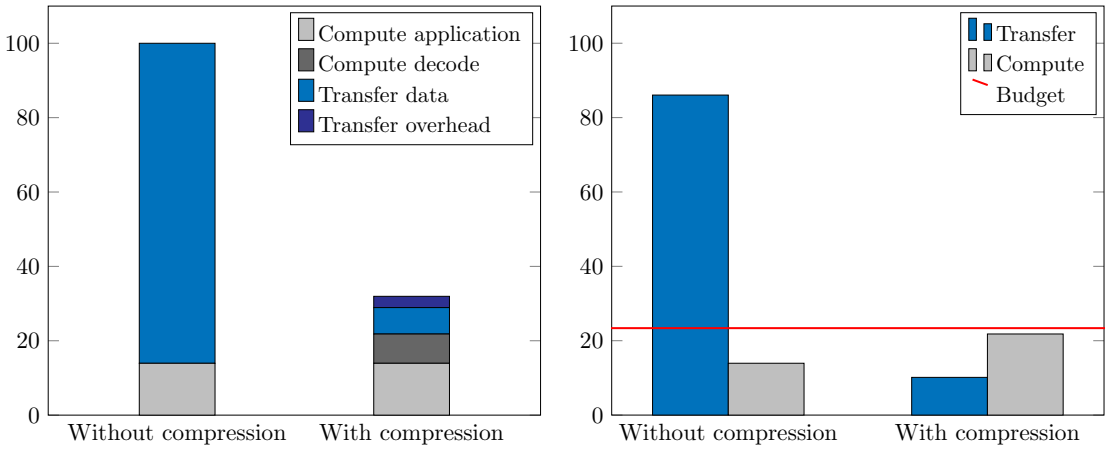
C (Chunk Size)	1	2	4	8	16
R_e (Effective Compression Ratio)	8.49	9.98	10.94	11.49	11.79
Decoding throughput (GiB/s)	131.27	87.41	77.70	60.09	35.62

Table 5.5: WHFF dataset R_e and decoding throughput for different values of C

This confirms our assumption that with a partition size of 32 the required preprocessing step is very efficient. In addition, we observe the same throughput degradation for chunk sizes larger than 1. If we look to the overhead of the CUDA implementation of Strategy 3 as shown in Figure 5.9b, then we see the clear advantage of Strategy 3. Under Strategy 1, the overhead is 64 bits per chunk, while under Strategy 3 the overhead is $64 + 31 \cdot 16$ bits per partition, which is 17.5 bits per chunk. This is an overhead reduction of a factor of 3.66 for less than 3% loss in throughput. This shows clearly that Strategy 3 is well suited for GPUs. However, reducing the size overhead further involves a trade-off as increasing the chunk size impacts the throughput significantly. This trade-off is data-dependent in the case of ZFP and it requires consideration of the application requirements.

5.7 WHFF model results

After analyzing the strategies individually we now look at the results of applying the ZFP variable rate GPU implementation to the WHFF model. The dataset to be compressed is the WHFF dataset, which is 2D and single precision. When compressing the WHFF dataset using ZFP with an application specific accuracy, we can get compression ratios up to $R = 12.1$. As decompression will be done on the GPU we choose to use Strategy 3, the Hybrid approach, with $P = 32$ and multiple chunk sizes. For all these we measure the decoding throughput and effective compression ratio R_e. The results are shown in Table 5.5.



(a) Execution time breakdown

(b) Normalized execution time and available time budget

Figure 5.10: WHFF execution time using ZFP variable rate GPU decompression on Tesla V100, using Strategy 3 with $P = 32$, $C = 1$

From Table 5.5 it is clear that the required decoding throughput of 100 GiB/s can only be achieved with $C = 1$. Figure 5.10 shows the resulting normalized breakdown of execution time of the WHFF model when applying ZFP with these parameters. To analyze if the solution meets the requirements, we look into the time budget for pipelined execution. Note that pipelined here refers to pipelining of the transfer and computation step. For decoding and the application we assume that they are executed sequentially, meaning there is no overlap in their execution times.

