Universidade Estadual de Campinas
Instituto de Computação

Thiago José Mazarão Maltempi

Checkpointing Optimization in Adjoint-Mode Applications: A Prefetching and Compression-Based Approach

Otimização de *Checkpointing* em Aplicações de Modo Adjunto: Uma Abordagem Baseada em *Prefetching* e Compressão

CAMPINAS
2025

Thiago José Mazarão Maltempi

Checkpointing Optimization in Adjoint-Mode Applications: A Prefetching and Compression-Based Approach

Otimização de *Checkpointing* em Aplicações de Modo Adjunto: Uma Abordagem Baseada em *Prefetching* e Compressão

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Sandro Rigo
Co-supervisor/Coorientador: Prof. Dr. Guido Costa Souza de Araújo

Este exemplar corresponde à versão final da Dissertação defendida por Thiago José Mazarão Maltempi e orientada pelo Prof. Dr. Sandro Rigo.

CAMPINAS
2025

Ficha catalográfica
Universidade Estadual de Campinas (UNICAMP)
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

Maltempi, Thiago José Mazarão, 1993-
M298c Checkpointing optimization in adjoint-mode applications : a prefetching and compression-based approach / Thiago José Mazarão Maltempi. – Campinas, SP : [s.n.], 2025.

Orientador: Sandro Rigo.
Coorientador: Guido Costa Souza de Araújo.
Dissertação (mestrado) – Universidade Estadual de Campinas (UNICAMP), Instituto de Computação.

1. Computação de alto desempenho. 2. Compressão de dados (Computação). 3. Migração reversa no tempo. 4. Otimização de checkpoint. 5. Pré-busca de dados. I. Rigo, Sandro, 1975-. II. Araújo, Guido Costa Souza de, 1962-. III. Universidade Estadual de Campinas (UNICAMP). Instituto de Computação. IV. Título.

Informações complementares

Título em outro idioma: Otimização de checkpointing em aplicações de modo adjunto : uma abordagem baseada em prefetching e compressão
Palavras-chave em inglês:
High-performance computing
Data compression (Computer science)
Reverse time migration
Checkpointing optimization
Prefetching
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Sandro Rigo [Orientador]
Rodolfo Jardim de Azevedo
Arthur Francisco Lorenzon
Data de defesa: 06-08-2025
Programa de Pós-Graduação: Ciência da Computação

Objetivos de Desenvolvimento Sustentável (ODS)
Não se aplica

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0009-0007-9970-8197
- Currículo Lattes do autor: http://lattes.cnpq.br/0861374037150896

- Prof. Dr. Sandro Rigo
- Prof. Dr. Rodolfo Jardim de Azevedo
- Prof Dr. Arthur Francisco Lorenzon

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Dedication

To my lovely wife, Aline Maria, my partner for all moments. Love you.

Lord, give success to the work of our hands
(Psalms 89(90):16-17)

Acknowledgements

First, I would like to thank God and the intercession of Saint Joseph, the Patron Saint of Workers.

I am thankful for my lovely wife and partner, Aline Maria, who is always by my side, and for the love we share. I am grateful for my beloved son Francisco, who taught me a new kind of love. I wish to thank my family, who have encouraged me at various stages of my life, including my Parents and in-laws, my sisters and brother, especially Caio, who listened to my repeated training sessions when I was preparing for EuroPAR'24.

If I wrote this whole MSc dissertation, it is because I was *standing on the shoulders of giants*: Here is my very special thanks to Professor Sandro Rigo, the supervisor of this work, which I am so pleased to be part of; also to Professor Guido Araújo, the co-supervisor, for the valuable lessons in MO644, my first contact with parallel programming in 2015. I wish everyone could have such good Professors, advisors, and friends as I had in this MSc program.

I would like to thank my colleagues from the Laboratory of Computer Systems (LSC) at the Institute of Computing, particularly Marcio Pereira, PhD, Professor Hervé Yviquel, and Gustavo Leite, who assisted me with the learning curve of Awave-3D. Speaking about Awave-3D, I wish to meet with Professor Jessé Costa (UFPA) to thank him for all his help with seismic research and the "millennium bug".

For all professional colleagues who influenced me and taught me a lot during all those years of my career, in particular, my current employer, ASCEND Cardiovascular, which incentivized me and gave me enough flexibility to conclude the MSc. program; also for my co-workers, Derek Such, John Alex, Gustavo Lelis, Christina Coakley, Jeffrey Soble, MD, and Professor James Wolfer, you guys are great!

Last but not least, a big thank you to the examiners of my qualification exam and defense, Professor Arthur Lorenzon, Professor Rodolfo Azevedo, and Professor Alexandro Baldassin, as well as to all the numerous partners who provided valuable feedback and enabled us to improve our research and achieve these results.

The author gratefully acknowledges the financial support for this work provided by the São Paulo Research Foundation (FAPESP), Brazil, under grant [2019/26702-8], and by Petrobras, Brazil, under grants [2018/00347-4] and [2022/00096-7]. The author also thanks the Laboratório Nacional de Computação Científica (LNCC), Brazil, and SENAI/CIMATEC, Brazil, for providing the computational resources used in this work.

Resumo

Sistemas de computação heterogêneos com GPUs são essenciais para lidar com tarefas computacionalmente intensivas nas áreas de física, aprendizado de máquina e imageamento sísmico. Essas aplicações frequentemente utilizam métodos de diferenciação reversa (*adjoint/reverse-mode*), que demandam recomputação. Para reduzir a complexidade algorítmica, aplica-se a técnica de *checkpointing*, que busca equilibrar recomputação e uso de memória. Entretanto, quando os dados de *checkpoint* precisam ser armazenados na memória do host devido às limitações de memória da GPU, a latência na comunicação torna-se um gargalo significativo, podendo consumir até 75% do tempo total de execução.

Esta dissertação propõe uma nova abordagem para mitigar esse problema: combinar um mecanismo de *prefetching* de *checkpoint* com compressão de dados dentro da GPU. Para isso, foi desenvolvida a biblioteca modular GPUZIP, que foi testada com algoritmos de *checkpointing* que acessam seus dados de forma temporal e determinística, como *Revolve*, *zCut* e *Uniform*, permitindo *caching* e *prefetching* de dados de *checkpoint*. O mecanismo de *prefetching* agenda proativamente transferências assíncronas da memória do *host* para a memória da GPU que ocorrem de forma simultânea à computação. Entretanto, o mecanismo de *prefetching* pode não ter tempo suficiente para transferir todos os dados antes de sua utilização. Para que a transferência possa ser mais rápida, a GPUZIP integra compressão com perda (*lossy*) dentro da GPU (*cuZFP* e *NVIDIA Bitcomp*) ao mecanismo de *prefetching*.

Foram avaliados os mecanismos de *prefetching* e de compressão isoladamente, considerando diferentes tamanhos de *cache* e diversas configurações de parâmetros de compressão, com a finalidade de identificar os ajustes que proporcionam maior aceleração sem comprometer a qualidade dos resultados, além de quantificar o impacto isolado e combinado de cada técnica na mesma aplicação.

Os resultados demonstram ganhos expressivos de desempenho em múltiplos *datasets* e estratégias de *checkpointing*. A combinação de *prefetching* e compressão foi a configuração que obteve os melhores ganhos de desempenho de até $5,1\times$ com *Revolve*, $8,9\times$ com *zCut* e $5,8\times$ com *Uniform*. A GPUZIP reduziu significativamente os tempos de bloqueio nas transferências *Host-GPU* e eliminou movimentos redundantes de dados.

Esta pesquisa mostra que a combinação de *prefetching* com compressão é uma estratégia eficaz para superar o gargalo de comunicação em aplicações baseadas em GPU que utilizam diferenciação reversa. A GPUZIP é projetada para ser extensível e compatível com diferentes algoritmos de *checkpointing* e bibliotecas de compressão.

Abstract

Heterogeneous computing systems with GPUs are essential for tackling computationally intensive physics, machine learning, and seismic imaging tasks. These applications often rely on adjoint/reverse-mode methods, requiring recomputation. Checkpointing is employed to balance memory usage and recomputation to reduce algorithmic complexity. However, when checkpoint data must be stored in host memory due to GPU memory limitations, communication latency becomes a significant bottleneck responsible for up to 75% of total execution time.

This dissertation proposes a novel approach to mitigate this problem by combining a checkpoint prefetching mechanism with GPU-based data compression. For this purpose, the modular library GPUZIP was developed. GPUZIP is designed for checkpointing algorithms with temporal and deterministic data access patterns, such as *Revolve*, *zCut*, and *Uniform*, enabling efficient caching and prefetching of checkpoint data. The prefetching mechanism proactively schedules asynchronous *Host-to-Device* transfers to overlap communication with computation. However, prefetching alone may not always complete transfers on time. To accelerate the data movement, GPUZIP integrates lossy compression directly on the GPU by leveraging cuZFP and NVIDIA Bitcomp.

The prefetching and compression mechanisms were evaluated independently and in combination, exploring various cache sizes and compression parameters to identify configurations that maximize speedup while preserving result quality. It also quantifies the isolated and combined impacts of both techniques on the same application.

The results show substantial performance improvements across multiple datasets and checkpointing strategies. The combined use of prefetching and compression achieved the best speedups: up to 5.1× with *Revolve*, 8.9× with *zCut*, and 5.8× with *Uniform*. GPUZIP significantly reduced blocking times during *Host-to-Device* transfers and eliminated redundant data movement.

This research demonstrates that combining prefetching with compression is an effective strategy for overcoming the communication bottleneck in GPU-based reverse-mode applications. GPUZIP is designed to be extensible and compatible with various checkpointing algorithms and compression libraries.

List of Figures

List of Tables

List of Acronyms & Glossary

API	Application Programming Interface.
Awave-3D	Benchmark RTM implementation with checkpointing.
BACKWARD	Backward computation.
Bitcomp	A GPU-based lossy compression algorithm by NVIDIA's NVCOMP.
BPC	Bit-Plane Coding.
Cache Hit	Event where the requested data is found in the cache, allowing for fast access without retrieving it from main memory or recomputing it.
Cache Miss	An event where the requested data is not found in the cache and must be retrieved from a slower memory source or recomputed.
Checkpoint Pool	Checkpointing data storage.
Checkpointing	A technique for saving partial results during forward computation to avoid full recomputation during the backward phase.
CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecture, a parallel computing platform and programming model developed by NVIDIA.
cuSZp	A GPU-based lossy compression algorithm from the SZ family.
cuZFP	A GPU-based lossy compression algorithm from the ZFP family.
D2D	Device-to-Device memory copy.
D2H	Device-to-Host memory copy.
DCT	Discrete Cosine Transform.
DRAM	Dynamic Random-Access Memory.
Error-bounded	Compression that ensures the error introduced by the compression process remains within a predefined limit. Error bounds can be absolute or relative.
FDM	Finite Difference Method.
Fixed-ratio	Compression that guarantees a specific compression ratio. It is suitable for scenarios with strict memory constraints, though it may sacrifice quality.
FLOPS	Floating Point Operation per Second.
FORWARD	Forward computation.
FPGA	Field-Programmable Gate Array.

GPU	Graphics Processing Unit.
H2D	Host-to-Device memory copy.
HBM2	High Bandwidth Memory 2.
HDD	Hard Disk Drive.
HPC	High-Performance Computing.
Lossy Compression	A data compression method that reduces data size by permanently eliminating certain information, particularly in floating-point precision.
LRU	Least Recently Used.
MNMG	Multi-Node Multi-GPU.
MRU	Most Recently Used.
NSight	NVIDIA profiling and trace tool.
NVTX	NVIDIA Tools Extension.
OMPC	OpenMP Cluster, a library that allows OpenMP directives to distribute work across multiple nodes using MPI (Message Passing Interface).
OSS	Open-Source Software.
PAV	Prefetch Action Vector, an array that schedules prefetching operations.
PCIe	Peripheral Component Interconnect Express, a standard interface for communication between the CPU and peripheral devices like GPUs.
PCIe	Peripheral Component Interconnect Express.
Prefetching	The technique of loading data in advance to reduce wait times and improve system efficiency.
PSA	Prefetch Setup Algorithm, method responsible for predicting cache misses and generating the PAV.
PSNR	Peak Signal-to-Noise Ratio.
Raw Cache Miss	Cache miss that occurs in the baseline, before any optimization is applied.
RESTORE	Operation to load or rematerialize the checkpoint data from the cache or pool.
Revolve	A well-known checkpointing algorithm [24].
RQ	Research Question.
RTM	Reverse Time Migration.
SAVE	Operation to take a snapshot/checkpoint of the current state of the computation to the cache or pool.
Scalability	System's ability to maintain efficiency as resources such as nodes or cores are increased.

Shot	A single instance of wave propagation from a seismic source in a survey.
SIMD	Single Instruction, Multiple Data.
SIMT	Single Instruction, Multiple Threads.
Snapshot	The materialization of the computation data (checkpoint).
SNMG	Single-Node Multi-GPU.
SO	Specific Objective.
Speedup	Performance gain observed when comparing the execution time of a baseline method with an optimized method.
SSD	Solid-State Drive.
SSIM	Structural Similarity Index Measure.
Timeliness	Measures the interval of when a prefetch action initiated to the moment the data is requested.
Uniform	Checkpointing algorithm that sets fixed intervals across the execution timeline.
zCut	A checkpointing algorithm based on computational graph [68].

Contents

Chapter 1

Introduction

High-performance computing (HPC) is essential for advancing scientific research and solving complex computational problems in deep learning [63], physics [36], bioinformatics [54], and seismic imaging [5, 58]. These problems require intensive computational resources to handle large datasets and can take days, weeks, or even months to be solved. To meet this demand, heterogeneous systems of accelerators such as Graphics Processing Units (GPUs) are adopted to accelerate calculations. Such systems are capable of delivering petaflops-scale performance [66] while some exaflops-scale clusters are already operational [67].

Even though the hardware can achieve that peak of FLOPS, implementing software that can utilize all that power is what scientists in multidisciplinary fields from all over the world strive to achieve. Multiple techniques are used, but the dream of high performance stumbles at the communication wall.

Communication between nodes in a cluster, between SSDs and DRAM, and from the host's DRAM to the GPU's DRAM creates a significant bottleneck for applications [53, 60]. This issue frequently causes high-speed accelerators to wait for data processing. To illustrate the potential impact of data access locations on application speed, consider the bandwidth disparity between host-GPU communication and GPU memory access, for instance, an NVIDIA v100 GPU using PCIe 3.0 provides a host-device bandwidth of 32 GB/s, whereas access to HBM2 memory (the GPU's DRAM) can reach an impressive 900 GB/s (a 28-fold difference!) [14]. Therefore, the timing and data access method are determining factors for the application's performance.

Applications such as Reverse Time Migration (RTM), used in seismic imaging, are solved through reverse-mode or adjoint computational differentiation. This technique computes gradients or wave equations of discretized timesteps, then computes them backward and correlates them to resolve the problem. Each backward timestep requires the corresponding forward timestep.

At this moment, an attention point for performance comes into action: computing all timesteps forward for each timestep in backward generates an algorithm of $\mathcal{O}(timesteps^2)$, which is highly inefficient. An alternative for that is saving each forward timestep and reusing it, but would generate terabytes of data to be stored, and then the communication between layers of storage (e.g., SSD→Host DRAM→GPU DRAM) becomes a big communication wall [64].

As a solution, *checkpointing* is adopted as a technique to balance between recomputation and storage size [64]. Checkpointing relies on an algorithm to determine the most effective way to manage this balance. During the forward phase, the checkpointing algorithm saves snapshots based on a specific strategy aligned with its design. In the backward phase, these snapshots are restored, allowing for a shortcut in recomputation by resuming from the moment of the snapshot, then avoiding recomputation. *Revolve* [24] checkpointing algorithm, for example, transforms the problem into an $\mathcal{O}(n \log n)$ problem.

When using large datasets, the checkpointing data does not fit in the GPU memory, requiring the use of the host's memory as storage. Moreover, once again, the one is trying to make their application faster, finds a new hurdle with communication: the data needs to be transferred from the GPU memory to the host memory and then restored from it, which is a bottleneck in the checkpointing strategy.

This MSc. dissertation targets the described scenario and aims to reduce the Host-Device communication overhead using an RTM implementation as a case study.

1.1 Problem Definition

This research focuses on Awave-3D as a case study. Awave-3D is an RTM implementation that uses GPU acceleration and checkpointing to minimize recomputation of timesteps. Due to the limited size of the GPU DRAM, Awave-3D stores the checkpointing data in the host memory.

Awave-3D's checkpointing currently spends around 75% of the time communicating between host and GPUs to save and restore checkpointing data, which makes checkpointing in Awave-3D inefficient.

Checkpointing algorithms such as *Revolve*, *zCut*, and *Uniform* exhibit two important properties:

- **Temporal locality:** A checkpoint is often used multiple times in rapid succession.
- **Determinism:** The checkpointing trace (i.e., when to save and restore each checkpoint) is predictable in advance and can be simulated beforehand.

These properties enable two optimization opportunities:

- **Caching:** Frequently reused snapshots can be stored in GPU memory to minimize redundant data transfers.
- **Prefetching:** Since the checkpointing pattern is predictable, data can be proactively transferred from the host to the GPU while other computations are in progress.

Additionally, the blocking time can be reduced if less data is transferred through the PCI-e. For instance, if compressed data can achieve a compression ratio of 2 times, the transfer would be 2 times faster – ignoring the compression/decompression overhead. Data in RTM typically consists of large floating-point tensors, making it a suitable target for lossy compression over the lossless compression [8, 18]. Considering lossy compression, a

trade-off between compression ratio, speed, and data fidelity is an important decision that needs to be carefully considered, so extensive experimentation and research on compressors are required to achieve good results.

In summary, the traditional checkpointing strategy applied in adjoint applications is hindered by inefficient data movement. Nevertheless, the characteristics of the problem present promising opportunities for improvements through techniques such as caching, prefetching, and compression. The challenge of this work is effectively integrating these strategies while dealing with different checkpointing algorithms, memory access patterns, GPU memory limitations, performance requirements, and data fidelity constraints.

1.2 Objectives & Research Questions

The main objective of this research is to achieve overall speedup on Awave-3D by reducing the GPU-Host data movement caused by checkpointing. This research aims to explore prefetching strategies and data compression techniques to alleviate the performance bottlenecks caused by frequent data transfers.

To achieve the main objective, this dissertation must answer the following Research Questions (RQ). Specific Objectives (SO) are established for each one as ways to answer the research questions.

- **RQ1:** Can data access patterns of the targeted checkpointing algorithms (*Revolve*, *zCut*, and *Uniform*) efficiently provide enough hinting of when a checkpoint can be brought ahead of time?
  - **SO1:** Develop a caching mechanism in the GPU memory to store the prefetched data and the recently used checkpoint. Along with this objective, the caching eviction policy and a synchronization mechanism must be defined to avoid data corruption.
  - **SO2:** Write an algorithm to identify when a *raw cache miss* would occur and schedule prefetch action to retrieve the checkpoint to the cache beforehand.
  - **SO3:** Identify the cache size that best fits each checkpointing algorithm's needs.
- **RQ2:** Can data compression reduce PCIe overhead without compromising the quality of results?
  - **SO4:** Evaluate the compression ratio, quality (PSNR, SSIM, visual evaluation), and speed of the main GPU-based floating-point compressors in the bibliography.
  - **SO5:** Conduct a sensitivity analysis on the relationship between compression ratio and data quality for the chosen compressors to identify parameter settings that optimize performance while maintaining acceptable fidelity in the datasets used in this research.

- **RQ3:** Can compression and prefetching mechanisms be combined inside the GPU to achieve better speedup?
  - **SO6:** Evaluate prefetching (SO1, SO2, SO3) and compression mechanisms (SO4, SO5) combined.

1.3 Contributions

This research results in GPUZIP, an independent library from Awave-3D that wraps all the source code written for this research for prefetching and compressing checkpointing data. GPUZIP was designed to be extensible and support beyond this dissertation's compressors and checkpointing algorithms. GPUZIP is available on GitHub [3][1]; it offers APIs in C++/Cuda, and the compression module has a Python wrapper.

This research contributed to academic production by:

- Poster presented at Super Computing 2023 in Denver, CO, USA [45].
  This poster introduced GPUZIP, the combination of prefetching of checkpointing data using *Revolve* and compression with cuZFP compressor; The experimental results show speedups of $1.98 - 2.53\times$.
- Paper accepted on Euro-PAR 2024 in Madrid, Spain [44]
  This paper matures the prefetching and compression combination by introducing one more compressor (NVIDIA's NVCOMP Bitcomp) and detailing the methodology adopted on GPUZIP. Speedups were achieved at $3.45 - 3.90\times$.
- Article published in the International Journal of High-Performance Computing Applications (IJHPCA), 2025 [46]
  This article introduces GPUZIP 2.0. It involves the GPU cache mechanism and the prefetching algorithm, which allows speedup gains and extends the prefetching algorithm to other checkpointing algorithms (*zCut* and *Uniform*). The speedups achieved were $5.12\times$ for *Revolve*, $8.9\times$ for *zCut*, and $5.8\times$ for *Uniform*.

Committed to transparency, this work stores the datasets (inputs) used in the experiments, along with the raw results presented in the charts and tables, in REDU (*Repositório de Dados de Pesquisa da Unicamp*) [47]. The repository includes comprehensive documentation on accessing and interpreting the data files, log files, experimental inputs and outputs, and NSight trace files. In addition to supporting reproducibility, this dataset is intended to serve as an input resource for future researchers and students working on RTM, since finding datasets with all geometry and parameter configurations is not trivial.

1.4 Text Organization

This manuscript outlines the latest version of the research that led to the development of GPUZIP. Chapter 2 provides the background context for this research, introducing concepts such as RTM, checkpointing algorithms, the computer architecture and organization

[1] Pending approval for open-sourcing under MIT license.

of heterogeneous computing systems, and an overview of Awave-3D. Chapter 3 presents a literature review, highlighting findings from the academic community that support our specific objectives and detailing the differences between GPUZIP and similar works.

Armed with sufficient background knowledge and an understanding of related research, this manuscript turns to the aims of the current study, which are presented in self-contained chapters. Chapter 4 discusses the checkpoint prefetching methodology and evaluates experimental results, focusing solely on the isolated prefetching mechanism. Chapter 5 follows with the methodology and evaluation of checkpoint data compression, excluding any prefetching mechanisms, to assess the actual impact of compression on the outcomes. Chapter 6 details the current state of GPUZIP, exploring the combination of prefetching and compression, how these modules interact, and the resulting experimental findings. Finally, Chapter 7 provides an in-depth look at GPUZIP as an open-source software library, including instructions on using GPUZIP in other projects. The manuscript concludes with the Chapter 8, which outlines the limitations of this work, addresses each research question, and summarizes the study's findings.

Chapter 2

Background

This chapter provides the necessary theoretical and technical background to support the design decisions, optimizations, and evaluations presented throughout this dissertation. It is organized as follows: Section 2.1 introduces the principles of seismic imaging and the RTM algorithm. Section 2.2 discusses checkpointing techniques, crucial for memory optimization in adjoint computations. Section 2.3 reviews the architecture of heterogeneous systems, focusing on GPU programming and its associated memory and execution models. Finally, Section 2.4 presents the Awave-3D, an RTM implementation used in this dissertation.

2.1 Seismic Imaging & RTM

Seismic imaging is a fundamental technique in exploration geophysics, used to generate detailed subsurface models of the Earth. It plays a key role in resource exploration, including oil, natural gas, and mineral deposits. The process begins with seismic surveys, where a controlled energy *source* generates waves that travel through geological layers. On land, this source is typically a vibroseis truck or dynamite explosion, while in marine environments, an air gun releases compressed air, as illustrated in Figure 2.1. These waves reflect off subsurface structures and are recorded by *receivers* that may be geophones (on land) or hydrophones (at sea), forming seismic traces [25]. Multiple *shots* of vibration are recorded at different locations to cover a broader surface region.