Figure 5.10b shows that both the transfer and computation are now executed in the available time budget. The latency as defined in Equations 1.1 and 1.3 has been reduced by a factor **3.13**. The period as defined in Equations 1.2 and 1.4 has been reduced by a factor **3.95**. As the period is the execution time of the slowest stage in a pipelined implementation, it is the inverse of the throughput. This means that our solution increases the effective throughput of the application with a factor of 3.95.

The results shown in Table 5.5 are generated using the WHFF dataset. To verify and recreate our results with the Brain dataset, use the following:

- Dataset: Brain
- Precision: Single
- Dimensions: 2D, 17730 x 1000
- Mode: Fixed accuracy, 0.05
- Side-channel strategy: 3 (Hybrid)
- Partition size: 32
- Chunk size: 1

- GPU: NVidia Tesla V100

With these settings we achieved the following results:

- $R = 11.1$
- $R_e = 8.0$
- Decoding throughput = 115 GiB/s

Conclusions and Recommendations

6

In this thesis we have proposed a strategy for PVLD. Our goal is to answer the following research questions:

1. **Can we develop a solution for PVLD which**
   - is generic and can be applied to multiple compression algorithms
   - can be used efficiently on multiple HW platforms
   - allows the end user to make a trade-off between compression ratio and decoding throughput
2. **Can we make an implementation that is able to run the WHFF model in its requirements?**

We will conclude each question individually.

6.1 Conclusions

Can we develop a solution for PVLD which is generic and can be applied to multiple compression algorithms?
Our solution is based on exploiting the parallelism introduced tiling. We recognize that not all algorithms use tiling and our solution is not completely generic. However many algorithms do and we believe that our solution can be applied to these. We have proven this for one tiled algorithm, which is ZFP. We conclude that our solution can be considered generic at least for tiled algorithms.

Can we develop a solution for PVLD which can be used efficiently on multiple HW platforms?
We have proposed three side channel information encoding strategies, each with different properties. We have implemented them in OpenMP for CPUs and in CUDA for GPUs. With these implementations we show that the optimal choice of strategy and chunk size is dependent on the HW platform used. We conclude that our solution can be used efficiently on multiple HW platforms.

Can we develop a solution for PVLD which allows the end user to make a trade-off between compression ratio and decoding throughput?
We have developed multiple strategies which allows the user to choose their preferred strategy. Furthermore we have introduced the chunk size as a user defined parameter. This allows the user to reduce the side channel information size, often at the cost of decoding throughput. We conclude that this does allow the user to trade-off between these metrics in order to adjust the performance to their application requirements.

Can we make an implementation that is able to run the WHFF model within its requirements?
We have implemented our strategies in the compression algorithm ZFP. This implementation is publicly available on GitHub [1]. With this implementation we have achieved a decoding throughput of 131.27 GiB/s on an NVidia Tesla V100 GPU, which is enough to run the WHFF model within its requirements. With this result we conclude that we have successfully achieved all of our research goals in this thesis.

6.2 Recommendations

In this work, our approach to minimize the size of the side channel information has been to increase the chunk size. However on a GPU this significantly reduces the decoding throughput. On a CPU increasing the chunk size can cause problems in applications that require random access. We suggest future research to investigate alternative options to reduce the size of the side channel information, for example by applying compression on this information.

On a CPU we have used OpenMP static scheduling. We have observed that the speedup with respect to the number of parallel threads is dependent on the dataset. We have noted that using a different scheduling policy can be beneficial to some applications and we suggest future research to further investigate the impact of using different chunk sizes and OpenMP scheduling policies. We believe that this can lead to a framework that determines the optimal chunk size and scheduling policy based on dataset characteristics.

Finally, in this work we have assumed that decoding is part of the critical path while encoding is done offline. Future research can focus on fast and efficient encoding for applications where this is required.