To accurately reconstruct the subsurface, seismic data undergoes *migration*, a process that corrects distortions caused by variations in wave velocity across different geological layers. Migration repositions reflectors at their true depths, converting seismic recordings into interpretable structural images. Several migration techniques exist, each with specific trade-offs, including Kirchhoff Migration, Beam Migration, One-Way Wave Equation Migration (OWEM), and RTM [22].

RTM, introduced by Baysal *et al.* in 1983 [9], is widely recognized as one of the most accurate migration methods, particularly for complex geological structures with steep dips and strong velocity contrasts. Compared to Kirchhoff and Beam Migration, RTM better handles wavefield multipathing and sharp velocity changes. However, this accuracy comes at a significant computational cost due to its reliance on full wavefield propagation

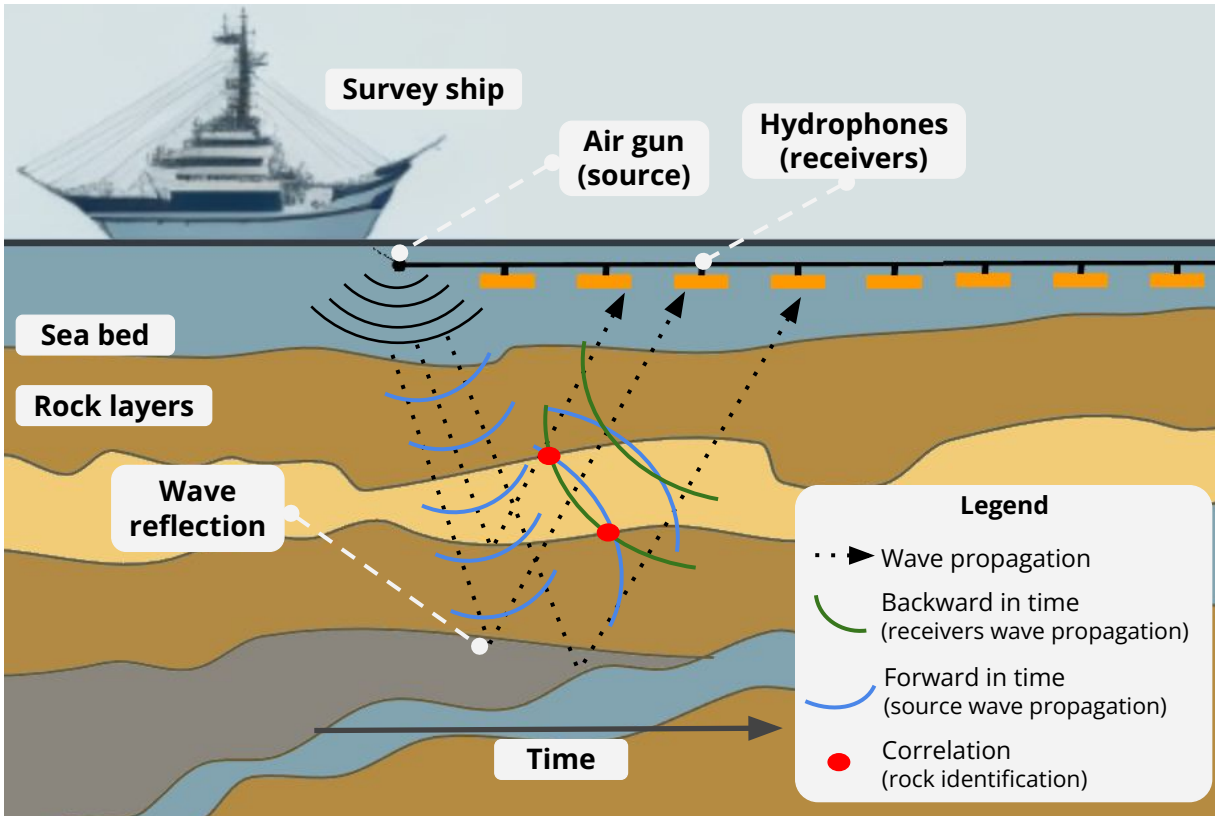


Figure 2.1: Seismic survey process. Blue arcs represent forward wave propagation, green arcs represent backward propagation, and red dots indicate crosscorrelation points.

and time-reversal techniques. RTM's imaging quality compared to other methods may be found in [22], Figure 6.

RTM operates iteratively over multiple *shots*, each representing the propagation of waves from a single seismic source to the receivers (see dotted lines in Figure 2.1). The wave equation is solved for each shot over a series of discrete time steps to model wave propagation. To do this, the continuous wave equation is discretized using the Finite Difference Method (FDM), which represents space and time as a structured computational grid. In this numerical approach, the wavefield at each grid point is updated iteratively based on values from previous time steps [22].

Once the wave equation is discretized, the RTM process unfolds in two main phases. In the **Forward Phase**, the wavefield is computed forward in time from the source location (see Figure 2.1, blue arches). Then the **Backward Phase** begins; here, the recorded seismic data is injected at receiver positions and propagated in reverse time (see Figure 2.1, green arches). At each step, the backward-propagated wavefield is cross-correlated with the stored forward wavefield to construct the final migrated image (see Figure 2.1, red dots) [22].

The cross-correlation step presents a computational challenge beyond the inherent cost of solving the wave equation with FDM: both forward and backward wavefields must be available in memory simultaneously. Storing full wavefield snapshots requires substantial memory, making large-scale RTM computations impractical without optimization. To address this, researchers have developed strategies such as wavefield reconstruction from

boundary conditions and checkpointing methods, which aim to reduce storage requirements while preserving computational accuracy [20, 64].

2.2 Checkpointing

As introduced in Subsection 2.1, one of the challenges in RTM computation is ensuring that the results from a given timestep in the forward computation are available during the backward phase for cross-correlation. A naive approach to address this challenge is presented in Algorithm 1, where each timestep in the backward phase (line 2) recomputes the forward phase entirely (line 4), resulting in an algorithm with $\mathcal{O}(timesteps^2)$ complexity [64]; which highly inefficient in terms of processing time, although storage-friendly since it discards the forward data after backward computation and cross-correlation.

Algorithm 1 Naive forward and backward phases with cross-correlation. Algorithm complexity: $\mathcal{O}(timesteps^2)$.

```
1: Initialize bwdData;
2: for tsBwd = timesteps − 1; tsBwd ≥ 0; tsBwd = tsBwd − 1 do
3:    Initialize fwdData;
4:    for tsFwd = 1; tsFwd ≤ tsBwd; tsFwd = tsFwd + 1 do
5:       ComputeForward(fwdData, tsFwd);
6:    end for
7:    ComputeBackward(bwdData, tsBwd);
8:    Crosscorrelate(fwdData, bwdData);
9: end for=0
```

To reduce the algorithm complexity, the program could run the forward loop once and save all results (snapshots) for later cross-correlation during the backward phase, which is impractical due to memory constraints. For instance, the result of a single iteration of the Marmousi3D dataset requires approximately $500MB$, with $6,751$ timesteps, the total storage demand would exceed $3.2TB$, making it infeasible. Even if larger storage solutions like HDDs or SSDs were used, the overhead of transferring data could be more costly than recomputing it on accelerators like GPUs. To address this, checkpointing aims to balance the extremes of full recomputation and full storage, optimizing memory consumption while minimizing redundant computations. The effectiveness of each technique depends on factors like memory availability and hardware constraints [64].

Beyond defining the ideal number of snapshots and recomputation needed to solve the problem, checkpointing algorithms orchestrate what and when the timesteps should be processed backward and forward, and when a checkpoint should be taken or restored. To simplify the understanding, this dissertation provides a unified terminology: `FORWARD` (compute forward), `SAVE` (store a checkpoint), `RESTORE` (re-materialize a checkpoint), and `BACKWARD` (compute the backward).

Figure 2.2 illustrates checkpointing in action with only five timesteps. Initially, at (1), the process computes forward through timesteps, saving snapshots (c1) and (c2). The

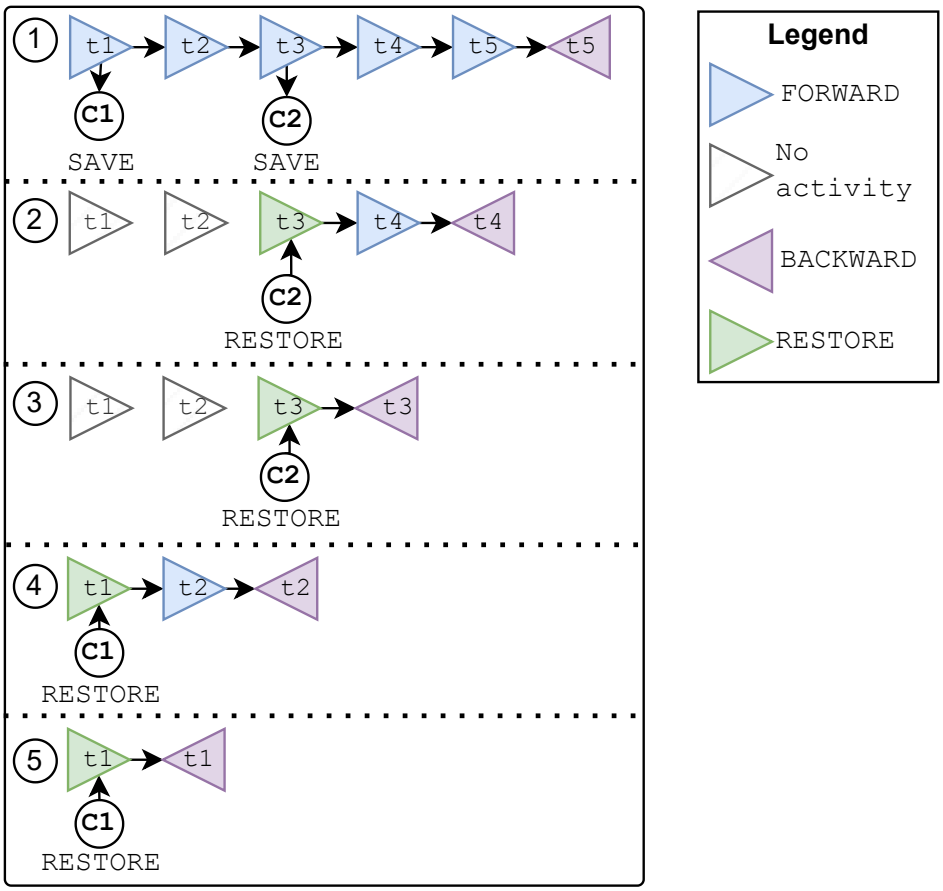


Figure 2.2: Example of forward and backward checkpointing in adjoint computation with five timesteps and two snapshots.

forward computation continues until `t5`, after which backward computation begins using available forward-phase data.

At ②, if no checkpointing were used, the entire forward computation from `t1` to `t4` would need to be recomputed. However, checkpoint (c2) allows the re-materialization of `t3`, then only recomputation of `t4` is required, avoiding redundant calculations. The process continues similarly with snapshots (c1) and (c2). Without checkpointing, this example would require 15 forward computations, while with checkpointing, only seven are needed, highlighting the method's effectiveness in reducing redundant work.

This dissertation concentrates on three checkpointing algorithms: *Revolve* [24], *Uniform*, and *zCut* [68]; however, the goal of this research is not to determine the best algorithm but to demonstrate that the techniques implemented in GPUZIP are adaptable across different checkpointing approaches. The three checkpointing algorithms used are:

- ***Revolve*** [24] is a well-established checkpointing algorithm, originally proposed in 1992 [23] and later implemented as a C-library in 2000 [24]. Even after three decades, it remains widely used, for instance, in the *Devito Framework* [42, 43], a popular open-source toolbox for finite difference problems, image processing, and machine learning.

  Revolve optimizes the problem presented in Algorithm 1, reducing its complexity

from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$. It implements a *bisection-based checkpointing* strategy, where the number and placement of snapshots are computed recursively, requiring about $\log_4(\text{timesteps})$ checkpoints for optimal efficiency. The algorithm begins with a `FORWARD` computation, storing the first snapshot at $(\frac{\text{timesteps}}{2})$, then recursively saving additional checkpoints at midpoints according to the bisection scheme until all are placed. During the `BACKWARD` computation, *Revolve* restores the closest checkpoint preceding the desired `BACKWARD` timestep, recomputes `FORWARD` from it, and continues until the target state is reached. When a checkpoint is no longer required, its slot is reused for a new snapshot at a strategically chosen point, ensuring minimal recomputation and optimal use of the available checkpoint memory.

- ***Uniform*** is a straightforward checkpointing strategy that takes snapshots evenly across the execution timeline. The user defines the maximum number of snapshots based on available memory, and the algorithm calculates uniform intervals for saving and restoring them. For example, in a simulation with 6,751 timesteps and a storage capacity for 100 snapshots, *Uniform* would place a checkpoint approximately every 67 timesteps. The *Uniform* algorithm used in this work is an in-house developed code; however, the idea of "uniformly spaced checkpointing" is not a novelty [4].

- ***zCut*** [68] algorithm tackles the problem by representing it as a directed acyclic graph, known as a computational graph. In this graph, nodes represent computational tasks (kernels), and edges show the data dependencies between these tasks. It determines which nodes should be saved. The nodes not saved during the forward phase must be recomputed from the most recent checkpoint during the backward phase. *zCut* takes snapshots for all nodes for a given interval of execution, then after this chunk is completed, the nodes are discarded and moves to the next chunk.

Despite their differences and trade-offs, *Revolve*, *zCut*, and *Uniform* share common traits. They are deterministic, ensuring identical inputs produce the same checkpoint sequence. Additionally, all three rely on a structured control loop to dictate execution steps: `FORWARD`, `SAVE`, `RESTORE`, `BACKWARD`, and `TERMINATE`, making them systematic and predictable solutions for large-scale adjoint computations.

2.3 Hegerogeneous Computing Systems & GPUs

Over the last two decades, the semiconductor industry and the field of computer science have increasingly focused on parallel computing architectures to address the growing demand for high-performance computing [26]. For numerically intensive applications like RTM, which rely heavily on floating-point operations, heterogeneous computing systems that combine traditional CPUs with accelerators like GPUs or FPGAs have become widely adopted [20, 40, 60].

GPUs have emerged as massively parallel devices capable of achieving impressive performance. For instance, an NVIDIA V100 GPU can reach up to 15.7 teraflops in FP32 and 7.8 teraflops in FP64 [14]. Newer NVIDIA architectures continue to push these limits, with the A100 delivering up to 19.5 teraflops in FP32 [15], the H100 reaching

approximately 67 teraflops in FP32 [16], and the B100 achieving around 80 teraflops in FP32 [17]. NVIDIA competing vendors also offer high-performance accelerators, such as AMD's MI300X GPUs [6] and Intel's Data Center GPU Max series [32], both of which target HPC and AI workloads. In contrast, a high-end CPU like the Intel Xeon Gold 6240 delivers around 0.921 teraflops [33]. This raw comparison suggests a potential 15×–85× performance gap, but each processor is designed for a different goal: CPUs are optimized for low-latency sequential tasks, while GPUs prioritize high-throughput parallelism [31].

To better understand this distinction, Figure 2.3 illustrates a typical heterogeneous computing system composed of a GPU ("Device") and a CPU ("Host") connected via the PCIe bus. While CPUs consist of a few cores, each with its own control, cache, and ALU (Arithmetic Logic Unit), the GPU consists of thousands of ALUs, allowing it to process many threads simultaneously. For example, the NVIDIA V100 features 5120 FP32 cores and 2560 FP64 cores [14]. This dense arrangement of ALUs and the GPU's access to high-bandwidth memory (HBM2) enables it to perform floating-point arithmetic much faster than CPUs [31].

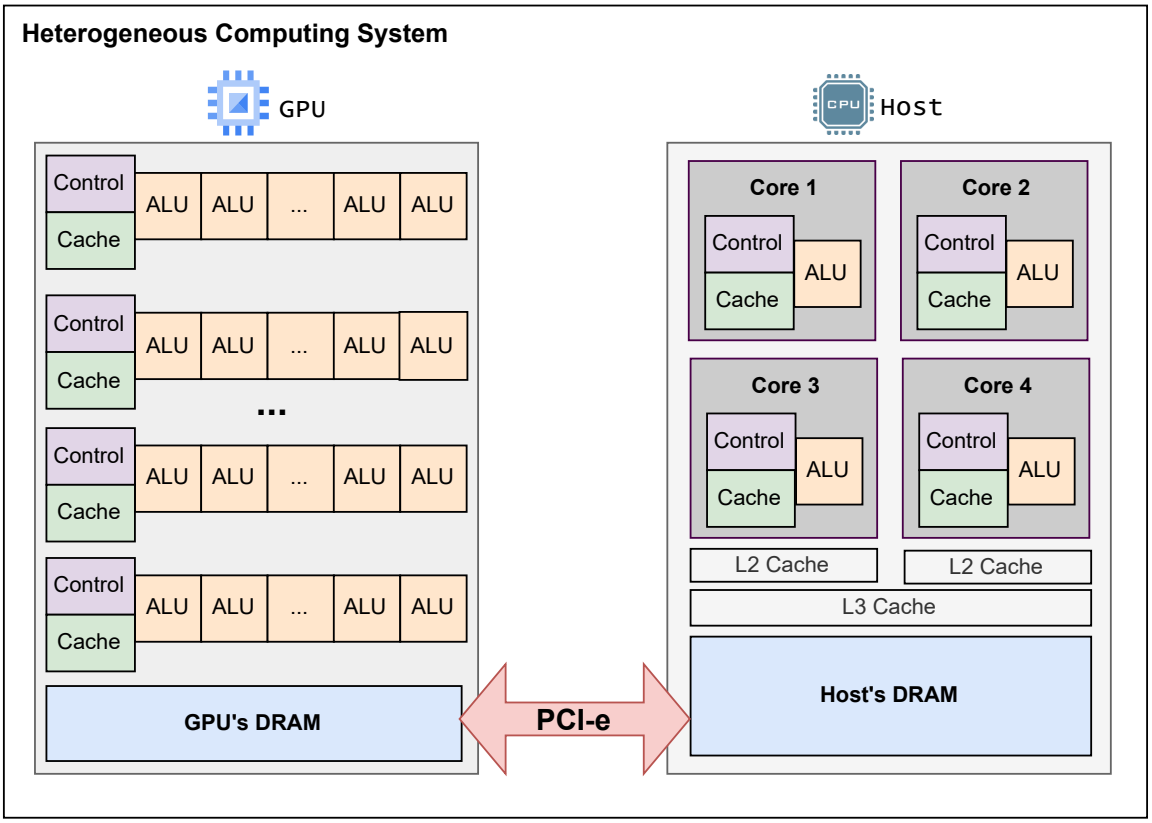


Figure 2.3: Diagram of a Heterogeneous Computing System composed of a GPU and CPU.

GPUs' high throughput stems from their ability to exploit massive parallelism through the SIMT (Single Instruction, Multiple Threads) execution model. SIMT shares some characteristics with SIMD (Single Instruction, Multiple Data; from Flynn's taxonomy), as both allow the execution of the same instruction across multiple data elements. However, SIMT differs by combining this parallelism with independent thread control, allowing not only arithmetic operations but also control flow and branching logic within each

thread [57].

GPUs are managed by programming models such as OpenCL or NVIDIA's CUDA. Both support the C/C++ environment and have their own directives and extensions to allow the development of programs that use GPUs [31]. This dissertation concentrates on the CUDA programming model.

CUDA introduces two essential concepts: thread blocks and grids. A group of threads is called a "thread block" and can work together using fast on-chip shared memory and synchronize with each other. A grid is formed by collecting these thread blocks covering the entire input data [57].

From the programmer's perspective, CUDA separates code that runs on the host (CPU) from code that runs on the device (GPU). CUDA kernels are functions marked with the `__global__` keyword launched from the host and executed in parallel on the GPU. The example in Listing 7.1 shows a simple kernel that adds two vectors element-wise.

```
__global__ void vecAddKernel(float *a, float*b, float*c, int n) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n) {
       c[i] = a[i] + b[i];
    }
}

main() {
   float *A_h, *B_h, *C_h, *A_d, *B_d, *C_d;

   int na = readFile(&A_h, "./a.bin"); // returns vector size
   int nb = readFile(&B_h, "./b.bin"); // returns vector size

   if (na != nb) { fprintf(stderr, "Array size mismatch\n"); return 1;}

   int n = n_a;
   int size = n * sizeof(float);
   C_h = (float *)malloc(size);

   cudaMalloc((void **) &A_d, size);
   cudaMalloc((void **) &B_d, size);
   cudaMalloc((void **) &C_d, size);

   cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
   cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);

   vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);

   // C_h can be used by the host after d2h. C_d is used only in GPU.
   cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);

   cudaFree(A_d); cudaFree(B_d); cudaFree(C_d);
   free(A_h);free(B_h);free(C_h);
}
```

Listing 2.1: CUDA kernel and host code for element-wise vector addition. This is an adapted version from the example given in Hwu, Wen-Mei W *et al.* (2022) [31]

The host code handles memory allocation (`cudaMalloc`), data transfer (`cudaMemcpy`), and invokes kernel using the triple angle bracket syntax to run in the device: `<<<blocks, threads>>>` [57]. For example, with $n = 1000$ and 256 threads per block, the launch configuration `<<<4, 256>>>` creates 1024 threads. The extra 24 threads are discarded using the guard condition `if (i < n)` inside the kernel [31].

CUDA also allows device memory allocation and data movement between the host and the device [57]. In the example from Listing 7.1, memory allocations occur in lines 20–22, *Host-to-Device* transfers in lines 24–25, and the result is copied back to the host (*Device-to-Host*) in line 30. These operations are synchronous by default, meaning the GPU must wait idle during transfers.

Even though GPUs have impressive FLOPS capacity and potential for massive parallelism, developing applications for heterogeneous computer systems can be challenging. One significant hurdle is the high latency communication between devices, which can shatter the dreams of accelerating applications with GPUs [31, 40, 57].

As illustrated in Figure 2.3, *Device-to-Host* and *Host-to-Device* communication happens over the PCIe bus [31], which is much slower than accessing GPU-local memory. For instance, NVIDIA v100 (with PCIe 3.0) offers a host-device bandwidth of $32GB/s$, while access to HBM2 memory (GPU's DRAM) achieves $900GB/s$ — a staggering 28× difference [14].

CUDA provides asynchronous data transfer mechanisms to minimize idle time using `cudaStream`. With `cudaMemcpyAsync`, data transfer can occur concurrently with kernel execution, since they are assigned to different `cudaStreams`. CUDA offers synchronization mechanisms like `cudaStreamSynchronize` to control execution without blocking all streams as `cudaDeviceSynchronize` would [31, 57].

In summary, parallel programming accelerators offer significant opportunities to accelerate applications. However, as Amdahl's laws suggest [26], programmers must know when and how to use them to accelerate their programs, and which part of the program, respecting the system and the application's requirements and limitations.

2.4 Awave-3D

Awave-3D is RTM implementation that solves 3D fields. It was developed through a collaborative effort between the Faculty of Geophysics[1] at Pará Federal University (UFPA), the Institute of Mathematics, Statistics and Scientific Computing (IMECC)[2], and the Institute of Computing (IC)[3] at Unicamp. It was designed as a research-oriented platform to explore and advance computational strategies for seismic imaging and wavefield modeling.

Awave-3D supports a variety of computational flavors tailored to different levels of hardware complexity and parallelism: CPU-only; GPU (single device); **SNMG** – *Single-Node Multi-GPU*, which decomposes the wavefield across multiple GPUs on the same

[1] https://cpgf.propesp.ufpa.br/index.php/en/ - Accessed May 26, 2025
[2] https://www.ime.unicamp.br/en - Accessed May 26, 2025
[3] https://ic.unicamp.br/en/ - Accessed May 26, 2025

machine; and **MNMG** – *Multi-Node Multi-GPU*, which extends SNMG and distributes shots across nodes using intra-node GPU decomposition.

All these versions incorporate checkpointing, initially powered by the *Revolve* algorithm, to optimize memory usage during wavefield reconstruction. For this work, however, Awave-3D was refactored to support interchangeable checkpointing libraries, as Chapter 7 will discuss further.

The storage used for checkpointing data in Awave-3D is organized as shown in Figure 2.4: when a `SAVE` ① operation occurs, data is transferred from GPU memory to host memory (*Device-to-Host*). Then, during a `RESTORE` ② operation, data is fetched back from host memory to GPU memory (*Host-to-Device*).

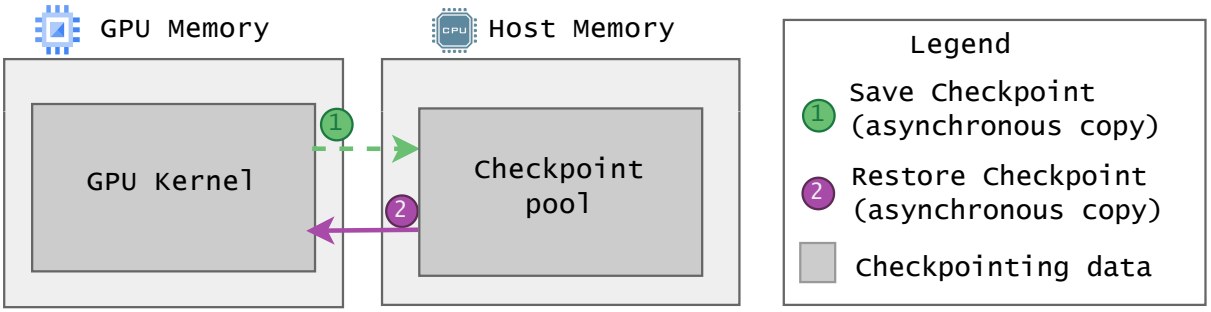


Figure 2.4: Baseline memory architecture for checkpointing in Awave-3D.

In SNMG, the computational domain is spatially decomposed across multiple GPUs. Each GPU handles a portion of the wavefield using the following CUDA kernels:

- `InjectionForward`: injects the seismic source into the forward wavefield.
- `UpdateForward`: propagates the wavefield forward in time using finite-difference stencils.
- `InjectionBackward`: injects the receiver wavefield into the backward propagation pass.
- `UpdateBackward`: performs the time-reversed propagation for imaging.

This spatial decomposition allows each GPU to compute its subdomain while exchanging boundary information with neighboring GPUs as needed.

The Awave-3D's MNMG flavor extends the SNMG approach by distributing *shots* across multiple compute nodes. This distribution is implemented using OpenMP Cluster (OMPC) [71], a library developed by the Computer Systems Laboratory (LSC)[4]. OMPC builds on the familiar OpenMP programming model, adding transparent inter-node communication via MPI to eliminate the need for programmers to explicitly write MPI code. Importantly, OMPC is used solely for distributing workloads across the cluster's nodes—once the data reaches a node, all computation is performed using CUDA kernels. In Awave-3D, OMPC directives distribute shots from the central node to the worker nodes (e.g., `#pragma omp parallel`), after which each node launches asynchronous CUDA tasks

[4] https://lsc.ic.unicamp.br/ — Accessed May 26, 2025

using `#pragma omp target nowait map(to:...)`[5][6]. Each node then processes its assigned workload using the SNMG decomposition strategy. Finally, a synchronization phase aggregates the partial results to produce the final migrated image.

2.5 Chapter Conclusion

This chapter established the theoretical and technical foundations for the research presented in this dissertation. The discussion began with an introduction to seismic imaging and the RTM method, emphasizing its suitability for complex geological structures and the computational challenges it presents, particularly in terms of storing and reconstructing the wavefield. Checkpointing strategies were then reviewed, including the *Revolve*, *Uniform*, and *zCut* algorithms, which aim to balance memory usage and recomputation.

Subsequently, heterogeneous computing systems were examined, with particular attention to NVIDIA GPU architectures, their execution models, memory hierarchies, and data transfer mechanisms, all of which are critical for high-performance RTM implementations. Awave-3D was then described, detailing its multi-level parallelization strategies and integration of checkpointing methods.

The concepts and technologies discussed in this chapter form the basis for the design decisions, optimizations, and experimental methodologies explored in the subsequent chapters.

[5] https://ompcluster.readthedocs.io/en/latest/programming.html#asynchronous-target-task — Accessed May 26, 2025

[6] https://ompcluster.readthedocs.io/en/latest/programming.html#manage-the-device-data-environment — Accessed May 26, 2025

Chapter 3

Related Work

This chapter reviews the relevant literature on the techniques discussed in this dissertation. GPUZIP introduces a novel approach by integrating GPU-based lossy compression with prefetching checkpoint data from the host to GPU memory. It works seamlessly with different checkpointing algorithms and allows users to fine-tune through compression and prefetching parameters. Prior research has typically treated compression and prefetching as distinct processes, often using different methodologies or focusing on other objectives.

This chapter is organized as follows: Section 3.1 presents a review of the relevant literature related to this work, while Section 3.2 distinguishes features of the proposed approach.

3.1 Literature Review

This literature review examines previous research on the application of compression and prefetching techniques in scientific computing and high-performance computing workflows. First, it discusses the application of data compression in scientific computing, highlighting the success of using lossy compression to reduce communication bottlenecks. Additionally, the review examines prefetching, caching, and asynchronous data transferring strategies across various software layers, highlighting their role in mitigating data movement issues. Together, these topics provide the theoretical and practical context for the methods developed in this dissertation.

Data compression has long served as a fundamental technique for reducing storage requirements, with applications spanning file systems, audio, video, and image processing [72]. The relevance of data compression extends beyond storage efficiency to mitigating communication bottlenecks by decreasing the volume of data prior to transmission, not only in HPC but in multiple computing systems fields. Early studies, such as Mogul *et al.* [56], demonstrated network traffic reduction in HTTP through compression. More recent works in HPC highlight its benefits in reducing CPU and GPU memory transfers [61, 62] and in optimizing multi-stage I/O pipelines, where compressed data flows from network storage to local SSDs and then into main memory and GPU memory [52].

Within scientific computing, the choice between lossless and lossy compression has been widely discussed. Lossless methods ensure exact data recovery but typically offer

limited compression ratios, while lossy compression provides a bigger compression ratio and, when well parametrized, guarantees good data quality [12, 18, 41, 72]. Lossy approaches have gained traction in multiple scientific computing domains; modern GPU and CPU compressors, such as FZ-GPU [72], cuSZp [29], SZ3 [38], and cuZFP [39], have been evaluated across diverse domains, including in their initial work results for climate, cosmology, quantum structure, and RTM data.

Several CPU-based approaches exploit this trade-off to reduce recomputation and I/O costs. Margetis *et al.* [48, 49] applied lossy compression on the CPU to store additional wavefield snapshots. Huang *et al.* [30] developed a CPU framework that compresses forward wavefields after each timestep and decompresses them during backward propagation, alleviating I/O bottlenecks and achieving up to 6.6× speedup. In parallel, GPU-based methods have been shown to enhance efficiency further. Kukreja *et al.* [37] demonstrated that GPU compression increases the number of checkpoints storable within limited GPU memory. Dmitriev *et al.* [19] used cuZFP and NVIDIA Bitcomp to compress seismic data while maintaining imaging quality, and Barbosa and Coutinho [8] compared lossy and lossless methods in RTM, revealing significant reductions in storage and I/O overhead. Shen *et al.* [61, 62] additionally reported up to 21% reductions in host–device transfer times and memory usage, resulting in 2.09× speedups for FP32 datasets. Maurya *et al.* [51] advanced this direction by proposing a GPU-resident pipeline that accumulates compressed data before host transfer, exemplifying the integration of compression directly into scientific workflows—an approach closely aligned with the methods adopted in this dissertation.

Prefetching, another relevant performance optimization, is a well-established latency-hiding technique that proactively stages data from slower to faster storage tiers, thereby reducing access delays and avoiding execution stalls [55,69]. It may be implemented across both hardware and software layers. At the hardware level, prefetching operates within CPU cache hierarchies (L1, L2, L3) to preload instructions and data [21, 26, 55]. In contrast, at higher system levels, it stages data from disks into DRAM [11,59] or coordinates multi-level transfers from disk to host memory and onward to GPU memory [52]. Liu *et al.* [41] reported up to 2× speedups in distributed DNN workloads by combining GPU data prefetching with a reuse-distance-based cache eviction policy and additional load-balancing mechanisms. Other works explore asynchronous copy operations between host and GPU memory to hide communication latency, even without explicit pattern-based prefetching [34, 35]. Rigon *et al.* [60] demonstrated up to 62% performance gains in seismic applications through asynchronous prefetching and Unified Memory optimizations, underscoring the general effectiveness of proactive data movement strategies.

Checkpoint-specific prefetching has also been explored. Maurya *et al.* [52] implemented a multi-level storage prefetching mechanism in the VeloC[1] checkpoint library, which supports multi-level checkpoint-restart runtimes. Unlike *Revolve* or *zCut*, which algorithmically determine checkpoint timing, VeloC allows user-defined snapshot placement. Building on this flexibility, their method prefetches checkpoints from lower storage tiers (e.g., CPU memory) to higher ones (e.g., GPU memory) based on execution hints, with RTM serving as their primary case study. Insights from these prefetching approaches

[1] https://veloc.readthedocs.io/ - accessed May 26, 2025

informed the design of the hinted-based prefetching mechanism, which includes an LRU cache, as presented in this dissertation.

3.2 This work

GPUZIP proposes communication optimization in GPU-based adjoint/reverse-mode solver implementations with checkpointing by combining checkpoint prefetching and lossy data compression on the GPU side. It introduces a checkpoint cache system and a prediction algorithm that can avoid cache misses by scheduling anticipated asynchronous retrievals of checkpoint data from the host to GPU memory ahead of time. GPUZIP applies GPU-based lossy compression to checkpoint data already residing in GPU memory, reducing the volume of data transferred over PCIe. The prefetching and compression techniques are modular and can be used independently or in combination, providing flexible configuration through the GPUZIP API.

One of the key features of GPUZIP is its ability to integrate with existing checkpointing libraries seamlessly. It accomplishes this by wrapping these libraries and conducting a dry-run simulation, which generates an optimized prefetch schedule. This functionality has been tested with multiple checkpointing strategies across various deterministic algorithms, including *Revolve*, *zCut*, and *Uniform*.

While many prior works have explored prefetching or compression independently, fewer studies combine both techniques in the context of checkpointing. Maurya *et al.* (2023a) [52] proposed a system incorporating prefetching and compression, leveraging the VELOC library. Their work addresses heterogeneous memory hierarchies and schedules data movement across storage tiers, including network storage, SSDs, host DRAM, and GPU memory. Although their architecture includes compression in the pipeline (as shown in Figure 3 of their paper), it does not isolate the impact of compression versus prefetching, nor does it explain how their prefetching mechanism would behave in the absence of compression.

Furthermore, the checkpointing library (VeloC) used by Maurya *et al.* (2023a) [52] does not include algorithms specifically designed for reverse-mode or adjoint computations, where minimizing recomputation is critical, as achieved by *Revolve*, *zCut*, or *Uniform*. In VeloC, the responsibility for deciding when to save or restore checkpoint data is left entirely to the user, and the framework does not orchestrate application execution. In contrast, this dissertation automates the prefetching process by conducting a dry-run simulation of the selected checkpointing algorithm (*Revolve*, *zCut*, or *Uniform*), producing a prefetching schedule ahead of execution.

3.3 Chapter Conclusion

This chapter reviews related work, noting that these techniques are typically used separately. It introduces GPUZIP, highlighting its novelty in combining GPU-based lossy compression with checkpoint prefetching in a modular, algorithm-independent way. The next chapter will detail the prefetching & caching methodology.

Chapter 4

Prefetching Checkpoint Data

Prefetching is a well-established technique in computer systems that improves performance by proactively moving data from slower to faster storage tiers, thereby minimizing latency and avoiding execution stalls [55,69]. In the context of this work, the prefetching technique aims to transfer checkpoint data from host memory to GPU memory ahead of its use, mitigating bottlenecks caused by synchronous data copying and improving overall system efficiency.

Prefetching mechanisms typically rely on *hinting*, where an algorithm either recognizes memory access patterns or has prior knowledge of future data requirements [52,59]. When prefetching is implemented, a caching or buffering mechanism is often required [11,52,59] to store prefetched data temporarily until it is accessed. The implementation complexity of a caching mechanism relies on the system's characteristics and application. If prefetched data is guaranteed not to be overwritten, minimal control is required. However, if storage must compete with other processes or memory constraints, replacement policies such as Least Recently Used (LRU) and Most Recently Used (MRU) become essential [11,13].

This dissertation adopts terminology similar to Emma *et al.* (2005) [21] and Ho *et al.* (2025) [27], standardizing concepts and facilitating evaluation. First, *raw misses* refer to cache misses that would occur without prefetching, while *misses* denote the actual cache misses that happen during execution with prefetching; a *real hit* is when the data is already in the cache but no prefetch was needed and *hit* is when the checkpoint is in the cache at the moment it need to be used. *Timely prefetches* are the ones that could finish their transfers before their usage, and *late prefetches* refers to the ones that could not be copied in time before their usage (opposite to timely prefetches). Finally, *timeliness* describes the time interval between initiating a prefetch and its actual use, which determines whether the data is available just in time.

One key indicator of prefetching efficiency in this work is the relationship between *raw misses* (misses that would occur without prefetching) and actual *misses* observed during execution. Ideally, the number of raw misses should be greater than the number of observed misses, indicating that prefetching effectively reduces the frequency of cache misses rather than introducing new ones.

Additionally, the blocking time associated with a raw miss should not exceed the blocking time of prefetching it. In other words, a prefetch must be issued with sufficient

timeliness to hide the data transfer latency. This can be expressed as:

$$T_{\text{timeliness}} > T_{\text{h2d}}$$

where $T_{\text{timeliness}}$ is the time between the prefetch being issued and the checkpoint being consumed, and T_{h2d} is the time required to transfer the checkpoint from host to device memory.

This chapter is organized as follows: Section 4.1 describes the prefetching mechanism, its components, and how it integrates into Awave-3D. And then, Section 4.2 presents the experimental results of the adopted technique in different datasets, checkpointing algorithms, and configurations.

4.1 Methodology

This section is organized into three subsections. First, Subsection 4.1.1 introduces the main components of GPUZIP's prefetching mechanism and explains how they interact within the context of the Awave-3D RTM implementation. Next, Subsection 4.1.2 provides a detailed explanation of the *GPU Checkpoint Cache* mechanism, describing its internal structure and behavior. Finally, Subsection 4.1.3 presents the *Prefetch Setup Algorithm*, outlining its steps and how it configures the prefetching process.

4.1.1 The Prefetching Mechanism

The proposed prefetching mechanism leverages the deterministic nature of checkpointing algorithms by performing a preliminary *Prefetch Setup*. The algorithm does a dry-run execution of the checkpointing algorithm to predict `SAVE` and `RESTORE` actions before actual computation begins. The *Prefetch Action Vector* (PAV), which schedules all prefetching operations, is the result of this setup.

Following Figure 4.1, the proposed memory architecture consists of a *GPU Checkpoint Cache*, where the checkpointing data is temporarily stored in the GPU memory (discussed further in Subsection 4.1.2); and the *Checkpoint Pool* stores all the checkpointing data in the host memory. All `SAVE` and `RESTORE` operations communicate directly with the *GPU Checkpoint Cache* that can retrieve or send data to the *Checkpoint Pool* in the host memory. The *Prefetch Action Vector* contains the iteration where the prefetch action is dispatched and what snapshot should be moved from the host to the GPU memory.

The basic checkpoint prefetching mechanism is explained at a high level through the Algorithm 2, representing the Awave-3D main loop performing actions from the checkpoint prefetching mechanism. The *Prefetch Setup* function (line 1) is the first step to be performed, which will return the PAV. Then, each iteration is checked whether a prefetching action is scheduled in the PAV at the current moment (line 4). If so, the prefetch dispatch method asynchronously transfers data from *Host-to-Device* (line 5) while a `FORWARD` or `BACKWARD` computation runs. When a `SAVE` operation occurs, GPUZIP first stores the data in the *GPU Checkpoint Cache* (line 8) before asynchronously saving it to the *Checkpoint Pool* (line 9). For `RESTORE` operations, GPUZIP first checks the cache for the requested

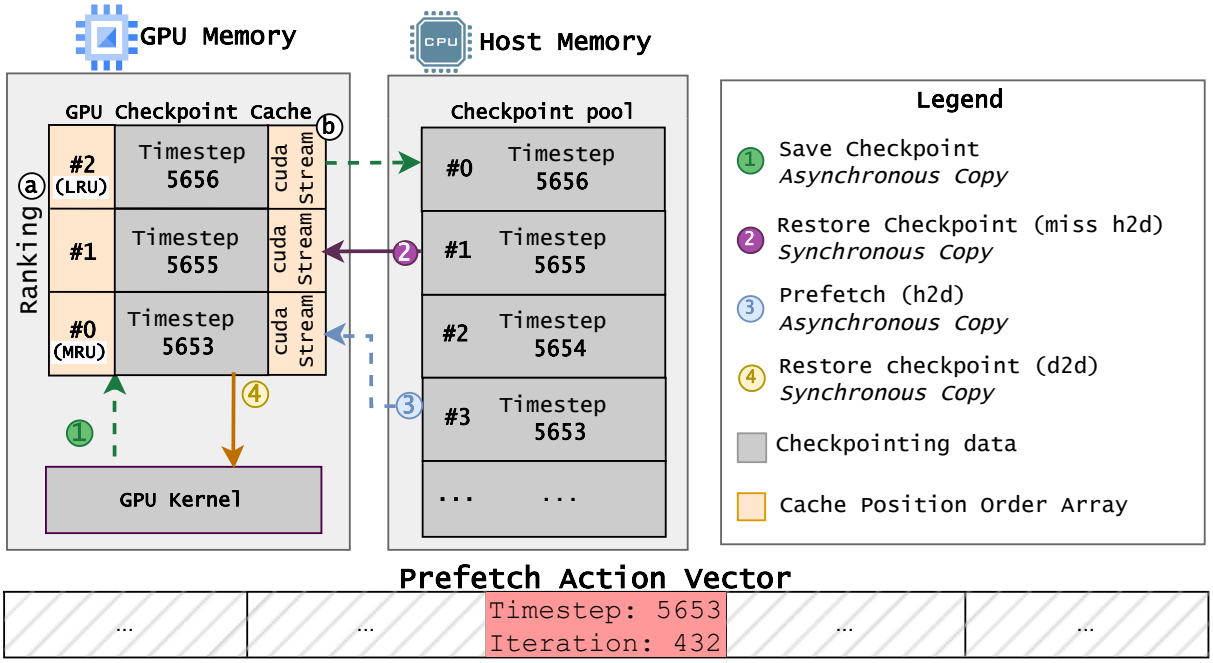


Figure 4.1: The prefetching mechanism memory architecture with a cache of 3 positions.

checkpoint (line 11). If found, a *cache hit* occurs (line 15), allowing a direct *Device-to-Device* copy to the necessary data structure. If the data is still being transferred, the system waits for the transfer to complete (lines 12 and 13). If the checkpoint is missing from the cache, the worst-case scenario occurs: a synchronous *Host-to-Device* transfer via PCIe, followed by a *Device-to-Device* copy to the required location (lines 16-19).

4.1.2 The Caching Mechanism

Caching is a widely used strategy in computer architecture that allows data to be stored for quick access. Four key questions were considered when designing the caching mechanism for GPUZIP: (A) What should the cache store? (B) How large should the cache be? (C) What is the expected access pattern? And (D) What should happen when the cache is full?

To address question (A), for every `SAVE` action, new data should be stored in the cache via a *Device-to-Device* copy, as illustrated in ① in Figure 4.1. After this, the checkpoint is asynchronously transferred from the cache to the *Checkpoint Pool* via a *Device-to-Host* operation. When a `RESTORE` ② or a prefetching ③ operation occurs, the checkpoint is loaded into the cache through a *Host-to-Device* copy. The *GPU Checkpoint Cache* is implemented as a fully associative cache [59], meaning that snapshots can be stored in any cache location regardless of their position in the *Checkpoint Pool*. The fully associative cache is exemplified in Figure 4.1, where the checkpoint at position `3` in the *Checkpoint Pool* is transferred to the position `0` in the *GPU Checkpoint Cache*.

This leads to question (B): How large should the cache be? The *GPU Checkpoint Cache* must have space for at least two snapshots. When a checkpoint is saved and copied to the *Checkpoint Pool*, another checkpoint is likely being prefetched into the cache, awaiting imminent use.

Algorithm 2 Checkpointing Prefetching Execution Loop

```
1: pav ← PrefetchSetup(config.cacheSize)
2: while action ≠ TERMINATE do
3:    action ← Checkpointing.GetAction()
4:    if pav.shouldDispatch() then
5:       pav.dispatch()
6:    end if
7:    if action == SAVE then
8:       Save computing data to cache (D2D)
9:       Save from cache to host (D2H)
10:   else if action == RETRIEVE then
11:      if currentTimestep is in cache then
12:         if transfer is still happening then
13:            Wait for the transfer to finish
14:         end if
15:         Copy from cache to computing data (D2D)
16:      else
17:         Copy from host to cache (H2D)
18:         Copy from cache to computing data (D2D)
19:      end if
20:   else if action == FORWARD or action == BACKWARD then
21:      Perform forward or backward computation
22:   end if
23: end while=0
```

Regarding access patterns (C), the chosen checkpointing algorithms likely reuse snapshots shortly after access. For instance, in the *Revolve* algorithm, the last checkpoint saved during the `FORWARD` phase is the first to be restored during the `BACKWARD` phase. Once a checkpoint is retrieved from the cache, it is often needed again because backward computation requires materializing snapshots at multiple stages.

Understanding the access pattern helps answer question (D): When the cache reaches full capacity, an existing checkpoint must be evicted to store the new one. GPUZIP employs an LRU replacement policy, meaning the least recently accessed checkpoint is removed to make room for the incoming one, as described in [11]. Whenever a checkpoint is accessed, it is promoted to the MRU position, ensuring that frequently used data remains available.

The *GPU Checkpoint Cache* includes the *Ranking Controller* (Figure 4.1, ⓐ) that tracks the order of cached checkpointing data, mapping their MRU/LRU status to specific GPU memory locations. This eliminates the need for expensive memory allocation, deallocation, or unnecessary data movement when promoting or evicting cache entries. For example, in a three-slot cache (Figure 4.1), if the checkpoint at position #2 is accessed, it is promoted to MRU (#0), shifting the previous MRU (#0) to position #1, and the previous #1 to LRU (#2).

GPUZIP assigns a `cudaStream` to every cache slot to ensure that each cache position operates independently (Figure 4.1ⓑ). This enables concurrent checkpoint operations,

preventing interference when multiple prefetching and save actions occur simultaneously. However, in cases where a `SAVE` action needs to overwrite the LRU slot while it is still transferring data to host memory, the cache enforces synchronization, ensuring that the data is fully saved before being overwritten. Although this scenario is not ideal, as it may cause computation stalls, maintaining data integrity takes priority over speed. Increasing the cache size can mitigate the issue if such delays become frequent.

This combination of independent `cudaStreams` and the ranking array provides a reliable and flexible mechanism for storing and retrieving snapshots, ensuring efficient data access and avoiding synchronization bottlenecks.

4.1.3 Prefetch Setup Algorithm

The *Prefetch Setup Algorithm* (PSA) is responsible for predicting all cache misses during execution and generating the *Prefetch Action Vector* (PAV). The PAV is a structure that stores in what checkpointing iteration a snapshot should be prefetched, for example, in Figure 4.1, the PAV says the prefetch of the checkpoint 5654 (in the *Checkpoint Pool*) will happen in the iteration 432.

To construct the PAV, PSA performs a dry run of the checkpointing execution without launching computationally expensive kernels or allocating large memory regions. By simulating the checkpointing process in this lightweight manner, PSA determines the exact state of the *GPU Checkpoint Cache* at the moment a *Cache Miss* would occur. With this knowledge, PSA preemptively schedules a prefetch action as early as possible, ensuring the requested checkpoint is available while sacrificing the *GPU Checkpoint Cache*'s LRU entry.