Bibliography

[1] L. Noordsij. (2019) Zfp parallel decompression. Last accessed on: May 7, 2019. [Online]. Available: https://github.com/LennartNoordsij/zfp/tree/zfp_parallel

[2] M. Burtscher and P. Ratanaworabhan, "FPC: A High-Speed Compressor for Double-Precision Floating-Point Data," *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 18–31, Jan 2009. [Online]. Available: https://doi.org/10.1109/TC.2008.131

[3] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[4] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *High Performance Computing for Computational Science – VECPAR 2010*, J. M. L. M. Palma, M. Daydé, O. Marques, and J. C. Lopes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–25. [Online]. Available: https://doi.org/10.1007/978-3-642-19328-6_1

[5] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *IEEE Micro*, vol. 31, no. 5, pp. 7–17, Sep. 2011. [Online]. Available: https://doi.org/10.1109/MM.2011.89

[6] S. Mittal and J. S. Vetter, "A Survey Of Architectural Approaches for Data Compression in Cache and Main Memory Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1524–1536, May 2016. [Online]. Available: https://doi.org/10.1109/TPDS.2015.2435788

[7] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[8] G. E. Moore *et al.*, "Cramming more components onto integrated circuits," 1965.

[9] C. A. Mack, "Fifty years of moore's law," *IEEE Transactions on semiconductor manufacturing*, vol. 24, no. 2, pp. 202–207, 2011.

[10] T. A. B. Harry J. Levinson, "Current challenges and opportunities for euv lithography," vol. 10809, 2018. [Online]. Available: https://doi.org/10.1117/12.2502791

[11] D. van den Hurk, S. Weiland, and K. van Berkel, "Modeling and localized feedforward control of thermal deformations induced by a moving heat load," in *2018 SICE International Symposium on Control Systems (SICE ISCS)*, March 2018, pp. 171–178.

[12] M. A. Bamakhrama, A. Arrizabalaga, F. Overman, J. Smeets, K. van der Sommen, R. van der Vossen, and J. Wagensveld, "GPU acceleration of real-time control loops," *CoRR*, vol. abs/1902.08018, 2019. [Online]. Available: http://arxiv.org/abs/1902.08018

[13] H. Jang, C. Kim, and J. W. Lee, "Practical speculative parallelization of variable-length decompression algorithms," in *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '13. New York, NY, USA: ACM, 2013, pp. 55–64. [Online]. Available: https://doi.org/10.1145/2491899.2465557

[14] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[15] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, Sep. 1972.

[16] Intel xeon platinum 9282 processor. Intel Corporation. Last accessed on: May 26, 2019. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/194146/intel-xeon-platinum-9282-processor-77m-cache-2-60-ghz.html

[17] Z. Liu. Rome in detail: A closer look at amd's 64-core 7nm epyc cpus. Last accessed on: May 26, 2019. [Online]. Available: https://www.tomshardware.com/news/amd-64-core-128-thread-7nm-rome-cpu,38032.html

[18] Openmp specifications. OpenMP. Last accessed on: May 26, 2019. [Online]. Available: https://www.openmp.org/specifications/

[19] (2016) Openmp: For & scheduling. Jaka's Corner. Last accessed on: May 26, 2019. [Online]. Available: http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

[20] (2018) Nvidia tesla v100. NVidia Corporation. Last accessed on: May 14, 2019. [Online]. Available: https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf

[21] J. Nickolls and W. J. Dally, "The gpu computing era," *IEEE micro*, vol. 30, no. 2, pp. 56–69, 2010.

[22] R. Lenhardt and J. Alakuijala, "Gipfeli-high speed compression algorithm," in *2012 Data Compression Conference*. IEEE, 2012, pp. 109–118.

[23] A. Yang, H. Mukka, F. Hesaaraki, and M. Burtscher, "Mpc: a massively parallel compression algorithm for scientific data," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 381–389.

[24] (2019) Zstandard compression algorithm. Facebook. Last accessed on: May 6, 2019. [Online]. Available: https://facebook.github.io/zstd/

[25] (2019) Snappy: A fast compressor/decompressor. Google. Last accessed on: May 7, 2019. [Online]. Available: http://google.github.io/snappy/

[26] M. A. O'Neil and M. Burtscher, "Floating-point data compression at 75 gb/s on a gpu," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 7:1–7:7. [Online]. Available: https://doi.org/10.1145/1964179.1964189

[27] P. Lindstrom, "Fixed-Rate Compressed Floating-Point Arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, Dec 2014. [Online]. Available: https://doi.org/10.1109/TVCG.2014.2346458