The *Prefetch Setup Algorithm* consists of three key components:

1. **Dummy Cache**: A dummy instance of the GPU Checkpoint Cache that mimics its behavior without actually performing GPU memory allocations;
2. **State Recorder**: A component that captures snapshots of the Dummmy Cache during iterations, enabling the system to restore a specific cache state at any point;
3. **Prefetch Action Vector (PAV)**: The algorithm's output that dictates the iterations at which prefetch actions should occur.

Algorithm 3 presents a pseudo-code representation of the *Prefetch Setup Algorithm*. It contains a main loop, such as executing the checkpointing. On every iteration, the *State Recorder* saves a snapshot (line 16) of the *GPU Checkpoint Cache* for future restoration. For `SAVE` actions, the *Cache* stores the corresponding timestep number (line 5). When a `RESTORE` action occurs, the system checks whether the requested checkpoint is already present in the *Cache* – referred to as a *Cache Hit*. If found, the checkpoint is accessed (line 13), promoted to MRU, and the cache order is updated accordingly.

In the event of a *Cache Miss*, PSA locates the iteration there the LRU checkpoint was used and then uses it as a candidate to be a good moment to dispatch a prefetch action (line 8).

Once a suitable candidate iteration is identified, a prefetch action is scheduled in the *Prefetch Action Vector*. The *State Recorder* then restores the cache state from the candidate iteration, it updates the cache to consider the prefetched data in the cache (line 10), updating the Dummy Cache's Ranking according to the prefetched data up to the current iteration.

The *Prefetch Setup Algorithm* ensures that prefetched data is never evicted before it is used. This is achieved by requiring a minimum of two cache slots, since the candidate iteration for data prefetch is the LRU iteration + 1 when a raw miss would occur; hence, the prefetched data will not be replaced while it is not used and there is an available cache position. As a result, all cache misses are prevented. Still, *late prefetches* remains a critical factor: even though the snapshot was prefetched, the data is still being copied, meaning GPU idle time with synchronization.

Algorithm 3 The *Prefetch Setup Algorithm*

```
1:  Iteration ← 0
2:  repeat
3:    action, timestep ← Chkpt.GetAction()
4:    if action is "SAVE" then
5:      DummyCache.Push(timestep)
6:    else if action is "RESTORE" then
7:      if Cache Miss then
8:        Candidate ← LRUIteration(DummyCache) + 1
9:        Schedule(PAV, Candidate, timestep)
10:       StateRecorder.Restore(Candidate, Cache)
11:       Update Cache from the Candidate
12:     else
13:       DummyCache.Touch(timestep) {Promotes cached item to MRU}
14:     end if
15:   end if
16:   StateRecorder.Snapshot(Iteration, Cache)
17:   Iteration ← Iteration + 1
18: until action is not "TERMINATE" =0
```

4.2 Experimental Results

The experiments were designed to evaluate the performance of the prefetching mechanism under varying conditions. Tests were conducted using three checkpointing algorithms — *Revolve*, *zCut*, and *Uniform* — across three datasets: Large, Marmousi3D, and Salt. M3D_Larger dataset was used to explore memory usage. Each configuration was tested using *GPU Checkpoint Cache* sizes ranging from 2 to 6.

Hardware and Software Environment. All experiments were performed on the Ogbon HPC cluster [65], maintained by SENAI CIMATEC. Each node in the cluster was equipped with the following hardware and software:

- 4× NVIDIA Tesla V100-SXM2 GPUs (32GB memory each)
- Intel Xeon Gold 6240 CPU @ 2.60GHz
- 384GB RAM (up to 340GB available for checkpointing data)
- Clang v15.0.1, CUDA 11.2, NVIDIA Driver 525.60.13
- Ubuntu 20.04 LTS.

The environment, complete with all necessary components for reproducibility, is containerized and available on Docker Hub as a public image [2].

Baseline Configuration. The baseline used for comparison corresponds to the default behavior of Awave-3D: all checkpointing data is stored in host memory and synchronously copied to the GPU during restore operations. Each checkpointing algorithm uses this default approach as its baseline. For example, the *Revolve*'s baseline runs *Revolve* without caching or prefetching; similarly, the *zCut*'s baseline is *zCut* without caching or prefetching; and the same happens for *Uniform*.

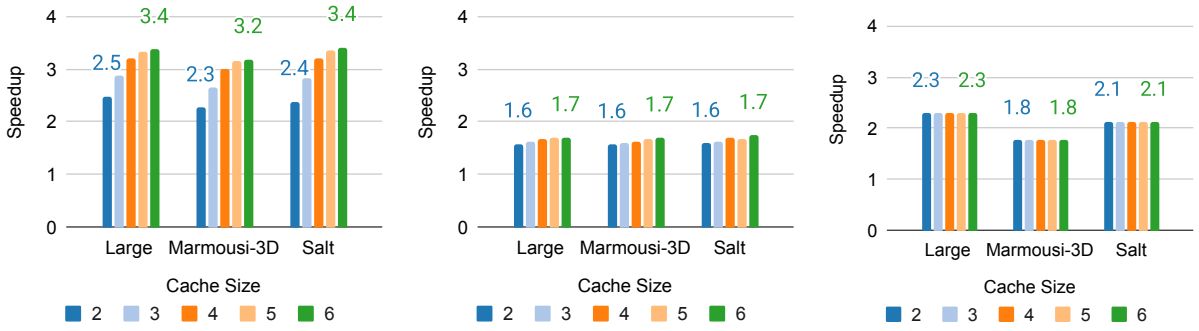
Datasets

- **Large**: In-house dataset with 3,001 timesteps and 15×14 shots, over a spatial grid of $500 \times 500 \times 500$ ($10,000 \times 10,000 \times 10,000$ meters). Migrations used 192 shots.
- **Marmousi3D**: A 3D version of the Marmousi-II model [50], with 6,751 timesteps and 13×3 shots over a $351 \times 901 \times 301$ grid ($3,510 \times 9,010 \times 3,010$ meters). Migrations used 36 shots.
- **M3D_Larger**: A variation of Marmousi3D with a larger third dimension. It features 6,751 timesteps and 13×69 shots over a $351 \times 901 \times 3,600$ grid ($3,510 \times 9,010 \times 36,000$ meters). Migrations used 88 shots.
- **Salt**: Based on the SEG/EAGE Salt and Overthrust models [7], containing 3,751 timesteps and 13×13 shots over a $310 \times 676 \times 676$ grid ($6,200 \times 13,520 \times 13,520$ meters). Migrations used 168 shots.

All datasets are available in the *Datasets* directory of the reproducibility data repository [47].

4.2.1 Overall Speedup

The first metric to be evaluated is the overall speedup. Considering the baseline as the version without any prefetching and caching mechanism, Figure 4.2 shows that using GPUZIP's checkpoint prefetching consistently improves speed, yielding speedups ranging from 2.3× to 3.4× for *Revolve*, approximately 1.6× for *zCut*, and from 1.8× to 2.3× for *Uniform*, depending on the dataset and cache size. The dataset influences the speedup result because of the number of timesteps and the checkpoint data size.



(a) Revolve (b) zCut (c) Uniform

Figure 4.2: Overall speedup achieved by *Revolve* (a), *zCut* (b), and *Uniform* (c) for each dataset (Large, Marmousi3D, and Salt) on cache sizes from 0 (baseline) to 6. The baseline is the execution of Awave-3D in its base form for the corresponding algorithm and dataset, without caching or prefetching.

Increasing the cache size influences the prefetching mechanism because with more space to store checkpointing data, the chances of a checkpoint being in the cache when a `RESTORE` occurs are higher (*real hits*). However, the speedup is highly dependent on the checkpointing algorithm in use.

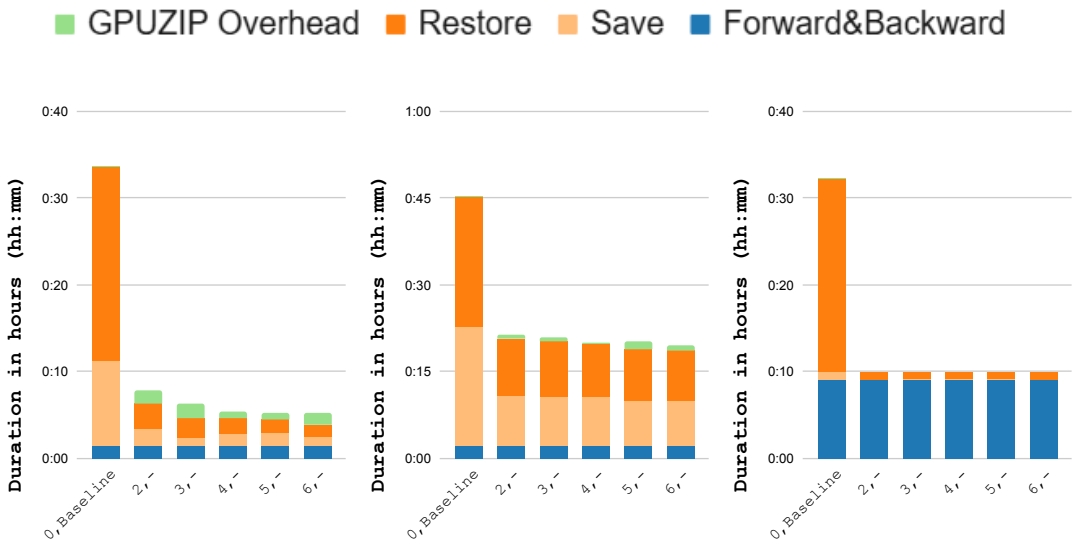
For example, the *Revolve* algorithm is the most susceptible to larger cache sizes. Processing a 'chunk' (as detailed Subsection 2.2) generates up to seven or eight snapshots (depending on the dataset) subsequent `SAVE`, which can lead to many cache replacements in a short period. This behavior removes snapshots that will likely be needed soon, as the LRU policy sacrifices the oldest checkpoint to make space for the new one. So, increasing the cache size will help make more snapshots available, causing more *real hits*. The Salt dataset reached $1.8M$ real cache hits for two cache positions, while the same dataset got $2.2M$ real cache hits with six cache positions.

In contrast, the *zCut* and *Uniform* algorithms are less sensitive to increases in cache size. *zCut* saves about 30 - 40 "intermediary snapshots" between major snapshots (depending on the dataset), which can quickly overload the cache. So, increasing the cache size from two to six does not yield significant improvements. This high demand for saving snapshots leads to delays in data transfers between the GPU and CPU while waiting for new entries to be copied.

Uniform captures snapshots only once during the forward phase and does not create additional snapshots afterward. As a result, the time between two successive snapshots can be extensive enough to make the checkpoint available when `RESTORE` is requested by the checkpointing algorithm.

The chart in Figure 4.3 illustrates how checkpoint prefetching impacts the blocking time spent on communication (orange bars) for `SAVE`, summing up the *Device-to-Host* and *Device-to-Device* copies; and `RESTORE`, summing up *Host-to-Device* and *Device-to-Device* copies.

The sensitivity to the cache size of *Revolve* is evident by the reduction in orange bars with larger caches. *zCut* also improved blocking transfer time, but bigger cache sizes (3,



(a) Revolve (b) zCut (c) Uniform

Figure 4.3: Execution time breakdown for the Salt dataset. Cache sizes from 2 to 6.

4, 5, and 6) do not improve compared to the cache size of 2. *Uniform* reduced to almost zero the blocking `RESTORE` due the long *timeliness*, the remaining orange bar refers to the *Device-to-Device* time used on restoration.

Another key observation is how asynchronous `SAVE` operations benefit from *GPU Checkpoint Cache* usage. Although `SAVE` is nominally asynchronous in the baseline, Awave-3D must wait for the transfer to complete to avoid overwriting data, blocking further computation. With the *GPU Checkpoint Cache*, *Device-to-Host* transfers are issued asynchronously, but synchronization can still occur if a new snapshot must overwrite a slot still in use. This explains the remaining `SAVE` orange bars for *Revolve* and *zCut*.

The overhead may be caused by the prefetching mechanism, and the memory used by the *GPU Checkpoint Cache* becomes a pertinent concern. The green bars in Figure 4.3 represent the overhead caused by GPUZIP in the execution time, precisely the time used for memory allocation and the *Prefetch Setup Algorithm*. The green bars demonstrate that the overhead associated with GPUZIP is justified, as it significantly reduces the blocking time for data transfers between GPU and CPU memory.

4.2.2 Prefetching Mechanism Efficiency

The prefetching mechanism achieves a 100% cache hit rate across datasets and algorithms because the checkpointing algorithms' structure allows prefetch scheduling before each `RESTORE`, which are interleaved with other tasks. However, *late prefetches* are the hurdle that still causes latencies in the *Host-to-Device* transfers, especially in *Revolve* and *zCut*. The Prefetch action is often issued too close to the checkpoint's rematerialization point, leaving insufficient time to complete the transfer before it is needed. When this occurs, CUDA stream synchronization is required, resulting in stalls. As illustrated in Figure 4.4, a prefetch is dispatched at moment (A1), initiating an asynchronous transfer. At the moment (A2), when a `RESTORE` is triggered for the in-transit checkpoint, the transfer

has not yet completed. Execution must then pause until the data becomes available at (A3). For instance, with the Marmousi3D dataset and *Revolve*, synchronization accounts for 4.7% of total execution time with a cache size of 2, and 4.0% with a cache size of 3. For *zCut*, the impact is higher 7.8% and 7.6% for cache sizes 2 and 3, respectively. In contrast, *Uniform* incurs no synchronization overhead, as its design ensures sufficient slack to prefetch all required checkpointing data.

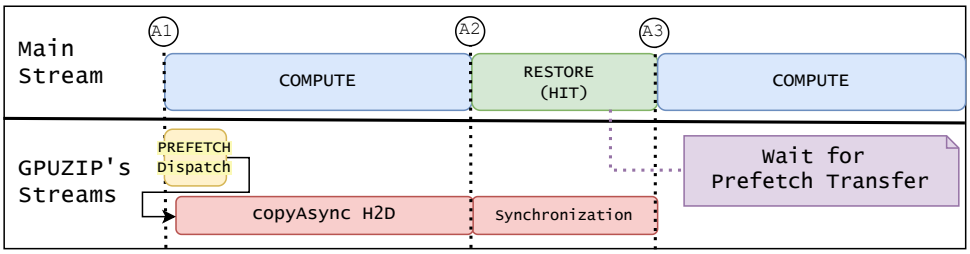


Figure 4.4: Representation of NSight trace for prefetching with Marmousi3D and *Revolve*.

A pertinent concern with the introduced *GPU Checkpoint Cache* is the required data in the GPU. Users must find a balance between their available GPU memory and the benefits that different cache sizes can bring to their setup. Table 4.1 provides the memory requirements for each *GPU Checkpoint Cache* for each GPU for various datasets. For example, the smaller fields have lower memory requirements, using about $3.1GB$ per GPU for a cache size of six positions with the Large dataset. In contrast, the M3D_Larger dataset requires $21GB$ of memory for a cache size of six, which may be unfeasible for some systems or applications.

CacheSize	Large(GB)	Marmousi3D(GB)	M3D_Larger(GB)	Salt(GB)
2	1.0	0.8	7.0	0.9
3	1.6	1.2	10.5	1.4
4	2.1	1.6	14.0	1.8
5	2.6	2.0	17.5	2.3
6	3.1	2.4	21.0	2.8

Table 4.1: GPU Memory required by the *GPU Checkpoint Cache* on each GPU.

4.3 Chapter Conclusion

This chapter discusses the prefetching mechanism that reduces checkpoint restoration latency in GPU seismic imaging by utilizing an associative GPU cache, concurrent CUDA streams, and a Prefetch Setup Algorithm to overlap data transfers and computation, thereby boosting throughput. Experiments demonstrated performance gains, particularly with temporally localized checkpointing, which narrowed the host-device memory speed gap and improved application efficiency. However, late prefetches can cause synchronization issues, and the cache memory requirements may be high. The next chapter (Chapter 5) will discuss compression before integrating it into the prefetching mechanism in Chapter 6.

Chapter 5

Applying Compression on Checkpoint Data

In Chapter 3 Section 3.1, data compression was presented as a fundamental technique used to reduce the size of digital information, with applications spanning from files and multimedia to large-scale scientific datasets often represented as high-dimensional floating-point tensors. Data compression is essential not only for saving storage space but also for improving data transfer efficiency within complex computing systems; by decreasing the volume of data moved across memory hierarchies and network channels, such as CPU–GPU transfers and multi-stage I/O pipelines, compression helps alleviate bandwidth bottlenecks and accelerates overall workflow performance, making it a critical tool for optimizing scientific applications constrained by data movement.

As discussed in Chapter 2 Section 2.4, the primary bottleneck in Awave-3D lies in the transfer of checkpointing data between the GPU and host memory. During a `SAVE` operation, the data must be copied from the GPU to the host before the computation can proceed, leading to idle GPU time. One strategy to mitigate this is to compress the data before transferring it, as GPUs are highly efficient at running compression algorithms [19]. Compression improves communication performance when the combined time for compressing, transferring, and decompressing data is less than the time required to transfer the data without compression. This condition is expressed in Equation 5.2, based on the total execution time with and without compression.

The total time when using compression is given by:

$$T_{\text{total}} = T_{\text{compress}} + T_{\text{transfer\_compressed}} + T_{\text{decompress}} \tag{5.1}$$

where:

- T_{compress} is the time to compress the data on the GPU,
- $T_{\text{transfer\_compressed}}$ is the time to transfer the compressed data over PCIe (*Device-to-Host* and *Host-to-Device*),
- $T_{\text{decompress}}$ is the time to decompress the data on the host or GPU.

Compression for communication is considered beneficial when the compression time is lower than the baseline (without compression):

$$T_{\text{total}} < T_{\text{baseline}} \tag{5.2}$$

Beyond cutting the data transfer time, data compression can reduce the size of the checkpointing data pool in the host memory. With smaller snapshots, there is more space in the host memory to make more snapshots, in the same direction as Kukreja *et al.* (2020) [37]. The *Uniform* algorithm, for example, can make more snapshots as long as more memory is available.

This chapter is organized as follows: first, Section 5.1 presents the methodology used to apply compression on the checkpoint data, and then Section 5.2 presents the results achieved by using compression.

5.1 Methodology

This work adopts lossy compression, a data compression method that reduces data size by permanently eliminating certain information, particularly in floating-point precision, in contrast to lossless compression, which preserves all the original data. Lossy compression results in less data, but the decompressed data will not be identical to the original data. However, the loss can be mitigated by a predefined error-control method, making the data loss insignificant, depending on the intended use of the data [18]. This technique has been successfully applied to RTM to improve compression ratios while maintaining high-quality seismic imaging [8, 18, 37], and it is increasingly adopted in other scientific domains as well [18, 41].

This section is organized as follows: Subsection 5.1.1 provides an overview of the core techniques behind lossy compression used in this work; next, Subsection 5.1.2 introduces the GPU-based lossy compressors integrated into GPUZIP and highlights their key differences; then, Subsection 5.1.3 presents the metrics used to evaluate these compressors and assess the impact of compression on data quality; and finally, Subsection 5.1.4 explains how the selected compressors were integrated into the Awave-3D through GPUZIP.

5.1.1 Lossy Compression

Compressors follow a structured compression pipeline incorporating multiple techniques, making each compressor unique. The methods used by compressors in this work (Bitcomp, cuZFP, and cuSZp) are: pointwise data prediction (PDP), quantization (QT), bit-plane coding (BPC), discrete cosine transform (DCT), and lossless encoding (LE) [18].

- **Pointwise Data Prediction (PDP)** is often the first or second step in modern compression algorithms. It utilizes a prediction function (e.g., Lorenzo Predictor, linear interpolation, and linear regression) that estimates the value of a data point based on its neighboring/adjacent values (spatial or temporal correlation), then computes the difference between the predicted and original values. This process generates close-to-zero values, which are easier to compress [18, 29, 72].

- **Quantization (QT)** further reduces the data's precision by mapping values to discrete levels, which may be based on either the original values or residuals from the prediction step (PDP) [18, 72].

- **Lossless encoding (LE)** usually occurs at the end of the compression process. Its primary purpose is to eliminate redundant values that arise from earlier steps, particularly after quantization (QT), since these values are often very sparse. Huffman Encoding is a well-known example of a lossless encoding algorithm [18].

- **Discrete Cosine Transform (DCT)** converts spatial data into frequency coefficients, which helps to decorrelate the data, making it more compressible. This transformation results in a few significant coefficients while many others become near zero, allowing these near-zero values to be efficiently encoded using fewer bits. This process enhances compressibility, leading to higher compression ratios and reduced storage requirements [18].

- **Bit-Plane Coding (BPC)** is often used after the transform step (e.g., DCT). It encodes data bit by bit, starting with the most significant bits to preserve critical information first. Higher bit planes store the sign, exponent, and the most significant fraction bits for floating-point data. Therefore, truncating lower bit planes reduces precision with minimal impact on the overall data value [18].

Beyond how compressors compress, there is also how they control errors during compression. The compressors used in this work are categorized based on whether they provide fixed-ratio or error-bounded compression.

Fixed-ratio compressors guarantee a specific compression ratio, making them suitable for scenarios with strict memory constraints. The trade-off is in quality, which is sacrificed to reach the desired compression ratio [18].

Error-bounded compressors prioritize data fidelity, ensuring that compression errors remain within a specified limit. These error bounds can be absolute, where the maximum allowable deviation is fixed, or relative, where the error scales proportionally to the data values [18].

5.1.2 GPU-based Lossy Compressors

There are multiple known GPU-based lossy compressors [18], and to choose the compressors to be used in this dissertation, their quality and speed in the related work were evaluated [8, 19, 37, 61, 62], and whether they provide APIs to be used by partners. Based on that, the following three libraries were initially chosen:

- **cuSZp** [29] is one of the latest GPU-based compressors in the SZ family. It utilizes Pointwise Data Prediction (PDP) using a lightweight Lorenzo Predictor, followed by Quantization (QT), which transforms floating-point values into integers using linear-scale quantization. Finally, it employs Lossless Encoding (LE) with a customized Huffman encoder. cuSZp controls its error using an absolute error bound parameter.

- **cuZFP** [39] is a compressor from ZFP family. In contrast to its CPU implementation, the GPU-based implementation only supports fixed-rate mode, which makes the compressed size predictable but does not guarantee quality. The cuZFP compression scheme begins by dividing the data into small fixed-size blocks. The floating-point values are then converted to a fixed-point representation before applying a transformation to decorrelate the data, similar to the DCT explained above, but with some modifications inherited from the ZFP compressor family. Finally, Bit-Plane Coding (BPC) encodes the coefficients. cuZFP expects the parameter `maxBits`[1] [2], which controls the number of bits allocated per value in fixed-rate mode.

- **NVIDIA's NVCOMP Bitcomp** [1], unlike the open-source compressors used in this study, NVCOMP Bitcomp does not have its source code publicly available. According to the available documentation[3], it is based on the quantization (QT) of floating-point values to integers. The compression relies on two main parameters: '`algorithm`', which can be set to either *sparse* or *default*. The *sparse* option performs better with sparse data that contains many zeros[4]. The second parameter, '`delta`', is used for error control, ensuring that errors remain within the specified threshold.

5.1.3 Evaluating Quality

Metrics such as **PSNR** (*Peak Signal-to-Noise Ratio*) and **SSIM** (*Structural Similarity Index Measure*) are commonly used to evaluate quantitative quality in studies involving compression [29, 72] and also in works addressing compression for seismic and other scientific fields [10, 37, 70].

PSNR measures the noise introduced by compression, expressed in decibels (dB). It compares the maximum possible signal power to the noise power on a logarithmic scale, reflecting how humans perceive signal intensity.

The formula for calculating PSNR is:

$$PSNR(f,g) = 10 \times \log_{10}\left(\frac{MAX^2}{MSE(f,g)}\right) \tag{5.3}$$

where f represents the baseline image, and g is the comparison image, MAX represents the maximum possible pixel value of the image and MSE is the Mean Square Error that is given by:

[1] https://zfp.readthedocs.io/en/release0.5.4/modes.html#c.zfp_stream.maxbits - Accessed May 26, 2025

[2] https://zfp.readthedocs.io/en/release0.5.4/modes.html#mode-fixed-rate - Accessed May 26, 2025

[3] https://docs.nvidia.com/cuda/nvcomp/native_api.html#_CPPv417bitcompCreatePlanP15bitcompHandle_t6size_t17bitcompDataType_t13bitcompMode_t18bitcompAlgorithm_t - Accessed May 26, 2025

[4] https://docs.nvidia.com/cuda/nvcomp/cpp_api.html#_CPPv4N6nvcomp23BitcompFormatSpecHeader4algoE - Accessed May 26, 2025

$$\mathrm{MSE}(f,g) = \frac{1}{M \times N} \sum_{i=1}^{M} \sum_{j=1}^{N} (f_{ij} - g_{ij})^2 \tag{5.4}$$

where $M \times N$ are the image dimentions [28].

Although PSNR effectively quantifies distortion, it is less suited for capturing structural differences between images [28]. To address this limitation, **SSIM** assesses perceptual similarity, producing values between 0 and 1, where values closer to 1 indicate higher fidelity and structural resemblance. SSIM models image distortion as a combination of luminance, contrast, and structure:

$$\mathrm{SSIM}(f,g) = l(f,g) \cdot c(f,g) \cdot s(f,g) \tag{5.5}$$

where f represents the baseline image, and g represents the comparison image [28]. The components are defined as:

- $l(f,g)$ is the luminance comparison function, which evaluates how similar the two images are;
- $c(f,g)$ is the contrast comparison function;
- $s(f,g)$ is the structure comparison function, which assesses the correlation coefficient between f and g.

5.1.4 Integrating Compression in Awave-3D

Figure 5.1 illustrates the checkpointing workflow in Awave-3D with compression. When a `SAVE` request is made ①, the checkpoint data is compressed and stored in a temporary buffer in the GPU. The compressed data is immediately transferred via PCIe to host memory. When a `RESTORE` request occurs ②, the compressed checkpoint is retrieved from the host memory, transferred to the GPU buffer, and decompressed before rematerialization.

The intermediary GPU buffer is allocated at the start of execution based on the expected compression ratio. For fixed-rate compressors, this is straightforward: allocate the checkpoint size divided by the known compression ratio. However, for error-bound compression, the required buffer size is unknown beforehand. Users can estimate it through experimentation on small executions. For example, using Bitcomp '`delta=1e-8`' on Marmousi3D, the worst-case compression ratio is 2.6×, defining the buffer size required before running all shots.

The main limitation of using a one-position buffer capable of holding only one compressed checkpoint is its restriction on data access concurrency. For example, if a `SAVE` operation is followed closely by a `RESTORE`, the `RESTORE` must wait for the `SAVE`'s data transfer to complete before it can proceed. Additionally, all `RESTORE` operations are executed synchronously, further reducing potential overlap and performance gains.

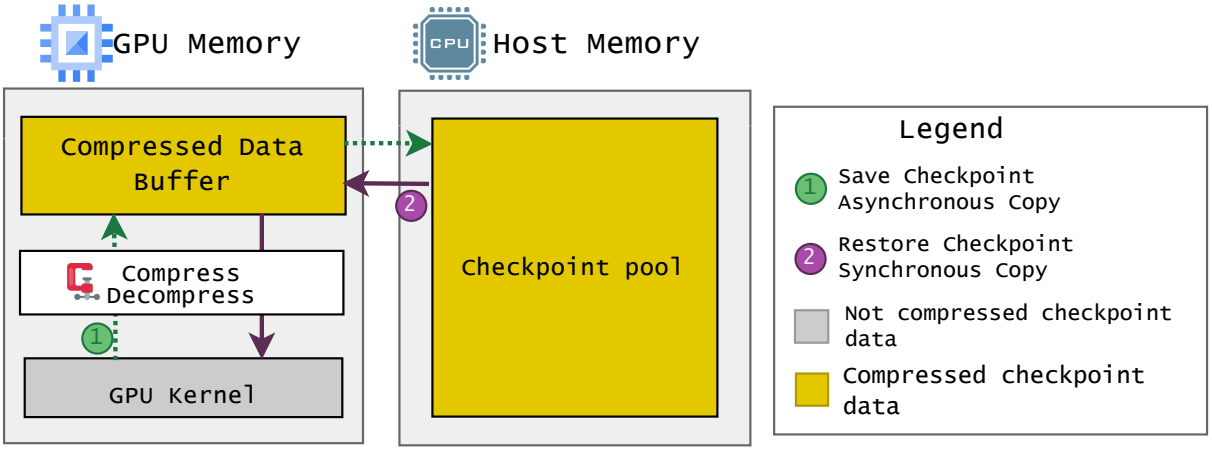


Figure 5.1: GPUZIP's memory architecture with compression enabled. For every `SAVE`, the data is compressed and then saved in a temporary buffer that will be transferred to the host memory. For every `RESTORE`, the data will be transferred from the host memory to the temporary GPU buffer and then decompressed and rematerialized.

5.2 Experimental Results

The effectiveness of compression in Awave-3D is given by balancing two essential aspects: reducing data movement overhead (speedup) and preserving the quality of the final migrated images — while higher compression ratios minimize the amount of data transferred through the PCIe interface, they may also introduce quality degradation.

This section is divided into four subsections, each covering a key aspect of the experimental evaluation. Section 5.2.1 describes the experimental setup used to assess the impact of compression. Section 5.2.2 presents warm-up experiments with fewer shots to identify optimal compression parameters, balancing compression ratio and output quality. The performance benefits of compression are detailed in Section 5.2.3, highlighting the speedup gains observed when applying compression to Awave-3D. Finally, Section 5.2.4 evaluates the quality of the full migrated field produced using the compressed data, analyzing both visual outcomes and quantitative error metrics.

5.2.1 Experimental Setup

Compression-only experiments were performed across three checkpointing algorithms (*Revolve*, *zCut*, and *Uniform*), using the Large, Marmousi3D, and Salt datasets. The same baselines, hardware, compiler versions, and NVIDIA drivers from the prefetching experiments were used (see Chapter 4 Section 4.2 for further details).

The optimal compression parameters for the speedup and quality experiments were determined through initial exploratory tests (described in Section 5.2.2). These configurations, applied in the performance analysis in Section 5.2.3 and the quality assessment in Section 5.2.4: cuZFP set with $maxBits = 8$ and Bitcomp with $delta = 1 \times 10^{-8}$ and default algorithm.

Although this dissertation primarily addresses communication bottlenecks rather than minimizing recomputation, the number of snapshots used during checkpointing can sig-

nificantly influence performance. That said, this study does not explore the optimal point at which increasing the number of snapshots yields diminishing returns. *Uniform* is the only checkpointing algorithm that attempts to allocate the maximum number of snapshots allowed by the available memory, calculated based on the worst-case compression ratio. In contrast, both *Revolve* and *zCut* use a fixed number of snapshots identical to their respective baselines.

The number of snapshots used in the *Uniform* setup for each dataset and compression configuration is as follows in Table 5.1.

Dataset	No Compression	Bitcomp	cuZFP
Large	149	599	599
Marmousi3D	177	519	674
Salt	149	374	535

Table 5.1: Number of snapshots used by *Uniform* under different compression configs.

5.2.2 Tuning Compression Parameters

The following experiments were called "warmup experiments" because they were conducted with only two shots for each dataset, aiming to calibrate the parameters and determine which compressors would yield the best results for Awave-3D. The experiments used the Bitcomp, cuSZp, and cuZFP compressors on the Large, Marmousi3D, and Salt datasets. For Bitcomp, the default algorithm was employed, combining three "`delta`" configurations: 1×10^{-2}, 1×10^{-4}, and 1×10^{-8}. cuSZp was configured to use relative errors with 1×10^{-2}, 1×10^{-4}, and 1×10^{-8} error bounds. In testing cuZFP, '`maxBits`' configurations of 2, 4, and 8 were assessed.

Table 5.2 summarizes the results by reporting the average Compression Ratio (CR), PSNR, and SSIM as quality indicators. To reflect the primary bottleneck of Awave-3D, the table presents the total time spent on data transfers, which is the sum of time used by all four GPUs during execution. Additionally, the table shows the total time spent on compression and decompression, evaluating the overhead introduced by each compressor.

Regarding quality, cuSZp performed poorly. It achieved the worst PSNR and SSIM metrics and had the highest compression and decompression overhead. Its compression ratio ranges from 2.1× to 2.38× for an error bound of 1×10^{-8}.

Bitcomp, on the other hand, performed better for the proposed datasets: considering the '`delta`' parameter of 1×10^{-8}, it had the highest compression ratios ($115 - 300\times$), the best quality metrics (SSIM very close to 1.00), and the fastest (de)compression times.

cuZFP also delivered satisfying quality results, with SSIM close to Bitcomp but more noise (PSNR) than Bitcomp. As a fixed-rate compressor, cuZFP is limited to its predefined ratio of 4× with FP32 on $maxBits = 8$.

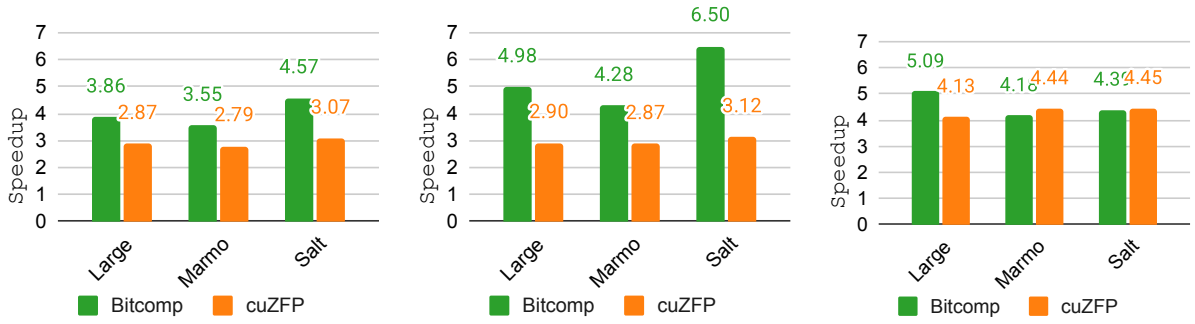
All compressors can significantly reduce the time spent on (de)compression, including the time involved in communication between the host and the GPU. For instance, when considering the Large dataset in the best-quality scenario for each compressor, the following times were recorded: 2 minutes and 08 seconds for Bitcomp, 06 minutes and 55

seconds for cuSZp, and 04 minutes and 51 seconds for cuZFP. In contrast, the baseline took 37 minutes and 52 seconds.

The collected data indicates that introducing compression can alleviate application bottlenecks. On the other hand, quality fidelity is important for this kind of application, so only scenarios that presented SSIM very close to 1 were considered for the all-shots experiments with Awave3D: Bitcomp $delta = 1 \times 10^{-8}$ and cuZFP with $maxBits = 8$.

5.2.3 Overall Speedup

Applying compression in Awave-3D provided substantial speedups across all checkpointing libraries, datasets, and compressors evaluated as displayed in Figure 5.2.



(a) Revolve (b) zCut (c) Uniform

Figure 5.2: Overall speedup achieved by applying Bitcomp and cuZFP compressors across three datasets (Large, Marmousi3D, and Salt). Each subfigure presents results for a specific checkpointing algorithm. Baselines consider no compression, saving checkpoint data directly to the host memory, and restoring from the host memory to be rematerialized in the GPU memory.

Starting with *Revolve* (Figure 5.2a), it is possible to observe that Bitcomp consistently achieves higher speedups, ranging from 3.55× to 4.57× across datasets. cuZFP also delivers noticeable improvements, with speedups between 2.79× and 3.07×. The source of these gains comes from the reduction in communication overhead. This behavior is clearly illustrated in Figure 5.3a, where the communication time (orange bars, representing device-to-host and host-to-device transfers) significantly decreases when compression is applied. In contrast, the overhead of compression and decompression operations is almost negligible compared to the overall execution time.

Moving to *zCut* (Figure 5.2b), the benefits of using compression become even more pronounced. Bitcomp achieves speedups as high as 6.5× for the Salt dataset, while cuZFP maintains improvements between 2.87× and 3.12×. Again, the main driver of these gains is the reduction in communication cost, as shown in Figure 5.3b, following the same pattern observed in Revolve.

Uniform can dynamically adapt to the available memory budget and allocate as many snapshots as possible. Thanks to cuZFP's predictable and stable compression ratio of 4× (considering 32-bit floating-point data and `maxBits` = 8) *Uniform* can consistently store more snapshots, minimizing redundant computations. In contrast, Bitcomp's compression

Profile	CR (AVG)	PSNR	SSIM	Time Spent d2h & h2d	Time spent (de)comp.
Large					
baseline	-	-	-	0:37:52	-
bitcomp, delta=1e-2	678.73	44.48	0.2481	0:00:00	0:00:10
bitcomp, delta=1e-4	228.99	67.42	0.7952	0:00:07	0:00:29
bitcomp, delta=1e-8	137.77	140.51	1.0000	0:02:08	0:00:40
cuSZp,errBnd=1e-2	10.23	37.73	0.0000	0:01:53	0:02:56
cuSZp,errBnd=1e-4	4.66	45.61	0.0000	0:03:29	0:03:46
cuSZp,errBnd=1e-8	2.27	45.62	0.0000	0:06:55	0:05:43
cuZFP, bitRate=2	15.66	66.82	0.7887	0:01:11	0:01:18
cuZFP, bitRate=4	7.83	84.51	0.9576	0:02:24	0:01:32
cuZFP, bitRate=8	3.91	111.31	0.9995	0:04:51	0:01:44
Marmousi3D					
baseline	-	-	-	1:16:10	-
bitcomp, delta=1e-2	597.03	41.70	0.0057	0:00:04	0:01:00
bitcomp, delta=1e-4	205.95	65.40	0.4589	0:00:31	0:01:04
bitcomp, delta=1e-8	115.96	95.00	1.0000	0:06:49	0:01:27
cuSZp,errBnd=1e-2	13.71	41.42	0.0000	0:03:53	0:06:14
cuSZp,errBnd=1e-4	5.42	41.51	0.0001	0:06:41	0:08:03
cuSZp,errBnd=1e-8	2.38	41.75	0.0001	0:15:22	0:12:51
cuZFP, bitRate=2	15.71	48.03	0.7190	0:02:45	0:02:51
cuZFP, bitRate=4	7.85	68.72	0.9834	0:05:33	0:03:32
cuZFP, bitRate=8	3.93	98.02	0.9996	0:11:16	0:04:01
Salt					
baseline	-	-	-	0:54:10	-
bitcomp, delta=1e-2	670.02	43.57	0.7910	0:00:02	0:00:41
bitcomp, delta=1e-4	371.55	49.41	0.8798	0:00:15	0:00:44
bitcomp, delta=1e-8	309.45	127.36	1.0000	0:01:37	0:00:50
cuSZp,errBnd=1e-2	14.86	64.61	0.0640	0:02:32	0:04:00
cuSZp,errBnd=1e-4	4.63	45.51	0.0641	0:05:28	0:05:32
cuSZp,errBnd=1e-8	2.12	46.87	0.0640	0:12:52	0:08:53
cuZFP, bitRate=2	15.77	43.76	0.8464	0:01:45	0:01:26
cuZFP, bitRate=4	7.89	62.52	0.9182	0:03:32	0:01:36
cuZFP, bitRate=8	3.94	87.18	0.9919	0:07:10	0:01:45

Table 5.2: Compression warm-up experiment results using Awave-3D on two shots. The columns are defined as follows: **CR (AVG)**: Average Compression Ratio (higher is better); **PSNR**: higher is better; **SSIM**: values closer to 1 indicate more similarity to the baseline; **Time Spent d2h & h2d**: Total time (h:m:s) for four GPUs during *Device-to-Host* (d2h) and *Host-to-Device* (h2d) communication; **Comp. & Decomp.**: Total time (h:m:s) for compression and decompression across the same four GPUs.

ratio varies between 2.0× and 2.6×, based on the worst-case values observed during the warmup experiments (as seen in Table 5.2). So cuZFP slightly outperforms Bitcomp for the Marmousi3D and Salt dataset for *Uniform* cases, since with cuZFP *Uniform* could take more snapshots and avoid more recomputation.

An exception appears in the Large dataset. Due to its sparse nature, its worst-case compression ratio with Bitcomp is higher than other datasets and achieves the same number of snapshots as cuZFP. In this scenario, Bitcomp provides a greater overall speedup because it can reduce communication more effectively than cuZFP, as observed in *Revolve* and *zCut* experimentation.

It is important to highlight that the speedup observed for *Uniform* is primarily influenced by its ability to increase the number of snapshots due to compression. For instance, in the baseline execution of the Marmousi3D dataset, only 177 snapshots were used. When using Bitcomp, this number increased to 519, and with cuZFP, to 674. If the compressed versions had been restricted to the same 177 snapshots as the baseline, the speedups would have dropped significantly, from 4.18× to 1.88× for Bitcomp, and from 4.44× to 1.79× for cuZFP. This underscores that beyond compression reduces data transfer overhead, it also enables the storage of more snapshots, thereby decreasing recomputation costs; an advantage also highlighted in previous studies such as [37].

The *Uniform* behavior described is further illustrated in Figure 5.3c, showing that compression in *Uniform* helped reduce computation and communication time. Unlike other checkpointing schemes, *Uniform* performs its `SAVE` operations only at the beginning of the execution, and does not take more intermediate data after the backward phase starts.

A common trend observed in all checkpoint algorithms executions in Figure 5.3: decompression times are longer than compression times. This discrepancy is not due to decompression being inherently slower; rather, it is because `RESTORE` operations, which necessitate decompression, occur much more often than `SAVE` operations, which involve compression. In general, decompression operations are faster than compression.

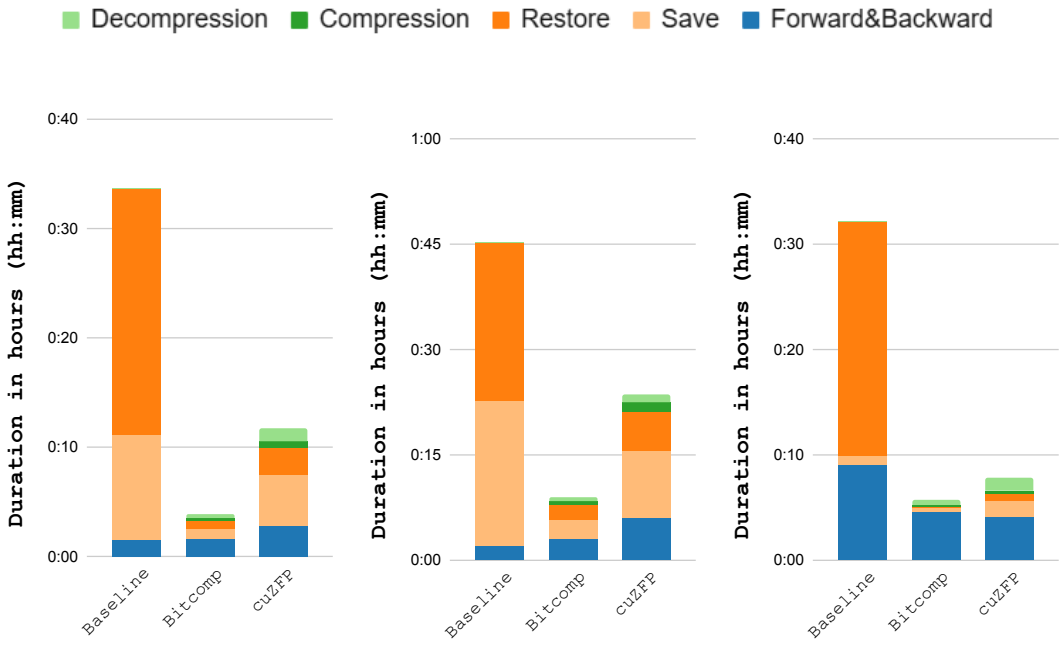
Overall, these results demonstrate the effectiveness of compressors in reducing communication bottlenecks and accelerating Awave-3D while introducing minimal overhead from compression and decompression tasks, satisfying in all scenarios the Equation 5.2.

5.2.4 Quality Assessment

A qualitative assessment was performed to evaluate whether compression would visually impact the final migrated images generated by Awave-3D.

Identifying perceptible differences between the migrated images using compression and the baseline without compression is challenging after visually inspecting the resulting images. In Figure 5.4a, a frame from the Salt dataset is shown without compression, along with the compressed versions, Bitcomp (Figure 5.4d) and cuZFP (Figure 5.4c). Figure 5.4b presents an absolute error map that reveals some minor discrepancies in shades of red. Additionally, it is essential to note that the highest error in this dataset is 0.08 within the -750 to 750 interval, indicating that the images produced by the baseline and GPUZIP appear identical to the naked eye. The overall 3D field and other datasets demonstrate similar results, which can be examined and viewed in the reproducibility data available in [47].

For the quantitative analysis, Table 5.3 summarizes the results of the PSNR (Peak Signal-to-Noise Ratio) and SSIM (Structural Similarity Index) metrics across the three



(a) Revolve (b) zCut (c) Uniform

Figure 5.3: Execution time breakdown for Salt dataset.

datasets used in this study. Regarding PSNR, Bitcomp consistently achieves higher values than cuZFP across all datasets. However, the SSIM values for both compressors are very close to 1.00 across all datasets, suggesting that, from a structural similarity perspective, the compressed images retain excellent fidelity to the original images.

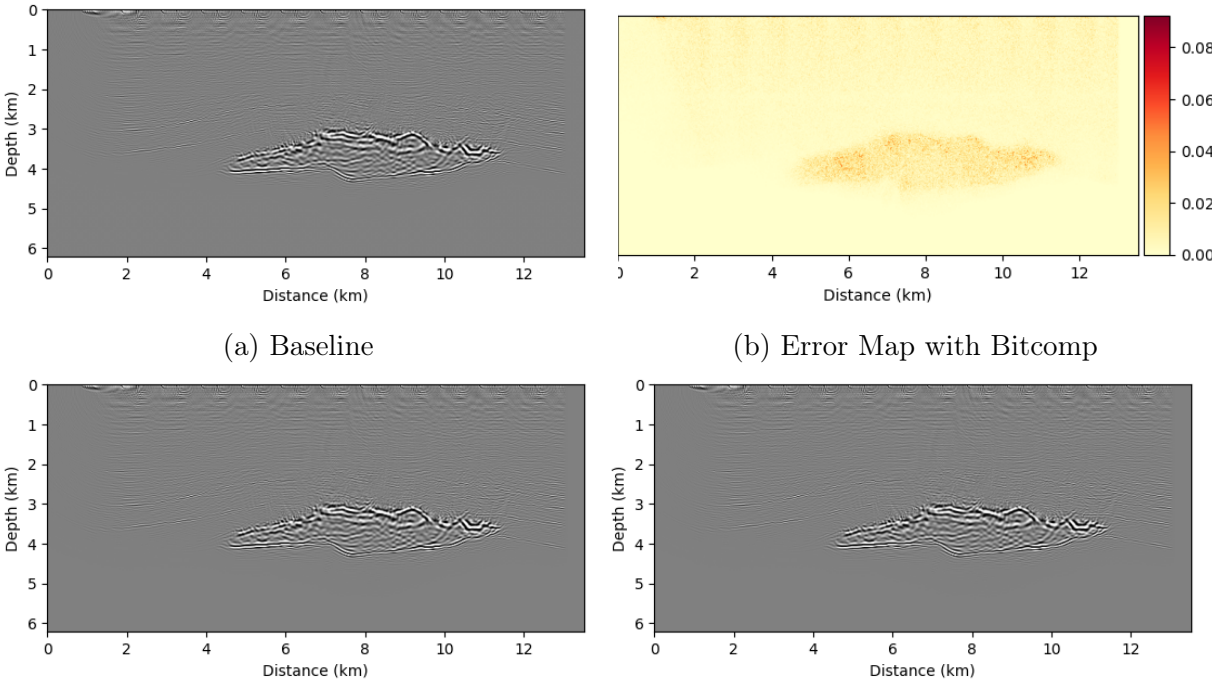
The quantitative result aligns with the qualitative analysis and supports the conclusion that GPUZIP compression preserves the integrity of the final seismic images, even with aggressive data reduction strategies.