[28] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 730–739. [Online]. Available: https://doi.org/10.1109/IPDPS.2016.11

[29] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[30] P. Lindstrom. (2019) Zfp. Last accessed on: March 5, 2019. [Online]. Available: https://github.com/LLNL/zfp/

[31] zfp & fpzip: Floating point compression. Lawrence Livermore National Laboratory. Last accessed on: May 23, 2019. [Online]. Available: https://computation.llnl.gov/projects/floating-point-compression

[32] Exascale computing project. United States Department of Energy. Last accessed on: May 23, 2019. [Online]. Available: https://www.exascaleproject.org/

[33] J. Diffenderfer, A. Fox, J. Hittinger, G. Sanders, and P. Lindstrom, "Error analysis of zfp compression for floating-point data," *arXiv preprint arXiv:1805.00546*, 2018.

[34] A. Balevic, "Parallel variable-length encoding on gpgpus," in *Euro-Par 2009 – Parallel Processing Workshops*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 26–35.

[35] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens, "Parallel lossless data compression on the gpu," in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–9. [Online]. Available: https://doi.org/10.1109/InPar.2012.6339599

[36] J. Pool, A. Lastra, and M. Singh, "Lossless compression of variable-precision floating-point buffers on gpus," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '12. New York, NY, USA: ACM, 2012, pp. 47–54. [Online]. Available: https://doi.org/10.1145/2159616.2159624

[37] Hung-Chi Fang, Yu-Wei Chang, Tu-Chih Wang, Chung-Jr Lian, and Liang-Gee Chen, "Parallel embedded block coding architecture for jpeg 2000," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 9, pp. 1086–1097, Sep. 2005.

[38] B. Pieters, J. D. Cock, C. Hollemeersch, J. Wielandt, P. Lambert, and R. Van de Walle, "Ultra high definition video decoding with motion jpeg xr using the gpu," in *2011 18th IEEE International Conference on Image Processing*, Sep. 2011, pp. 377–380.

[39] K. Zhu and J.-M. Kim, "Method and apparatus for generating jpeg files suitable for parallel decoding," Sep. 17 2013, uS Patent 8,538,180.

[40] G. E. Blelloch, "Prefix sums and their applications," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-90-190, November 1990. [Online]. Available: http://www.cs.cmu.edu/~guyb/papers/Ble93.pdf

[41] A. Moffat and L. Stuiver, "Binary interpolative coding for effective index compression," *Information Retrieval*, vol. 3, no. 1, pp. 25–47, Jul 2000. [Online]. Available: https://doi.org/10.1023/A:1013002601898

[42] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with cuda," *GPU gems*, vol. 3, no. 39, pp. 851–876, 2007.

[43] (2019) Nvidia visual profiler. NVidia Corporation. Last accessed on: May 10, 2019. [Online]. Available: https://developer.nvidia.com/nvidia-visual-profiler

[44] L. Noordsij. (2019) Unroll deposit bit plane loop. Last accessed on: May 10, 2019. [Online]. Available: https://github.com/LLNL/zfp/pull/35

[45] F. Cappello, M. Ainsworth, J. Bessac, M. Burtscher, J. Choi, E. Constantinescu, S. Di, H. Guo, P. Lindstrom, and O. Tugluk. (2019) Scientific data reduction benchmarks. Last accessed on: March 4, 2019. [Online]. Available: https://sdrbench.github.io/

[46] (2016) Nvidia tesla p100. NVidia Corporation. Last accessed on: May 14, 2019. [Online]. Available: https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf

[47] (2017) Nvidia tesla v100 gpu architecture. NVidia Corporation. Last accessed on: May 15, 2019. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[48] *Nvidia CUDA C programming guide*, Nvidia Corporation, 2018, last accessed on: March 5, 2019. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

ZFP patch for CUDA & OpenMP

A

The developed patch adds parallel decompression to ZFP. This patch will be contributed to the ZFP repository after aligning on the exact implementation with the maintainers [1]. This has a large impact as ZFP is a widely used open source library and this patch would allow users to decode faster on a CPU when using OpenMP and to use GPU decompression for variable rate modes. The patch consists of 1.279 lines of code added and 441 deleted in a total of 14 files. It contains new functionality and changes in existing code for indentation consistency. Below we highlight a number of code snippets that form the core of our patch.