Dataset	Compressor	PSNR	SSIM
Large	cuZFP	105.98	0.99
	Bitcomp	138.06	0.99
Marmousi3D	cuZFP	94.23	0.99
	Bitcomp	120.94	0.99
Salt	cuZFP	78.01	0.99
	Bitcomp	135.82	0.99

Table 5.3: Comparison of PSNR and SSIM for cuZFP and Bitcomp across all datasets. Higher PSNR and SSIM values indicate better quality and greater similarity to the baseline image.

5.3 Chapter Conclusion

This chapter examined compression for checkpoint data in Awave-3D, addressing the communication bottlenecks of transferring large data between GPU and host memory. By



(a) Baseline

(b) Error Map with Bitcomp

(c) cuZFP

(d) Bitcomp

Figure 5.4: Comparison of migrated images for Salt dataset

compressing checkpoint data, the system reduces transfer time, improves computational efficiency, and minimizes GPU idle time. Compression is beneficial when the total time for compression, transfer, and decompression is less than that for uncompressed transfer, as shown in Equation 5.2.

The methodology used lossy compression with two GPU-based compressors: Bitcomp and cuZFP. Bitcomp provides error-bounded compression with strong fidelity guarantees, while cuZFP offers fixed-ratio compression for predictable memory usage.

Experiments showed speedups from reduced communication overhead and increased snapshot capacity in *Uniform* checkpointing. Quality assessments confirmed high data fidelity, with SSIM near 1.0 and PSNR above acceptable thresholds.

In conclusion, effective compression reduces communication overhead, accelerates execution, and improves memory efficiency in GPU-accelerated applications. The next chapter will combine prefetching techniques from the previous chapter with the compression findings presented in this chapter.

Chapter 6

Combining Compression and Prefetching

The checkpoint prefetching mechanism benefits Awave-3D, with speedups up to 3.4× with *Revolve*, 1.7× with *zCut*, and 2.3× with *Uniform*. These gains are primarily due to reducing blocking time on the PCIe bus by retrieving checkpoint data in advance and reusing data already stored in the cache.

Applying checkpointing data compression is also worthwhile to Awave-3D. Depending on the dataset, it can increase speedups to 4.57× with *Revolve*, 6.5× with *zCut*, and 5.09× with *Uniform*. These improvements stem from reducing the amount of data transferred over PCIe, thus alleviating one of Awave-3D's main performance bottlenecks.

Both techniques can be combined: compression can help prefetch be faster if the checkpoint data to be prefetched were smaller, so it can be transferred faster, meeting with the prefetching *timeliness*.

This chapter is organized as follows: Section 6.1 presents the decisions made regarding combining both techniques, and Section 6.2 presents the experimental results of the methodology adopted.

6.1 Methodology

The main hurdle for prefetching is that the prefetched checkpoint does not have enough time to come from the host memory to the GPU memory before the checkpoint algorithm performs a `RESTORE` action; this makes the GPU idle while the checkpoint data synchronization happens.

The timeline Ⓐ in Figure 6.1 illustrates the concurrency scenario: the *Prefetch Dispatch* (A1) asynchronously transfers a checkpoint of $400MB$ from the host to the GPU. At the same time, the computation of a kernel ("`COMPUTE`" that is a sequence of `FORWARD` and `BACKWARD`) is happening. However, when the `RESTORE` in (A2) occurs, the checkpoint requested is still being transferred. Hence, the GPU is idle waiting for the synchronization to finish, and then "`COMPUTE`" can forward in (A3).

In contrast, Timeline Ⓑ illustrates how compression effectively addresses the bottleneck observed in Ⓐ. During the *Prefetch Dispatch* phase (B1), a 4× compressed checkpoint

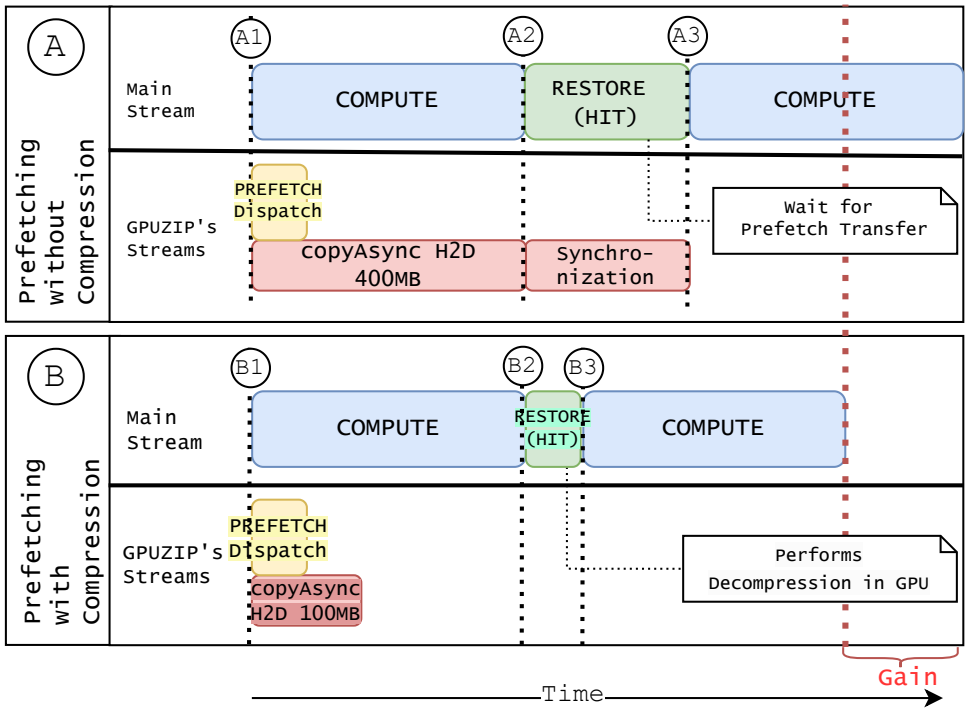


Figure 6.1: Timelines showing prefetching traces: Timeline "A" depicts no compression applied to checkpoint data, while Timeline "B" illustrates the effects of compression on prefetching data. The dotted red line highlights the efficiency gain from compression.

is transferred, significantly reducing *Host-to-Device* data movement. Consequently, when the `RESTORE` operation (B2) is triggered, the checkpoint data is already present in the GPU memory, allowing for immediate decompression and seamless resumption of execution (B3). The interval marked as "`Gain`" reflects the resulting performance benefit from this optimization.

Algorithm 4 outlines the execution loop for integrating prefetching and compression in Awave-3D. Initially, the *Prefetch Setup* occurs with the configured cache size (line 1). At this stage, the user can decide to increase the cache size, as compressed data will be smaller than uncompressed data, e.g., if it is known that the compressed data is 2× smaller, the cache size can be doubled compared to a situation without compression and then store more snapshots in the *GPU Checkpoint Cache*. Another scenario where compression can be beneficial is when dealing with a large data field that could not accommodate a minimum cache size of two positions; with compression, the smaller data could now fit into the available space.

Still following Algorithm 4, line 3 will determine the following checkpointing action. If prefetching conditions are met, a dispatch is triggered to load future checkpointing data in advance (lines 4–6). When a `SAVE` action occurs, the computing data is compressed directly into the *GPU Checkpoint Cache*. Compressors will perform a *Device-to-Device* (D2D) copy behind the scenes, and then the *GPU Checkpoint Cache* will transfer the compressed data to the host memory (D2H) (lines 7–9).

If a `RESTORE` action is issued, the system checks whether the required timestep is already in the cache (line 11). If it is and the transfer is still ongoing, it waits for

completion (lines 12–14), then it will decompress the data directly to the computation data (D2D) (line 15). Otherwise, the worst scenario, the cache miss, will happen: it transfers the compressed checkpoint from host to cache (H2D) and decompresses it for computation (D2D) (lines 16–19).

The latest version of GPUZIP is the result of combining compression and prefetching techniques. Figure 6.2 presents the GPUZIP architecture, which extends the prefetching architecture, preserving the *GPU Checkpoint Cache* and *Checkpoint Pool* structure along with its stream and policy mechanisms.

On `SAVE` operations, the checkpoint data is compressed and compressor dumps data directly to the *GPU Checkpoint Cache* ①, then GPUZIP explicitly copies the compressed data from the *GPU Checkpoint Cache* to the *Checkpoint Pool* (*Device-to-Host*). On `RESTORE` ② or prefetch ③, the data moves in the opposite direction, the compressed data is copied from the *Checkpoint Pool* to the *GPU Checkpoint Cache* (*Host-to-Device*), and before re-materialization, the data is decompressed. Compressor dumps the data directly to the Awave-3D reserved memory to go forward with the computation.

Algorithm 4 Checkpointing Prefetching with Compression Execution Loop

```
1:  pav ← PrefetchSetup(config.cacheSize)
2:  while action ≠ TERMINATE do
3:      action ← Checkpointing.GetAction()
4:      if pav.shouldDispatch() then
5:          pav.dispatch()
6:      end if
7:      if action == SAVE then
8:          Compress computing data directly to the cache (D2D)
9:          Save from cache to host (D2H)
10:     else if action == RETRIEVE then
11:         if currentTimestep is in cache then
12:             if transfer is still happening then
13:                 Wait for the transfer to finish
14:             end if
15:             Decompress from Cache directly to computing data (D2D) {Cache Hit!}
16:         else
17:             Copy from host to cache (H2D)
18:             Decompress from Cache directly to computing data (D2D)
19:         end if
20:     else if action == FORWARD or action == BACKWARD then
21:         Perform forward or backward computation
22:     end if
23: end while=0
```

6.2 Experimental Results

This section presents the experimental evaluation of combining GPUZIP's prefetching mechanism with data compression. The goal is not only to assess the effectiveness of this

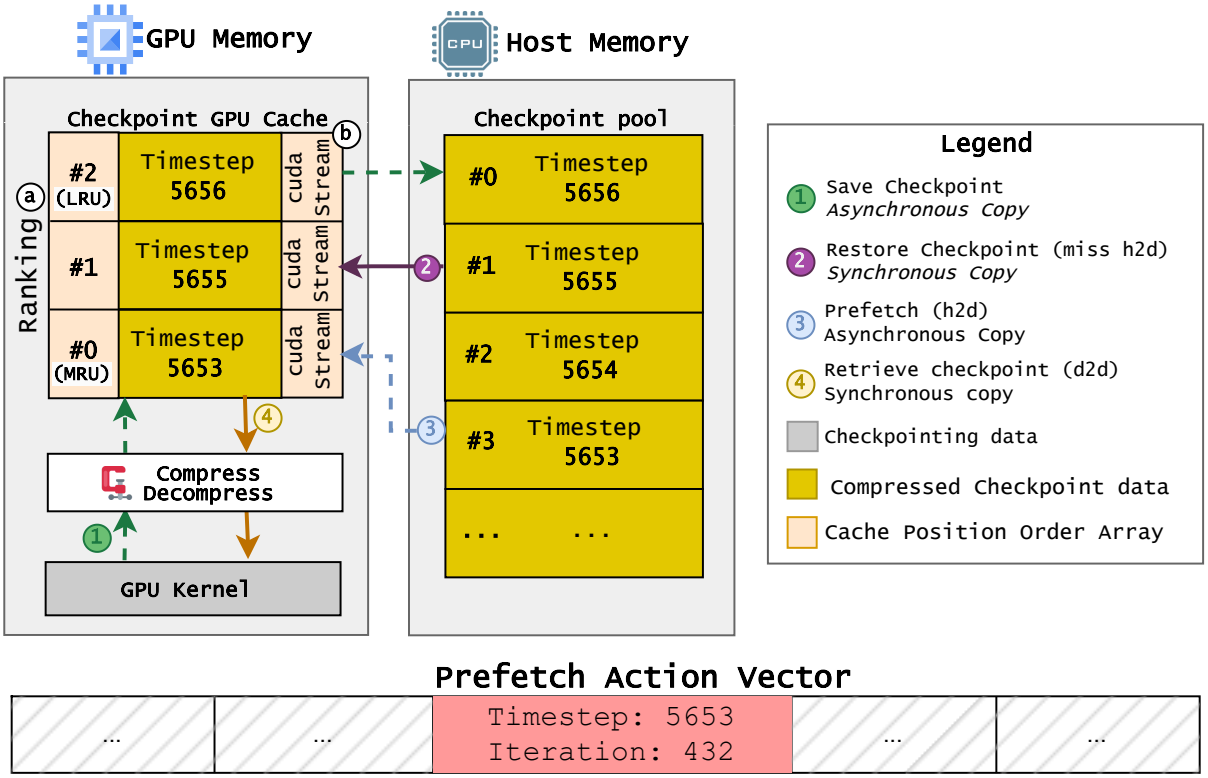


Figure 6.2: GPUZIP's memory architecture

integration but also to compare it with previously evaluated techniques – prefetching-only (Chapter 4) and compression-only (Chapter 5) – to identify the most effective configurations for each checkpointing algorithm and dataset.

This section is organized as follows:

- Section 6.2.1 details the experimental setup and profiles evaluated.
- Section 6.2.2 presents the speedups achieved by combining prefetching and compression.
- Section 6.2.3 analyzes memory usage under different configurations.
- Section 6.2.4 evaluates scalability across multiple nodes.

6.2.1 Experimental Setup

The experiments use the same three checkpointing algorithms—*Revolve*, *zCut*, and *Uniform* —and the same datasets: Large, Marmousi3D, and Salt. Hardware and software environments are identical to those described in Chapter 4 Section 4.2, and the compression parameters follow those used in Chapter 5 Section 5.2.1.

Each checkpointing algorithm is evaluated using its standard implementation in Awave-3D as the baseline. In this default setup, all checkpoint data is stored in host memory and synchronously transferred to the GPU during restore operations. These baselines do not use caching, compression, or prefetching.

The compression configurations are:

- **Bitcomp**: `delta` = 1×10^{-8}, `algorithm` = default
- **cuZFP**: `maxBits` = 8

The cache sizes evaluated range from 2 to 6, consistent with the configurations from Chapter 4. Cache sizes that produced identical results are not shown to maintain clarity and avoid redundancy in the result tables. Complete experimental data, including all cache sizes, is available in the reproducibility repository [47].

The evaluated profiles are:

- **Prefetching** – GPUZIP prefetching without compression.
- **cuZFP** – Compression-only using cuZFP.
- **Bitcomp** – Compression-only using NVIDIA's Bitcomp.
- **Prefetching + cuZFP** – Combination of prefetching and cuZFP.
- **Prefetching + Bitcomp** – Combination of prefetching and Bitcomp.

6.2.2 Overall Speedup

To evaluate the combined impact of prefetching and compression, Tables 6.1, 6.2 & 6.3 present the speedups achieved using each technique individually and then combined, with both compressors (cuZFP and Bitcomp).

Starting with *Revolve* (Table 6.1), prefetching increases speedups as the cache grows. Compression alone yields even greater gains, though this depends on how compressible the data is. The best performance results from combining both techniques. With cuZFP, speedups range from 4.1× to 4.6× at cache size 4; cache sizes 5 and 6 are omitted due to saturation. Bitcomp reaches its saturation point at cache size 4, since the smaller data size (due to compression) allows prefetching to occur in time before `RESTORE`.

As shown in Figure 6.3a, this combined approach nearly eliminates the blocking time (orange bars) caused by *Device-to-Host* and *Host-to-Device* transfers. The GPUZIP overhead (green bars) – which includes compression, decompression, and internal method calls like *Prefetch Setup Algorithm*– remains minimal. Prefetching-only configurations still show the `SAVE` operation as a bottleneck due to synchronization delays when a new snapshot must wait for the previous one to finish transferring. Compression alleviates this by making *Device-to-Host* faster, reducing `SAVE` blocking almost entirely.

Speedups for *zCut* (Table 6.2) are less sensitive to cache size, regardless of whether compression is applied. Compression alone already brings notable improvements, but combining it with prefetching further enhances performance, up to 8.8× with Bitcomp at cache size 2. The limited reactivity to cache size stems from *zCut*'s tendency to save many intermediate snapshots, quickly saturating the cache. Here, compression is crucial to reduce `SAVE` overhead. Still, the prefetching mechanism helps by preparing data for `RESTORE` ahead of time. Figure 6.3b confirms these observations: blocking times drop, and GPUZIP overhead remains negligible.

For *Uniform* (Table 6.3), compression has a strong impact by allowing more checkpointing data to be stored. Prefetching, on the other hand, significantly reduces *Host-to-Device* delays. Combining both techniques delivers the best results: Bitcomp performs better on the Large dataset, while cuZFP leads on Marmousi and Salt. This difference is attributed to cuZFP's higher fixed compression ratio (4×), which enables more snapshots than Bitcomp (2.6× worst case). Thus, *Uniform* benefits from three factors: (a) increased snapshots, (b) smaller transfer volume, and (c) hidden transfer latency through prefetching.

As shown in Figure 6.3c, restore operations are the main bottleneck in *Uniform*'s baseline. Combining compression and prefetching reduces this time to nearly zero. Worth noting that the remaining *Device-to-Device* copies presented in prefetching-only experimentation are hidden with compression because the compressor dumps the snapshot directly to the cache, so now, that *Device-to-Device* copy is hidden in the green bars.

An important point arises from *Uniform*: if restoration points are already far apart, why does not prefetching alone yield the same gains as the combination? The key is the checkpoint quantity. Compression enables storing more snapshots, reducing recomputation, and speeding up the overall execution. Without increasing the number of snapshots, compression brings no advantage over prefetching alone; speedups drop to 1.84× (Bitcomp) and 1.81× (cuZFP) on Marmousi3D. In short, compression helps not through faster transfers (as in other checkpointing algorithms) but by enabling more frequent snapshots.

In summary, across all configurations, Figure 6.3 shows that GPUZIP's internal overhead (green bars) is consistently low. This small cost is far outweighed by the reduction in communication time (orange bars), which remains the primary bottleneck addressed by GPUZIP.

	Prefetch Only					**Compression Only**		**Bitcomp**				**cuZFP**		
	Cache Size					**Comp. Type**		**Cache Size**				**Cache Size**		
	2	**3**	**4**	**5**	**6**	**Bit-comp**	**cu-ZFP**	**2**	**3**	**4**	**5**	**2**	**3**	**4**
L	2.5	2.9	3.2	3.3	3.4	3.8	2.8	4.5	4.6	4.6	4.6	3.9	4.0	4.1
M	2.3	2.7	3.0	3.2	3.2	3.5	2.7	4.5	4.6	4.7	4.7	3.9	4.1	4.2
S	2.4	2.8	3.2	3.3	3.4	4.5	3.0	5.0	5.1	5.1	5.1	4.2	4.4	4.6

Table 6.1: Overall speedup for *Revolve* (checkpoint prefetching + compression). Datasets: L = Large, M = Marmousi3D, S = Salt.

6.2.3 Memory Consumption

As discussed in this chapter, compression reduces data transfer time and significantly lowers memory consumption in the *GPU Checkpoint Cache*. The required memory footprint is reduced since checkpoint data is compressed on the GPU before being stored in the cache. More minor memory requirements are advantageous for large datasets or systems

	Prefetch Only Cache Size = 2	Compression Only		Bitcomp Cache Size = 2	cuZFP Cache Size =2
		Bitcomp	cuZFP		
L	1.6	4.9	2.9	7.8	5.4
M	1.6	4.2	2.8	6.9	5.5
S	1.6	6.5	3.1	8.8	5.5

Table 6.2: Overall speedup for *zCut* (checkpoint prefetching + compression). Datasets: L = Large, M = Marmousi3D, S = Salt.

	Prefetch Only	Compression Only		Prefetch + Bitcomp	Prefetch + cuZFP
Ds	Cache Size = 2	bitcomp	cuZFP	Cache Size = 2	Cache Size = 2
L	2.3	5.09	4.13	5.8	5.1
M	1.8	4.18	4.44	4.4	4.8
S	2.1	4.39	4.45	4.5	5.1

Table 6.3: Overall speedup for *Uniform* (checkpoint prefetching + compression). Datasets: L = Large, M = Marmousi3D, S = Salt.

(or programs) with limited available GPU memory, where fitting the required number of snapshots would otherwise be infeasible.

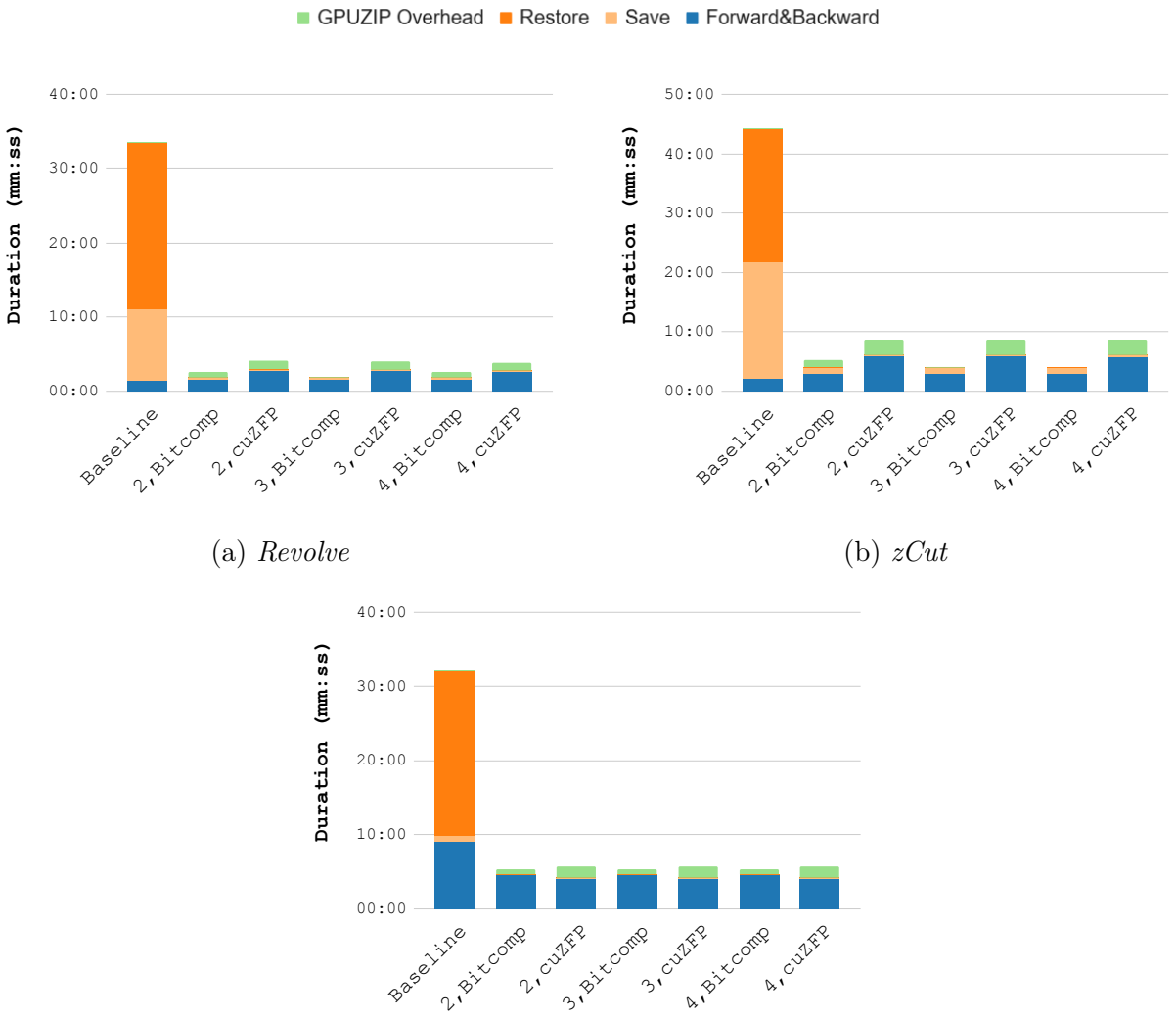
Table 6.4 presents the GPU memory required by the *GPU Checkpoint Cache* per GPU under different configurations. The table compares memory usage for each dataset when using: (a) no compression, (b) compression with Bitcomp, and (c) compression with cuZFP. For Bitcomp, the memory allocation is based on the worst-case compression ratio observed during the warm-up phase. At the same time, cuZFP is configured to maintain a fixed compression ratio of 4× (using `maxBits = 8`).

The results show that both compressors yield significant memory savings for all cache sizes and datasets compared to the uncompressed baseline. For instance, in the M3D_Larger case with cache size 6, memory consumption drops from $\sim 21.0GB$ (uncompressed) to $\sim 8.1GB$ with Bitcomp and $\sim 5.3GB$ with cuZFP.

6.2.4 Scalability

To assess the scalability of GPUZIP, the best-performing configuration, which combines Prefetching with Bitcomp compression, was tested in a distributed environment using multiple nodes. The parallelization strategy involved distributing the seismic shots among these nodes. The scalability figure shows that the observed speedup increases with the number of workers used. Ideally, this speedup should scale linearly with the number of nodes; however, due to the communication overhead in multi-node environments, actual performance tends to fall short of this theoretical maximum. A speedup of approximately 80% of the expected performance based on the number of nodes was achieved, a trend consistently observed across all evaluated datasets.

Based on its demonstrated balance between compression ratio, memory usage, and performance, the experiments used the *Revolve* profile, configured with Bitcomp and a checkpoint cache size of 3. The scaling evaluation compared the baseline single-node



(a) *Revolve*

(b) *zCut*

(c) *Uniform*

Figure 6.3: Execution time breakdown. The data corresponds to the Salt dataset for each checkpointing library. Label (N,L) means N cache positions, compressing using the L library. (N, -) means N cache positions with no compression.

execution against distributed runs with 2, 3, 4, and 6 nodes. OpenMP Cluster [71] coordinated the workload, with each worker responsible for processing a subset of the total shots.

6.3 Chapter Conclusion

This chapter demonstrated that combining checkpoint data compression and prefetching yields significant performance improvements in Awave-3D. Individually, each technique addresses a key bottleneck in the checkpointing workflow: compression reduces data volume over PCIe, and prefetching hides latency by overlapping communication with computation. When integrated, they complement each other, resulting in faster and more efficient execution. Additionally, compression significantly reduced memory usage in the cache, allowing for larger cache sizes even when GPU memory is limited.

Cache Size	Compressor	Large (GB)	Marmousi3D (GB)	M3D_Larger (GB)	Salt (GB)
2	-	1.0	0.8	7.0	0.9
2	Bitcomp	0.3	0.6	2.7	0.4
2	cuZFP	0.2	0.2	1.7	0.3
3	-	1.6	1.2	10.5	1.4
3	Bitcomp	0.4	0.5	4.0	0.6
3	cuZFP	0.7	0.3	2.6	0.4
4	-	2.1	1.6	14.0	1.8
4	Bitcomp	0.5	0.6	5.4	0.8
4	cuZFP	0.5	0.4	3.5	0.5
5	-	2.6	2.0	17.5	2.3
5	Bitcomp	0.6	0.8	6.7	1.0
5	cuZFP	0.6	0.5	4.4	0.7
6	-	3.1	2.4	21.0	2.8
6	Bitcomp	0.8	0.9	8.1	1.2
6	cuZFP	0.7	0.6	5.3	0.7

Table 6.4: GPU Memory required by the *GPU Checkpoint Cache* on each GPU in the system, considering no compression (same as Chapter 4) and the combination of checkpoint prefetching and compression with Bitcomp and cuZFP.

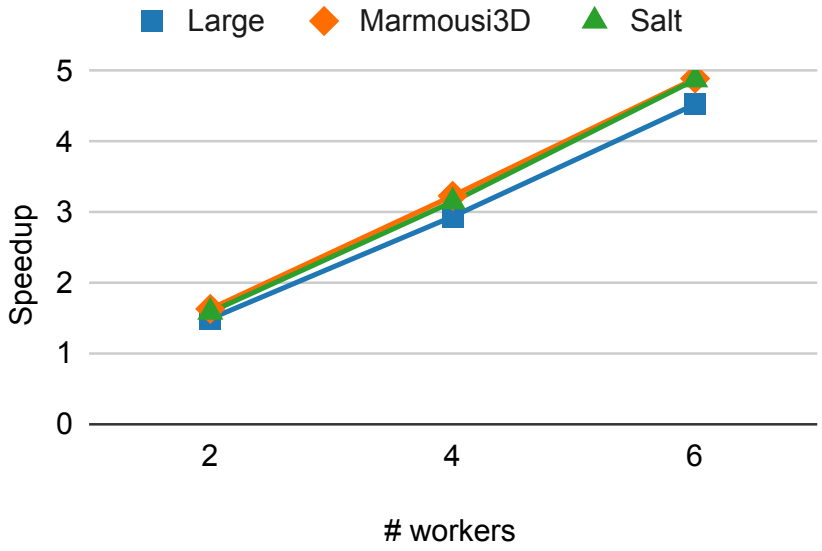


Figure 6.4: Scalability of GPUZIP with the *Revolve* profile (Bitcomp + *GPU Checkpoint Cache* size of 3) across varying numbers of workers (2, 3, 4, and 6) for the Large, Marmousi3D, and Salt datasets.

The methodology of the mechanisms that accelerate checkpointing is concluded. The next chapter will provide an in-depth look at the architecture of GPUZIP as a modular library and API that can be extended to other applications beyond Awave-3D.

Chapter 7

GPUZIP as an API

To conduct the experiments needed for this research, it was crucial to be flexible and swift in adjusting compression parameters and algorithms to identify the optimal configuration for maximum performance. Achieving this adaptability required considerable software engineering efforts to consolidate all functionalities into a reusable and modular library; as a result, GPUZIP was developed.

GPUZIP is written in C++/CUDA and provides a framework for checkpointing in GPU-based systems, incorporating checkpoint prefetching and GPU-based compression support. The implementation of GPUZIP is organized around three main modules: Compression, Checkpointing Interface, and Prefetching. Its source code is available on GitHub [3][1]

This chapter is organized as follows: Section 7.1 introduces GPUZIP's main modules, APIs, and organization; Section 7.2 provides an example of how GPUZIP can be integrated into a real-world application; Section 7.3 demonstrates how to set up logging in GPUZIP to help with troubleshooting and evaluation of the parameters; and finally, Section 7.4 demonstrates an example of GPUZIPY, a Python wrapper for GPUZIP's Compressor module.

7.1 The Software Architecture of GPUZIP

GPUZIP is architected around three modular components: Compression, Checkpointing Interface, and Prefetching. The Compression and Checkpointing modules are designed as standalone components, each exposing interfaces that allow them to be reused or integrated independently into other projects. The Prefetching module builds upon the same concept, but depends on Compression and Checkpointing Interface modules; however, it is still agnostic to the user's compressor or checkpointing algorithm.

Using interfaces and abstract classes helps GPUZIP extend the library to support new compressors or checkpointing algorithms, which can be introduced without modifying the core Prefetching logic, supporting clean and maintainable code evolution.

[1]Pending approval for open-sourcing under MIT license.

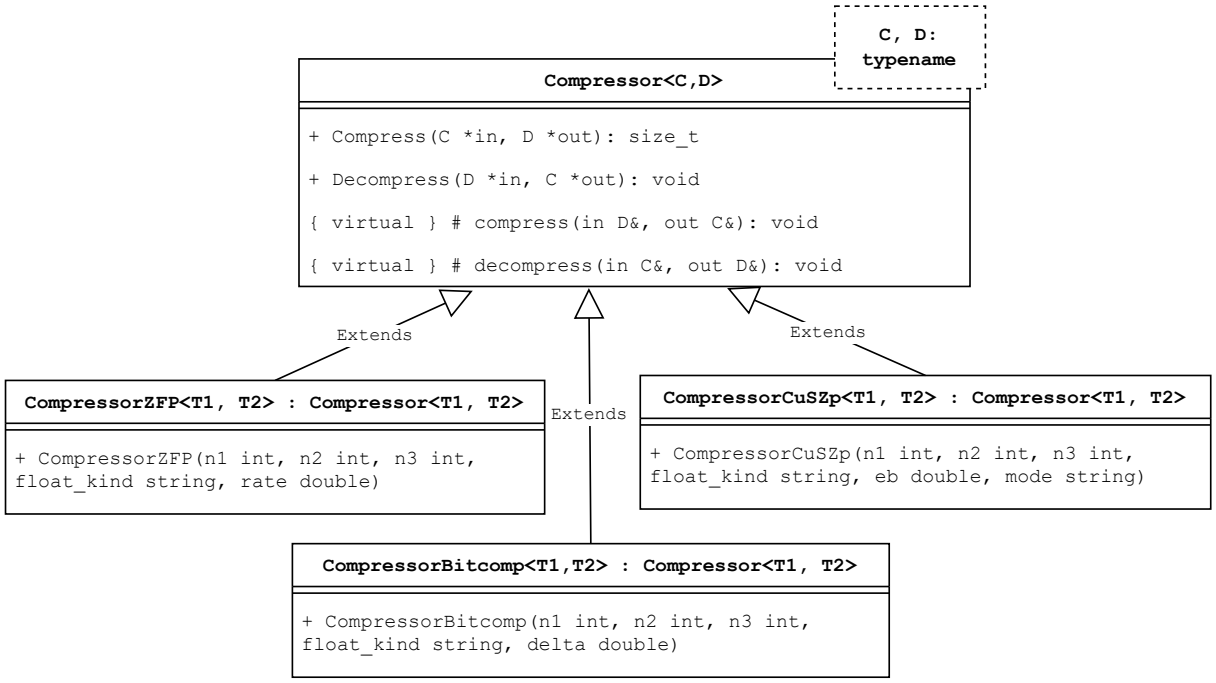


Figure 7.1: Compressor module class diagram (UML).

7.1.1 Compression Module

Following Figure 7.1, the Compression module defines a base abstract class (`Compressor<C, D>`) where "`C`" is the data type of the compressed data (e.g., integer) and "`D`" is the data type of the decompressed data (e.g., float).

The `Compressor` interface provides public methods to compress and decompress data on the GPU.

- `Compress(...)`: This function receives a pointer to uncompressed GPU data (`in`) and writes the compressed result to the output buffer (`out`). It returns the real size of the compressed data.
- `Decompress(...)`: This function receives a pointer to compressed GPU data (`in`) and writes the decompressed result to the output buffer (`out`).

Concrete implementations inherit from `Compressor<C, D>` abstract class to encapsulate the specific logic of each compression backend, such as `CompressorZFP`, `CompressorCuSZp`, and `CompressorBitcomp`. Each child class overrides the "`compress`" and "`decompress`" *virtual* methods, adapting them to the behavior and constraints of the respective compression library.

7.1.2 Checkpointing Interface Module

The second module of GPUZIP is the Checkpointing Interface, which defines a unified abstraction layer to decouple the application logic from the used checkpointing algorithm. This design allows users to switch between different checkpointing strategies, such as *Revolve*, *zCut*, or custom implementations, without modifying the user's code.

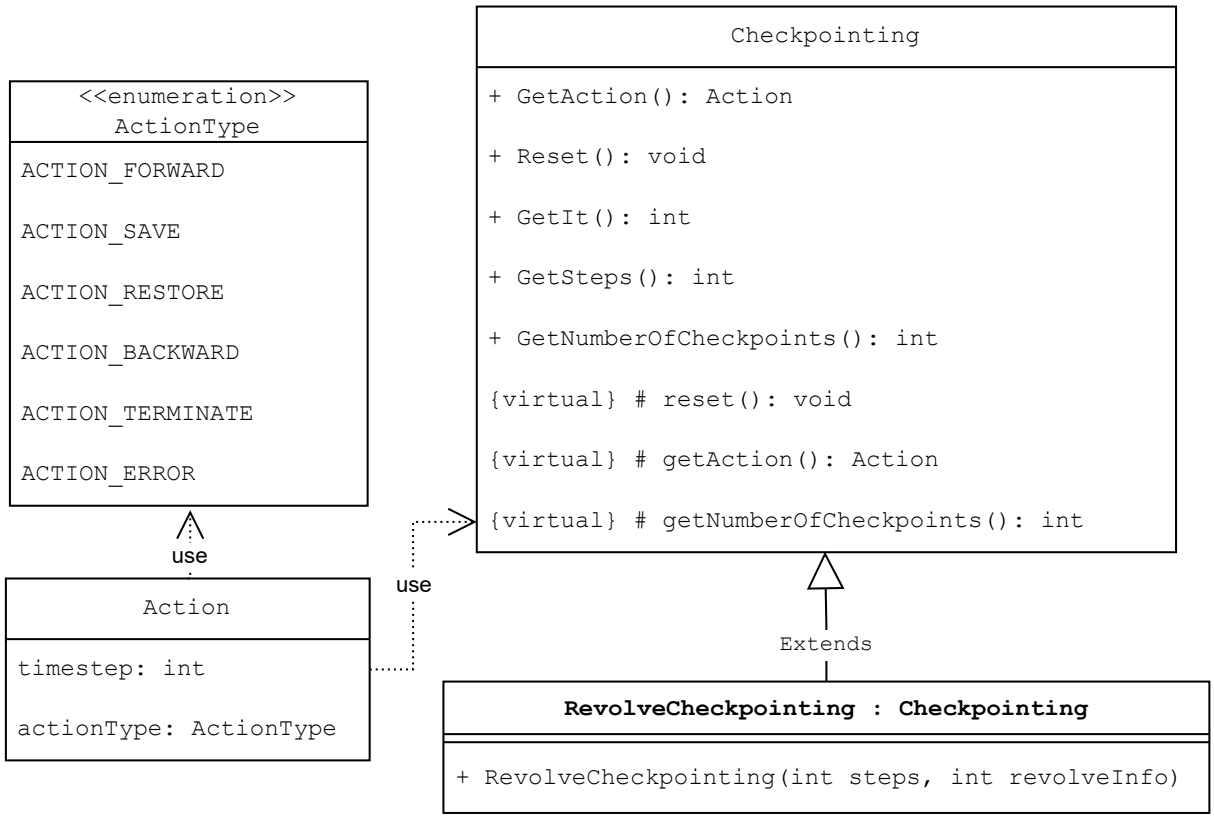


Figure 7.2: Checkpointing module class diagram (UML)

Figure 7.2 illustrates the internal architecture of this module. At its core, it includes an enumeration named `ActionType` and a `Action` structure, standardizing the semantics of checkpointing operations across all implementations. This standard representation enables consistent communication between the user's code and the checkpointing algorithm, regardless of the specific algorithm.

The central abstraction is the `Checkpointing` class, an abstract base class that exposes public methods. The most important, from the user's perspective, is the `GetAction()` method: it returns an `Action` object indicating the type of operation to perform (e.g., `ACTION_SAVE`, `ACTION_RESTORE`, `ACTION_BACKWARD`) and the timestep at which it should occur.

Specific checkpointing algorithms are implemented by extending the `Checkpointing` class. For instance, `RevolveCheckpointing` inherits from `Checkpointing` and encapsulates the logic of the *Revolve* algorithm.

7.1.3 Prefetching Module

The Prefetching module implements the core logic of the prefetching mechanism and checkpoint data compression, as detailed in Chapter 4. This module depends on the `Checkpointing` and `Compressor` interfaces but remains agnostic to the specific libraries being used.

As illustrated in Figure 7.3, the `Prefetch` class internally maintains two key components: the *GPU Checkpoint Cache* and the *Prefetch Action Vector* (PAV), which are

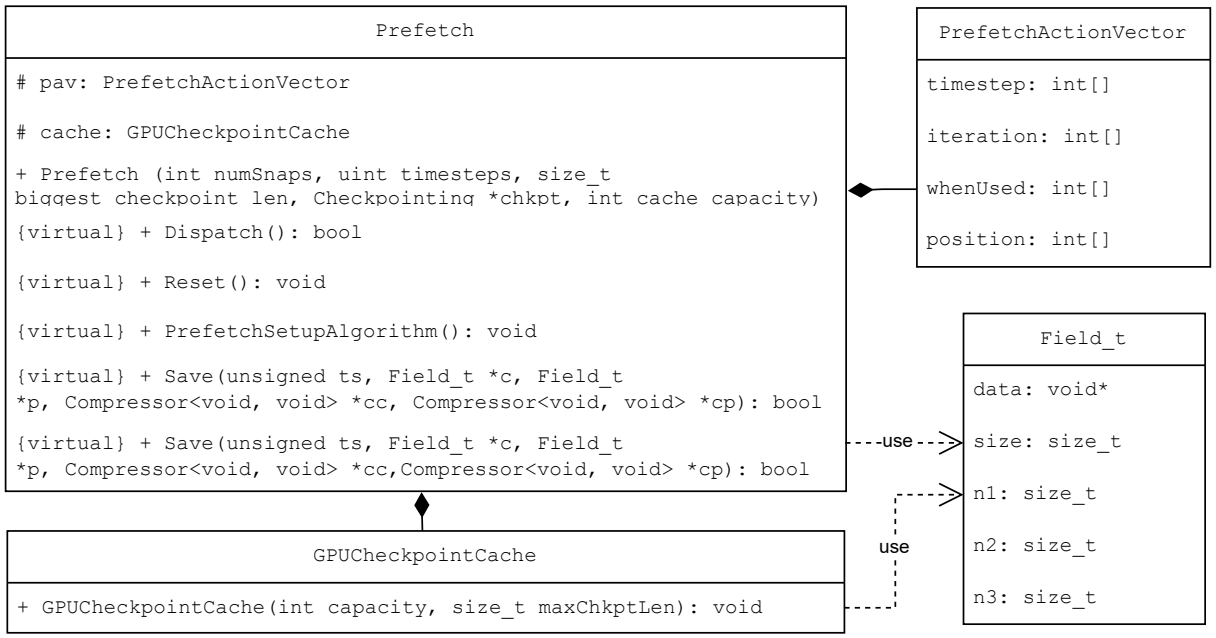


Figure 7.3: Prefetching module class diagram (UML).

protected attributes. All public methods operate on a unified abstraction of the data fields to be checkpointed, represented by the `Field_t` structure. This structure includes a pointer to a GPU-allocated buffer (`void*`), the field size in bytes, and the dimensions of the 3D data (n1, n2, n3).

The core functionality is exposed via the `Save(...)` and `Retrieve(...)` methods. These methods accept `Field_t` structures for the current and previous fields and optional compressor instances when compression is used.

The *Prefetch Setup Algorithm* (PSA), responsible for computing the PAV, is implemented as a `virtual` method, which enables the module to be extended with custom PSA implementations by subclassing and overriding `setup()` method.

Once PSA computes the PAV, the user must call `Dispatch()` in every iteration. This method checks if a prefetch action is scheduled for the current timestep and, if so, initiates the asynchronous memory transfer.

The module also provides a `Reset()` method to reinitialize all internal counters and clear the cache and PAV at the beginning of a new shot.

The `Prefetch` class is the central component for enabling prefetching in GPUZIP and adheres to the same object-oriented architecture as the `Checkpointing` and `Compressor` modules. For cases where prefetching is not desired, GPUZIP also includes a minimalistic implementation named `CheckpointingOnly`, which supports only saving and restoring snapshots, optionally with compression. This class serves as the experimental baseline for evaluating the benefits of the prefetching mechanism later integrated into Awave-3D.

7.2 GPUZIP Usage

In a nutshell, GPUZIP can offer checkpointing with prefetch and compression into an adjoint computation loop (such as RTM) in an immutable code that does not depend on

the configuration; the users do not need to change their source code because they want to experiment with other compressor libraries, use another checkpointing library or use a bigger checkpoint cache. Listing 7.1 shows how the integration is performed in a single GPU environment through a modular design that isolates configuration and functionality into builders and abstract interfaces.

```
#include "prefetch/Prefetch.cuh"
#include "prefetch/Checkpointing.hpp"
#include "common/GPUZIPBuilders.cpp"
#include "common/GPUZIPConfig.h"

Field_t BuildField(your_data_t* d) {
    size_t size = d->n1 * d->n2 * d->n3 * sizeof(float);
    return Field_t{d->n1, d->n2, d->n3, d->data, size};
}

void adjoint(gpuzip_config_t *cfg, your_data_t *data, int steps) {
    Checkpointing *chkpt = CheckpointingBuilder(cfg, steps);
    Prefetch *prefetch = PrefetchBuilder(cfg, steps, chkpt);

    for (int shot = 0; shot <= data->shots; ++shot) {
        bool done = false;
        prefetch->Setup();
        do {
            Action act = chkpt->GetAction();
            prefetch->Dispatch(chkpt->GetIt());

            if (act.actionType == ACTION_SAVE) {
                Field_t c = BuildField(data->curr);
                Field_t p = BuildField(data->prev);
                auto cc = CompressorBuilder(cfg, c.n1, c.n2, c.n3);
                auto cp = CompressorBuilder(cfg, p.n1, p.n2, p.n3);
                prefetch->Save(act.ts, &c, &p, cc.get(), cp.get());
            }
            if (act.actionType == ACTION_FORWARD)
                forward_computation(act.ts, data);
            if (act.actionType == ACTION_BACKWARD)
                backward_computation(act.ts, data);
            if (act.actionType == ACTION_RESTORE) {
                Field_t c = BuildField(data->curr);
                Field_t p = BuildField(data->prev);
                auto cc = CompressorBuilder(cfg, c.n1, c.n2, c.n3);
                auto cp = CompressorBuilder(cfg, p.n1, p.n2, p.n3);
                prefetch->Retrieve(act.ts, &c, &p, cc.get(), cp.get());
            }
            if (act.actionType == ACTION_TERMINATE)
                done = true;
            if (act.actionType == ACTION_ERROR)
                done = true;
        } while (!done);
    }
    prefetch->Free();
}
```

Listing 7.1: Example of an adjoint computation (such as RTM) integration with GPUZIP

The integration core begins in lines 12-13, where `CheckpointingBuilder` and `PrefetchBuilder` instantiate the appropriate checkpointing and caching strategy based on the parameters provided in `gpuzip_config_t`. This configuration `struct` can define the cache size, the compressor to be used (Bitcomp, cuSZp, cuZFP, etc.), and the checkpointing algorithm (*Revolve*, *zCut*, etc.) without requiring any further changes to the main application code.

Within the main shot loop (line 16), GPUZIP orchestrates the prefetching and checkpointing logic: `prefetch->Setup()` (line 17) runs the *Prefetch Setup Algorithm* that will define *Prefetch Action Vector*, making the system able to prefetch. In the inner loop (lines 18–44), the application receives checkpointing decisions from `chkpt->GetAction()` (line 19), which returns the next required operation, such as `SAVE`, `RESTORE`, `FORWARD` or `BACKWARD`, along with the associated timestep.

The `SAVE` operation (lines 22–28) sets up `Field_t` with the current and previous values computed in the last iteration, and then the compressor for the respective fields. Similarly, `RESTORE` (lines 33–39) retrieves the checkpoint data using the same pattern. `FORWARD` and `BACKWARD` computation kernels (lines 29-32) are untouched, demonstrating that GPUZIP integrates into the existing simulation loop without disrupting computational logic.

Finally, the user can `prefetch->Free()` (line 46) and release the host and GPU memory allocated by GPUZIP.

The modular design allows developers to change checkpointing strategies, compressors, or cache configurations entirely through the gpuzip_config_t structure, enabling rapid experimentation and tuning without modifying the simulation code. This design philosophy is critical for large-scale scientific codes, where maintainability and extensibility are essential.

A demonstration with multi-GPU and the required build configuration and dependencies to incorporate GPUZIP into a real-world project can be found in the Appendix A.

7.3 Logging: Evaluating and Troubleshooting

GPUZIP provides several mechanisms to aid in evaluating performance and diagnosing issues with its configuration and execution.