A.1 Side channel information allocation

We allocate the side channel information in a struct in order to allow configuration of the type of side channel information (hybrid or offsets, chunk size, etc.). We implement this with an alloc, set and free function to stay consistent with the rest of the ZFP library. The implementation can be seen in the code snippet below.

```
+++ src/zfp.c
+/* public functions: side channel info
   ---------------------------------------*/
+
+zfp_side_channel*
+side_channel_alloc()
+{
+  zfp_side_channel* side_channel = (zfp_side_channel*)malloc(
   sizeof(zfp_side_channel));
+  if (side_channel) {
+    side_channel->length_table = NULL;
+    side_channel->type = 0;
+    side_channel->chunk_size = 0;
+    side_channel->side_channel_data = NULL;
+  }
+  return side_channel;
+}
+
+int
+side_channel_set_params(zfp_side_channel* side_channel, uint16 *
   length_table, side_channel_type type, uint chunk_size, void *
   side_channel_data)
+{
+  if (!side_channel)
+    return 0;
```

```
+  side_channel->length_table = length_table;
+  side_channel->type = type;
+  side_channel->chunk_size = chunk_size;
+  side_channel->side_channel_data = side_channel_data;
+  return 1;
+}
+
+void
+free_side_channel(zfp_side_channel* side_channel)
+{
+  free(side_channel);
+}
+
```

A.2 Side channel information encoding

Encoding of the side channel information is a 2 step process. During compression, the length of each compressed block in bits is written to an array. When compression is done, this array of lengths is used to encode the side channel information based on the input parameters. The current version of this encoding support offsets and hybrid with a variable chunk size, as well as lengths with a chunk size of 1. For hybrid the chunk size is limited by the dimensionality and input data precision, from 16 for 3D double precision to 512 for 1D single precision. Below is a code snippet of this side channel information encoding.

```
+++ src/share/omp.c
@@ -22,4 +23,66 @@ chunk_count_omp(const zfp_stream* stream, uint
   blocks, uint threads)
   return MIN(chunks, blocks);
 }

+/* TODO: consider moving this to zfp.c to allow serial side
   channel encoding */
+static int
+encode_side_channel(const zfp_stream* stream, uint blocks)
+{
+  const side_channel_type table_type = stream->side_channel->type
   ;
+  const uint16* length_table = stream->side_channel->length_table
   ;
+  if (table_type != none) {
+    if (table_type == offset) {
+      uint chunk_size = stream->side_channel->chunk_size;
+      uint64* offset_table = (uint64*)stream->side_channel->
   side_channel_data;
+      uint64 sum = 0;
+      int i, chunk, block, chunks;
+      chunks = (blocks + chunk_size - 1) / chunk_size;
+      for (chunk = 0, block = 0; chunk < chunks; chunk++) {
```

```
+        offset_table[chunk] = sum;
+        for (i = 0; i < chunk_size; i++, block++) {
+          sum += length_table[block];
+        }
+      }
+    }
+    else if (table_type == hybrid) {
+      const uint chunk_size = stream->side_channel->chunk_size;
+      const uint chunks = (blocks + chunk_size - 1) / chunk_size;
+      uint16* hybrid_table16 = (uint16*)stream->side_channel->
side_channel_data;
+      uint64* hybrid_table64 = (uint64*)stream->side_channel->
side_channel_data;
+      const uint partitions = (chunks + PARTITION_SIZE - 1) /
PARTITION_SIZE;
+      uint j = 0;
+      uint lim = 0;
+      uint64 sum = 0;
+      uint16 partialsum = 0;
+      uint i = 0;
+      uint chunk = 0;
+      uint a = 0;
+      for (; i < partitions; i++) {
+      /* store the offset for partition i on position i*
PARTITION_BYTES */
+        hybrid_table64[i * 9] = sum;
+        for (chunk = 0; chunk < PARTITION_SIZE; chunk++) {
+          partialsum = 0;
+          for (a = 0; a < chunk_size && j < blocks; a++, j++) {
+            partialsum += length_table[j];
+          }
+          hybrid_table16[i * 36 + 4 + chunk] = partialsum;
+          sum += partialsum;
+        }
+      }
+      /* Finish the last partial partition */
+      for(; chunk < PARTITION_SIZE; chunk++) {
+        hybrid_table16[i * 36 + 4 + chunk] = 0;
+      }
+    }
+    else if (table_type == length) {
+      /* possible encoding of a new lengths table
+      current implementation: write fixed size (2 bytes per block
) lengths table to file */
+      if (stream->side_channel->side_channel_data != length_table
)
+        fprintf(stderr,"side channel information is not the
original lengths table\n");
+    }
+    else {
```

```
+       fprintf(stderr,"unknown side-channel information type to be
    encoded\n");
+     }
+  }
+}
+
 #endif
```

A.3 OpenMP decompression

We have added OpenMP decompression functions for 1D, 2D, 3D and 4D. The functions are consistent with the existing OpenMP encoding in terms of scheduling policy and load distribution, as well as functionality with respect to serial decompression. The current version support offsets with a variable chunk size. A code snippet of decompression for 2D arrays can be found below, the other dimensionalities are implemented similarly.

```
+++ src/template/ompdecompress.c
@@ -0,0 +1,299 @@
+#ifdef _OPENMP
+
+/* decompress 2d strided array in parallel */
+static void
+_t2(decompress_strided_omp, Scalar, 2)(zfp_stream* stream,
    zfp_field* field)
+{
+  if (stream->side_channel->type != offset) {
+    fprintf(stderr, "current version only supports OpenMP
    decompression with an offset table\n");
+    return;
+  }
+  Scalar* data = (Scalar*)field->data;
+  const uint nx = field->nx;
+  const uint ny = field->ny;
+  const int sx = field->sx ? field->sx : 1;
+  const int sy = field->sy ? field->sy : nx;
+  uint threads = thread_count_omp(stream);
+
+  /* calculate the number of blocks and chunks */
+  const uint bx = (nx + 3) / 4;
+  const uint by = (ny + 3) / 4;
+  const uint blocks = bx * by;
+  const uint chunk_size = stream->side_channel->chunk_size;
+  const uint chunks = (blocks + chunk_size - 1) / chunk_size;
+
+  /* allocate per-thread streams */
+  bitstream** bs = decompress_init_par(stream, field, chunks,
    blocks);
+
+  /* read 1 bit from the stream to prevent decompression failed
+  TODO: find a better fix */
```

```
+  stream_read_bit(stream->stream);
+
+  /* decompress chunks of blocks in parallel */
+  int chunk;
+  #pragma omp parallel for num_threads(threads)
+  for (chunk = 0; chunk < (int)chunks; chunk++) {
+    /* determine range of block indices assigned to this thread
   */
+    const uint bmin = chunk * chunk_size;
+    const uint bmax = MIN(bmin + chunk_size, blocks);
+    uint block;
+
+    /* set up thread-local bit stream */
+    zfp_stream s = *stream;
+    zfp_stream_set_bit_stream(&s, bs[chunk]);
+
+    /* decode all blocks in the chunk sequentially */
+    uint x, y;
+    Scalar * block_data;
+
+    for (block = bmin; block < bmax; block++) {
+      x = 4 * (block % bx);
+      y = 4 * (block / bx);
+      block_data = data + y * sy + x * sx;
+      if (nx - x < 4 || ny - y < 4)
+        _t2(zfp_decode_partial_block_strided, Scalar, 2)(&s,
   block_data, MIN(nx - x, 4u), MIN(ny - y, 4u), sx, sy);
+      else
+        _t2(zfp_decode_block_strided, Scalar, 2)(&s, block_data,
   sx, sy);
+    }
+  }
+  decompress_finish_par(bs, chunks);
+}
```

A.4 CUDA decompression

We have added CUDA variable rate decompression for 1D, 2D and 3D. We implemented this by modifying the already existing functions for fixed rate compression and decompression, by implementing checks on everything that is mode specific and adding the variable rate path there. We support hybrid and offsets side channel information, both with variable chunk sizes. Below there are two code snippets: the first shows the parameter setting for the different modes, the second shows the 2D decompression kernel which supports variable rate. The other dimensionalities are implemented similarly.

```
+++ src/cuda_zfp/cuZFP.cu
@@ -374,74 +365,103 @@ cuda_decompress(zfp_stream *stream,
   zfp_field *field)
   Word *d_stream = internal::setup_device_stream(stream, field);
```

```
+  Word * d_side_channel = NULL;
+  zfp_mode mode = zfp_stream_compression_mode(stream);
+
+  /* parameter needed to decode the bitstream differs per
   execution policy */
+  size_t table_size;
+  uint blocks;
+  int param;
+  uint chunk_size = stream->side_channel->chunk_size;
+  side_channel_type table_type = stream->side_channel->type;
+  if (mode == zfp_mode_fixed_rate) {
+    param = (int)stream->maxbits;
+    chunk_size = 1;
+  }
+  else if (mode == zfp_mode_fixed_accuracy || mode ==
   zfp_mode_fixed_precision) {
+    blocks = 1;
+    if (dims[0]) blocks *= ((dims[0] + 3)/4);
+    if (dims[1]) blocks *= ((dims[1] + 3)/4);
+    if (dims[2]) blocks *= ((dims[2] + 3)/4);
+    if (table_type == 1) {
+    /* Check why 'offset' doesnt work and this has to be
   hardcoded as 1 */
+      table_size = ((size_t)blocks + chunk_size - 1 ) /
   chunk_size * sizeof(Word);
+    }
+    else if (table_type == hybrid) {
+      /* Hardcoded to partitions of 32 values with a size of 72
   bytes, could be defines */
+      size_t partitions = (blocks + (32 * chunk_size) - 1) / (32
   * chunk_size);
+      table_size = partitions * 72;
+    }
+    else {
+      std::cerr<<"Non-supported side channel information type for
    GPU. Use hybrid or offset \n";
+      return;
+    }
+    d_side_channel = internal::setup_device_side_channel(stream,
   table_size);
+    param = (mode == zfp_mode_fixed_accuracy ? (int)stream->
   minexp : (int)stream->maxprec);
+  }
+  else {
+    std::cerr<<"Custom mode not supported on GPU\n";
+    return;
+  }

+++ src/cuda_zfp/decode2.cuh
 template<class Scalar, int BlockSize>
 __global__
```

```
 void
 cudaDecode2(Word *blocks,
+            Word *side_channel,
             Scalar *out,
             const uint2 dims,
             const int2 stride,
             const uint2 padded_dims,
-            uint maxbits)
+            const uint total_blocks,
+            const int param,
+            const uint chunk_size,
+            const zfp_mode mode,
+            const side_channel_type table_type)
 {
   typedef unsigned long long int ull;
   typedef long long int ll;
-  const ull blockId = blockIdx.x +
+
+  const uint blockId = blockIdx.x +
                       blockIdx.y * gridDim.x +
                       gridDim.x * gridDim.y * blockIdx.z;
-
-  // each thread gets a block so the block index is
-  // the global thread index
-  const ull block_idx = blockId * blockDim.x + threadIdx.x;
-
-  const int total_blocks = (padded_dims.x * padded_dims.y) / 16;
-
-  if(block_idx >= total_blocks)
-  {
-    return;
+  const uint chunk_idx = blockId * blockDim.x + threadIdx.x;
+  const int warp_idx = blockId * blockDim.x / 32;
+  const int thread_idx = threadIdx.x;
+
+  ll bit_offset;
+  if (mode == zfp_mode_fixed_rate)
+    bit_offset = param * chunk_idx;
+  else if (table_type == offset){
+    bit_offset = side_channel[chunk_idx];
+  }
+  else if (table_type == hybrid) {
+    __shared__ uint64 offsets[32];
+    uint64* data64 = (uint64 *)side_channel;
+    uint16* data16 = (uint16 *)side_channel;
+    data16 += warp_idx * 36 + 3;
+    offsets[thread_idx] = (uint64)data16[thread_idx];
+    offsets[0] = data64[warp_idx * 9];
+    int j;
+
+    for (int i = 0; i < 5; i++) {
```

```
+       j = (1 << i);
+       if (thread_idx + j < 32) {
+         offsets[thread_idx + j] += offsets[thread_idx];
+       }
+       __syncthreads();
+     }
+     bit_offset = offsets[thread_idx];
    }
-
-   BlockReader<BlockSize> reader(blocks, maxbits, block_idx,
    total_blocks);
-
-   Scalar result[BlockSize];
-   memset(result, 0, sizeof(Scalar) * BlockSize);
-
-   zfp_decode(reader, result, maxbits);

    // logical block dims
    uint2 block_dims;
-   block_dims.x = padded_dims.x >> 2;
-   block_dims.y = padded_dims.y >> 2;
-   // logical pos in 3d array
-   uint2 block;
-   block.x = (block_idx % block_dims.x) * 4;
-   block.y = ((block_idx/ block_dims.x) % block_dims.y) * 4;
+   block_dims.x = padded_dims.x >> 2;
+   block_dims.y = padded_dims.y >> 2;
+
+   BlockReader<BlockSize> reader(blocks, bit_offset);
+   uint block_idx = chunk_idx * chunk_size;
+   const uint lim = MIN(block_idx + chunk_size, total_blocks);

-   const ll offset = (ll)block.x * stride.x + (ll)block.y * stride
    .y;
-
-   bool partial = false;
-   if(block.x + 4 > dims.x) partial = true;
-   if(block.y + 4 > dims.y) partial = true;
-   if(partial)
-   {
-     const uint nx = block.x + 4 > dims.x ? dims.x - block.x : 4;
-     const uint ny = block.y + 4 > dims.y ? dims.y - block.y : 4;
-     scatter_partial2(result, out + offset, nx, ny, stride.x,
    stride.y);
-   }
-   else
-   {
-     scatter2(result, out + offset, stride.x, stride.y);
+   for (; block_idx < lim; block_idx++) {
+     Scalar result[BlockSize] = {0};
+     zfp_decode<Scalar,BlockSize>(reader, result, param, mode, 2);
```

```
+
+    // logical pos in 3d array
+    uint2 block;
+    block.x = (block_idx % block_dims.x) * 4;
+    block.y = ((block_idx / block_dims.x) % block_dims.y) * 4;
+
+    const ll offset = (ll)block.x * stride.x + (ll)block.y *
   stride.y;
+
+    if(block.x + 4 > dims.x || block.y + 4 > dims.y) {
+      const uint nx = block.x + 4 > dims.x ? dims.x - block.x :
   4;
+      const uint ny = block.y + 4 > dims.y ? dims.y - block.y :
   4;
+      scatter_partial2(result, out + offset, nx, ny, stride.x,
   stride.y);
+    }
+    else
+      scatter2(result, out + offset, stride.x, stride.y);
   }
 }
```

Deposit bit plane patch

B

This patch has been contributed to the ZFP repository and has been merged into the development branch [44]. It will significantly speed up the CUDA fixed rate decoding for all current users of the ZFP library. Furthermore, it also speeds up decoding in our new CUDA variable rate decoding patch.

```
diff --git a/src/cuda_zfp/decode.cuh b/src/cuda_zfp/decode.cuh
index d3d0877..656cf0b 100644
--- a/src/cuda_zfp/decode.cuh
+++ b/src/cuda_zfp/decode.cuh
@@ -127,12 +127,8 @@ void decode_ints(BlockReader<Size> &reader,
    uint &max_bits, UInt *data)
        for (; n < (Size - 1) && bits && (bits--, !reader.read_bit
           ()); n++);

     // deposit bit plane
+#if (CUDART_VERSION < 8000)
     #pragma unroll
+#else
+    #pragma unroll Size
+#endif
+    for (int i = 0; i < Size; i++, x >>= 1)
-    for (int i = 0; x; i++, x >>= 1)
     {
       data[i] += (UInt)(x & 1u) << k;
     }
```