GPUZIP offers a static utility class named `GPUZIPLogger` for more detailed logging. Enabling the performance trace feature via `GPUZIPLogger::PerfTraceSwitch(true);` allows users to see information about internal structures such as the length and memory usage of the *GPU Checkpoint Cache* and *Checkpoint Pool*.

GPUZIPLogger also supports configurable log levels, which are essential for troubleshooting. Logging verbosity can be set using `GPUZIPLogger::SetLevel(int)`, with supported levels being: `0=DEBUG`, `1=INFO`, `2=WARN`, and `3=ERROR`. At the `DEBUG` level, GPUZIP prints detailed trace messages, including internal *GPU Checkpoint Cache* states and every action the prefetching mechanism takes. The `INFO` level outputs general configuration details and includes messages from the `WARN` and `ERROR` levels, which report abnormal behavior such as misconfigurations or failures in memory transfers.

Invoking `prefetch->Report()` method at the end of the run. It provides a quick

overview of the current prefetching configuration's effectiveness. This lightweight method prints useful statistics to the console, including the number of cache hits and misses, as well as the number of cache misses successfully avoided by prefetching.

Additionally, GPUZIP supports integration with NVIDIA's NSight profiling tools through the NVTX (NVIDIA Tools Extension) library. By defining the macro `USE_NVTX` at compile time, developers can insert NVTX markers into the execution trace, allowing for fine-grained performance analysis. For example, this can be enabled in a `CMakeLists.txt` file with `add_definitions(-DUSE_NVTX)`.

7.4 GPUZIPy: A Python Wrapper for GPUZIP

GPUZIPy is a Python wrapper for GPUZIP designed to integrate GPU-based compression into Python applications. Currently, only the Compressor module is exposed to Python, enabling seamless compression and decompression using Python-based scientific computing stacks.

The wrapper is built using `pybind11`[2], a tool to expose C++ types in Python and vice-versa, and operates on `cuPy`[3], a NumPy-compatible[4] array library allowing GPU memory management and CUDA computation from Python.

To use GPUZIPy, data originally residing in the host (as a NumPy array) must first be transferred to the GPU using `cuPy`. Once on the GPU, the compression and decompression routines can be invoked using the interfaces provided by GPUZIPY. Listing 7.2 shows a basic example of setting up a compressor, transferring data to the GPU, and invoking the compression and decompression methods.

The example starts by importing necessary modules (`gpuzipy`, `cupy`, and `numpy`). A compressor is instantiated based on the configuration (`Bitcomp` or `cuZFP`), as shown in lines 10-23. Line 27 creates a Numpy array for exemplification and transfers the input data from host to device using `cuPy` in line 30. Line 31 obtains the estimated compressed buffer size to allocate memory for receiving the compressed data in line 32. Finally, line 33 compresses the data using the unified `compress(...)` helper function. The same approach applies to decompression with the `decompress(...)` helper (lines 36-37).

7.5 Chapter Conclusion

GPUZIP was developed to provide a modular, reusable, and high-performance framework for GPU-based checkpointing, prefetching, and compression. Its architecture, organized into the Compression, Checkpointing Interface, and Prefetching modules, ensures that each component can evolve independently while remaining interoperable. By exposing abstract interfaces, GPUZIP enables seamless integration of different algorithms, compressors, and caching strategies without modifying application logic, thereby fostering

[2] https://github.com/pybind/pybind11 – accessed Apr 25, 2025
[3] https://cupy.dev/ – accessed Apr 25, 2025
[4] https://numpy.org/ – accessed Apr 25, 2025

rapid experimentation and tuning. Through its API and configuration-driven design, GPUZIP facilitates the efficient integration of real-world scientific codes.

```
from gpuzipy import CompressorZFP, CompressorBitcomp,
    compressed_buffer_size, compressed_buffer_max_size, compress,
    decompress
import cupy as cp
import numpy as np
import math

BITCOMP = 1; CUZFP = 2

n1 = 100; n2 = 100; n3 = 100

def build_compressor(selected_compressor):
    compressor = None

    if selected_compressor == BITCOMP:
        ERROR_BOUND = 2
        ALGO_DEFAULT = 'default'
        delta=1e-8
        compressor = CompressorBitcomp(n1, n2, n3, ERROR_BOUND, 0.0,
    0.0, delta, 'float', ALGO_DEFAULT)

    elif selected_compressor == CUZFP:
        max_bits = 8
        compressor = CompressorZFP(n1, n2, n3, 'float', max_bits)

    return compressor

# Setup phase
compressor = build_compressor(CUZFP)
h_uncompressed = np.random.rand(n1, n2, n3).astype(np.float32)

# Compression phase
d_uncompressed = cp.asarray(h_uncompressed)
estimated_size = compressed_buffer_max_size(compressor)
d_compressed_ptr = cp.cuda.malloc_async(estimated_size)
compress(compressor, d_uncompressed.data.ptr, d_compressed_ptr.ptr)

# Decompression phase
d_decompressed = cp.empty((n1, n2, n3), dtype=np.float32)
decompress(compressor, d_compressed_ptr.ptr, d_decompressed.data.ptr)
```

Listing 7.2: Example of using GPUZIPY to compress data on the GPU

Chapter 8

Final Remarks

This chapter concludes the dissertation, Section 8.1 highlights the limitations and future directions for expanding GPUZIP. The Section 8.2 then summarizes the key elements of the dissertation, reviewing how the main objective, research questions, and specific goals were addressed through the methodology and the academic contributions of this work.

8.1 Limitations & Outlook

This work was developed within a specific time frame and computational context for HPC applications. As such, it inevitably carries certain limitations, many of which naturally suggest promising avenues for future research and development.

Scientific Domains: While GPUZIP was evaluated in the context of Reverse Time Migration (RTM), its design is general enough to apply to other scientific domains facing similar performance bottlenecks. For instance, the *zCut* checkpointing algorithm (proposed initially for Machine Learning workloads) demonstrated strong results when integrated with GPUZIP. This outcome suggests that GPUZIP could benefit Machine Learning workloads, as well as other domains where checkpointing is performance-critical.

Porting to other Hardware and Architectures: All experiments in this work were conducted on NVIDIA V100 GPUs using the programming models initially implemented in Awave-3D. Porting GPUZIP to alternative GPU architectures, such as those from AMD, which are currently employed in the main systems on the TOP500 list, could unlock performance gains for a broader range of applications.

Unified Virtual Memory (UVM) & Multi-Tiered Checkpointing Pool: Following the methodology outlined in [60], UVM can be utilized to store the entire checkpoint pool. Additionally, the Predictive Scheduler Algorithm (PSA) can be modified to provide direct hints to CUDA prefetching mechanisms, thereby replacing calls such as `Prefetch->Dispatch(it)`. In alignment with [52], it will be crucial to expand the pool and cache to other layers and incorporate multi-tiered storage systems utilizing SSDs. This enhancement is necessary for scaling to manage real-world data in larger seismic exploration fields that extend beyond the scope of this research. This approach is also relevant to other domains that require the handling of substantial amounts of data.

Multi-node & OMPC Integration: Distributed scalability tests in this work were

based on the same computational model used by Awave-3D, where OMPC distributes multiple shots across multiple nodes. While this achieved a speedup of approximately 80% of the theoretical ideal due to inter-node communication overheads, it also revealed opportunities for further optimization. In particular, integrating GPUZIP-based compression into OMPC's communication layer could reduce inter-node transfer volumes and improve scalability.

Energy Efficiency: The experiments demonstrate significant speedups, achieving up to 8.8× faster performance. In other words, the time required on working cluster nodes to solve the same problem is reduced by a factor of 8.8, which helps alleviate the long wait times typically experienced in HPC clusters. A study highlighting the energy savings achieved by GPUZIP would be highly valuable, particularly given the current discussions surrounding energy efficiency in both industry and academia.

The perspectives presented here stem from discussions between the author of this dissertation, Prof. Sandro Rigo, and the LSC members, as well as feedback from project collaborators and valuable insights provided by the dissertation examination board. Together, they outline a clear path for extending GPUZIP's applicability, performance, and portability in future research efforts.

8.2 Concluding Remarks

This research aimed to improve and accelerate checkpointing in heterogeneous computing systems with GPUs by reducing GPU-host communication overhead through prefetching techniques and data compression. Throughout this dissertation, the Research Questions (RQ) outlined in Chapter 1 were addressed through the development and validation of the Specific Objectives (SO).

RQ1 asked whether checkpointing algorithms provide sufficient hinting to support an effective prefetching mechanism. Chapter 4 answered this question addressing the SO1, SO2 and SO3. It described the implementation of the *GPU Checkpoint Cache* structure, which includes a synchronization mechanism and an LRU eviction policy (SO1). This structure proved essential given that checkpointing algorithms tend to reuse snapshots. The *Prefetch Setup Algorithm* was also implemented to identify when a *cache miss* would occur and to schedule a prefetch in advance (SO2).

SO3 involved testing multiple cache sizes (2 to 6) across all checkpointing algorithms to determine the most effective configuration. Results showed that *Revolve* is the most sensitive to cache size due to its frequent reuse of snapshots. In contrast, *zCut* suffers from cache saturation due to high snapshot density, and *Uniform* requires less cache due to the ample time between `RESTORE` actions, making two slots sufficient.

At this stage, speedups of up to 3.4× for *Revolve*, 1.7× for *zCut*, and 2.3× for *Uniform* were achieved by reducing (though not eliminating) PCIe blocking time. No cache misses occurred; however, in some cases, the prefetched data did not arrive in time, resulting in residual communication latency.

Thus, the answer to RQ1 is that checkpointing algorithms provide enough hinting for prefetching all required data. Despite this, timing limitations prevent prefetching alone

from eliminating PCIe transfer stalls.

RQ2, addressed in Chapter 5 through SO4 and SO5, evaluated whether compression could reduce PCIe overhead without compromising output quality. Available GPU-based compressors were assessed, and the best performers, cuZFP and NVIDIA's Bitcomp, were selected after warmup testing. Their optimal parameters were determined and used throughout the experiments.

Compression alone achieved speedups with Bitcomp for *Revolve* and *zCut* due to its higher compression ratios and faster throughput, reaching up to 4.57× and 6.50× respectively. For *Uniform*, Bitcomp outperformed on the Large dataset, while cuZFP delivered better results for Marmousi3D and Salt (up to 5.09×). Compression also enabled more snapshots to be stored, which reduced recomputation in *Uniform*. Additionally, both compressors preserved image quality, as confirmed by PSNR, SSIM metrics, and visual inspection. Therefore, RQ2 is answered: compression reduces PCIe overhead without significantly degrading the quality of the results.

RQ3, addressed in Chapter 6 through SO6, evaluated the combination of prefetching and compression. GPUZIP integrates both mechanisms, using the *GPU Checkpoint Cache* and *Prefetch Setup Algorithm* alongside the compressor modules. The combined approach achieved speedups up to 5.1× for *Revolve*, 8.8× for *zCut*, and 5.8× for *Uniform*, outperforming each technique in isolation.

Chapter 6 also demonstrated that compression enhances prefetching by reducing transfer size, making transferring faster, and enabling larger caches, even on GPUs with limited memory. Prefetching complements compression by hiding *Host-to-Device* transfer latency, which would otherwise be synchronous.

In Chapter 7, the research is wrapped into GPUZIP [3][1], a reusable and extensible library with a Python API. GPUZIP made experimentation efficient by allowing multiple configurations to be tested without changing the Awave-3D source code. GPUZIP invites contributions and adoption in other scientific domains.

Finally, this work generated academic contributions, the GPUZIP research was presented in a poster at Super Computing 2023 [45], a paper was accepted for the EUROPAR 2024 conference [44], and an article was published in the IJHPCA in 2025 [46].

[1]Pending approval for open-sourcing under MIT license.

Bibliography

[1] NVIDIA NVComp. https://developer.nvidia.com/nvcomp. Accessed: May 26, 2025.

[2] Docker hub image maltempi/awave-dev:ompc. https://hub.docker.com/layers/maltempi/awave-dev/ompc/images/sha256-25ddcffeae92ae033359acfd7e4dd5388bce4c23349037bee2eaed64644791e4, 2025. Accessed: May 26, 2025.

[3] Gpuzip - pending approval for open-source. https://github.com/LSC-Unicamp/GPUZIP, 2025. Accessed: May 26, 2025.

[4] Lauri Ahlroth, Olli Pottonen, and André Schumacher. *Approximately Uniform Online Checkpointing*, pages 297–306. 2011.

[5] A. Aldarwish and S. Almehaid. Novel approach to optimize heterogenous high-performance computing resource utilization for hybrid seismic imaging algorithms. 2025(1):1–5, 2025.

[6] AMD. Amd cdna 3 architecture - the all-new amd gpu architecture for the modern era of hpc and ai. Technical report, AMD, 2025.

[7] Fred Aminzadeh, Jérôme Brac, and Thomas Kunz. *3-D Salt and Overthrust Models*. SEG/EAGE 3-D Modeling Series No. 1. Society of Exploration Geophysicists, Tulsa, 1997.

[8] Carlos HS Barbosa and Alvaro LGA Coutinho. Reverse time migration with lossy and lossless wavefield compression. In *2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 192–201. IEEE, 10 2023.

[9] E. Baysal, D. D. Kosloff, and J. W.C. Sherwood. Reverse time migration. *Geophysics*, 48(11):1514–1524, 1983.

[10] Christian Boehm, Mauricio Hanzich, Josep de la Puente, and Andreas Fichtner. Wavefield compression for adjoint methods in full-waveform inversion. *Geophysics*, 81(6):R385 – R397, 2016-11. Received 23 November 2015, Accepted 8 July 2016, Published 13 September 2016.

[11] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. *IEEE Transactions on Computers*, 56:889–908, 7 2007.

[12] Franck Cappello, Allison Baker, Ebru Bozda, Martin Burtscher, Kyle Chard, Sheng Di, Paul Christopher O Grady, Peng Jiang, Shaomeng Li, Erik Lindahl, et al. Lossy compression of scientific data: Applications constrains and requirements. *arXiv preprint arXiv:2503.20031*, 2025.

[13] Richard W. Carr and John L. Hennessy. Wsclock—a simple and effective algorithm for virtual memory management. In *Proceedings of the eighth symposium on Operating systems principles - SOSP '81*, pages 87–95. ACM Press, 1981.

[14] NVIDIA Corporation. Nvidia tesla v100 gpu architecture. Technical report, NVIDIA Corporation, 2017.

[15] NVIDIA Corporation. Nvidia tesla a100 tensor core gpu architecture. Technical report, NVIDIA Corporation, 2020.

[16] NVIDIA Corporation. Nvidia h100 tensor core gpu architecture. Technical report, NVIDIA Corporation, 2023.

[17] NVIDIA Corporation. Nvidia blackwell architecture technical brief. Technical report, NVIDIA Corporation, 2025.

[18] Sheng Di, Jinyang Liu, Kai Zhao, Xin Liang, Robert Underwood, Zhaorui Zhang, Milan Shah, Yafan Huang, Jiajun Huang, Xiaodong Yu, Congrong Ren, Hanqi Guo, Grant Wilkins, Dingwen Tao, Jiannan Tian, Sian Jin, Zizhe Jian, Daoce Wang, Md Hasanur Rahman, Boyuan Zhang, Shihui Song, Jon Calhoun, Guanpeng Li, Kazutomo Yoshii, Khalid Alharthi, and Franck Cappello. A survey on error-bounded lossy compression for scientific datasets. *ACM Computing Surveys*, 5 2025.

[19] M Dmitriev, T Tonellot, HJ AlSalem, and S Di. Error-bounded lossy compression in reverse time migration. In *Sixth EAGE High Performance Computing Workshop*, volume 2022, pages 1–5. EAGE Publications BV, 2022.

[20] Eric Dussaud. Computational strategies for reverse-time migration. Technical report, 2008.

[21] P G Emma, A Hartstein, T R Puzak, and V Srinivasan. Exploring the limits of prefetching. Technical report, 2005.

[22] John Etgen, Samuel H. Gray, and Yu Zhang. An overview of depth imaging in exploration geophysics. *Geophysics*, 74(6):WCA5–WCA17, 12 2009.

[23] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1 1992.

[24] Andreas Griewank and Andrea Walther. Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Trans. Math. Sw.*, 26(1):19–45, mar 2000.

[25] Swapan Kumar Haldar. *Exploration Geophysics*, pages 103–122. Elsevier, 2018.

[26] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.

[27] Nam Ho, Carlos Falquez, Antoni Portero, Estela Suarez, and Dirk Pleiter. Memory prefetching evaluation of scientific applications on a modern hpc arm-based processor. *IEEE Access*, 2025.

[28] Alain Hore and Djemel Ziou. Image quality metrics: Psnr vs. ssim. In *2010 20th International Conference on Pattern Recognition*, pages 2366–2369. IEEE, 8 2010.

[29] Yafan Huang, Sheng Di, Xiaodong Yu, Guanpeng Li, and Franck Cappello. cuszp: An ultra-fast gpu error-bounded lossy compression framework with optimized end-to-end performance. SC'23. ACM, 2023.

[30] Yafan Huang, Kai Zhao, Sheng Di, Guanpeng Li, Maxim Dmitriev, Thierry-Laurent D. Tonellot, and Franck Cappello. Towards improving reverse time migration performance by high-speed lossy compression. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 651–661. IEEE, 5 2023.

[31] Wen-Mei W Hwu, David B Kirk, and Izzat El Hajj. *Programming massively parallel processors*. Morgan Kaufmann, London, England, 4 edition, September 2022.

[32] Intel. Intel data center gpu max series. Technical report, Intel, 2023.

[33] Intel. App metrics for intel microprocessors. Technical report, Intel, 2025.

[34] Donghun Jeong, Jihun Park, and Jungrae Kim. Demand memcpy: Overlapping of computation and data transfer for heterogeneous computing. *IEEE Access*, 10:79925–79938, 2022.

[35] Jaehoon Jung, Daeyoung Park, Youngdong Do, Jungho Park, and Jaejin Lee. Overlapping host-to-device copy and computation using hidden unified memory. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 321–335. ACM, 2 2020.

[36] Martin Karp, Estela Suarez, Jan H Meinke, Måns I Andersson, Philipp Schlatter, Stefano Markidis, and Niclas Jansson. Experience and analysis of scalable high-fidelity computational fluid dynamics on modular supercomputing architectures. *The International Journal of High Performance Computing Applications*, 39(3):329–344, 2025.

[37] Navjot Kukreja, Jan Hückelheim, Mathias Louboutin, John Washbourne, Paul H.J. Kelly, and Gerard J. Gorman. Lossy checkpoint compression in full waveform inversion: a case study with zfpv0.5.5 and the overthrust model. *Geoscientific Model Development*, 15:3815–3829, 9 2020.

[38] Xin Liang, Kai Zhao, Sheng Di, Sihuan Li, Robert Underwood, Ali M. Gok, Jiannan Tian, Junjing Deng, Jon C. Calhoun, Dingwen Tao, Zizhong Chen, and Franck Cappello. Sz3: A modular framework for composing prediction-based error-bounded lossy compressors. *IEEE Transactions on Big Data*, 9:485–498, 4 2023.

[39] Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20:2674–2683, 12 2014. ZFP article.

[40] Hongwei Liu, Bo Li, Hong Liu, Xiaolong Tong, Qin Liu, Xiwen Wang, and Wenqing Liu. The issues of prestack reverse time migration and solutions with graphic processing unit implementation. *Geophysical Prospecting*, 60:906–918, 9 2012.

[41] Jie Liu, Bogdan Nicolae, and Dong Li. Lobster: Load balance-aware i/o for distributed dnn training. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–11, 2022.

[42] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman. Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration. *Geoscientific Model Development*, 12(3):1165–1187, 2019.

[43] Fabio Luporini, Mathias Louboutin, Michael Lange, Navjot Kukreja, Philipp Witte, Jan Hückelheim, Charles Yount, Paul H. J. Kelly, Felix J. Herrmann, and Gerard J. Gorman. Architecture and performance of devito, a system for automated stencil computation. *ACM Trans. Math. Softw.*, 46(1), apr 2020.

[44] Thiago Maltempi, Sandro Rigo, Marcio Pereira, Hervé Yviquel, Jessé Costa, and Guido Araujo. Combining compression and prefetching to improve checkpointing for inverse seismic problems in gpus. In Jesus Carretero, Sameer Shende, Javier Garcia-Blas, Ivona Brandic, Katzalin Olcoz, and Martin Schreiber, editors, *Euro-Par 2024: Parallel Processing*, pages 167–181, Cham, 2024. Springer Nature Switzerland.

[45] Thiago Maltempi, Sandro Rigo, Marcio Pereira, Hervé Yviquel, Jessé Costa, Alan Souza, and Guido Araujo. Introducing prefetching and data compression to accelerate checkpointing for inverse seismic problems. https://sc23.supercomputing.org/proceedings/tech_poster/poster_files/rpost129s3-file2.pdf, 2023. Poster presented at Super Computing 2023 (SC'23), Denver, CO, USA.

[46] Thiago Maltempi, Sandro Rigo, Marcio Pereira, Hervé Yviquel, Gustavo Leite, Orlando Lee, Jessé Costa, and Guido Araujo. Checkpointing fine-tuning for accelerating seismic applications in gpus. *The International Journal of High Performance Computing Applications*, 2025.

[47] Thiago Maltempi, Sandro Rigo, Hervé Yviquel, Márcio Pereira, Jessé Costa, and Guido Araújo. Replication data for GPUZIP v2.0: accelerating checkpointing on GPUs with prefetching and compression, 2024.

[48] A. S.I. Margetis, E. M. Papoutsis-Kiachagias, and K. C. Giannakoglou. Lossy compression techniques supporting unsteady adjoint on 2d/3d unstructured grids. *Computer Methods in Applied Mechanics and Engineering*, 387:114152, 12 2021.

[49] Andreas Stefanos I. Margetis, Evangelos M. Papoutsis-Kiachagias, and Kyriakos C. Giannakoglou. Reducing memory requirements of unsteady adjoint by synergistically using check-pointing and compression. *International Journal for Numerical Methods in Fluids*, 95:23–43, 1 2023.

[50] Marfurt K. Martin G., Wiley R. Marmousi2: An elastic upgrade for marmousi. *The Leading Edge*, 25(2):156–166, February 2006.

[51] Avinash Maurya, Bogdan Nicolae, M. Mustafa Rafique, and Franck Cappello. Towards efficient i/o pipelines using accumulated compression. In *2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 256–265, 2023.

[52] Avinash Maurya, M. Mustafa Rafique, Thierry Tonellot, Hussain J. AlSalem, Franck Cappello, and Bogdan Nicolae. Gpu-enabled asynchronous multi-level checkpoint caching and prefetching. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 73–85. ACM, 8 2023.

[53] Avinash Maurya, Jie Ye, M. Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. Breaking the memory wall: A study of i/o patterns and gpu memory utilization for hybrid cpu-gpu offloaded optimizers. In *Proceedings of the 14th Workshop on AI and Scientific Computing at Scale using Flexible Computing Infrastructures*, pages 9–16. ACM, 6 2024.

[54] Mike Mikailov, Fu-Jyh Luo, Stuart Barkley, Lohit Valleru, Stephen Whitney, Zhichao Liu, Shraddha Thakkar, Weida Tong, and Nicholas Petrick. Scaling bioinformatics applications on hpc. *BMC Bioinformatics*, 18:501, 12 2017.

[55] Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys*, 49:1–35, 6 2017.

[56] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for http. In *Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 181–194. ACM, 10 1997.

[57] NVIDIA Corporation. *CUDA C Programming Guide*, 2025. Version 12.8.

[58] Nicholas Okita, Alexandre Camargo, José Ribeiro, Tiago Coimbra, Caian Benedicto, and Jorge Faccipieri Junior. High-performance computing strategies for seismic-imaging software on the cluster and cloud-computing environments. *Geophysical Prospecting*, 70, 10 2021.

[59] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the fifteenth ACM symposium on Operating systems principles - SOSP '95*, pages 79–95. ACM Press, 1995.

[60] Pedro Rigon, Brenda Schussler, Alexandre Sardinha, Pedro M. Silva, Fábio Oliveira, Alexandre Carissimi, Jairo Panetta, Filippo Spiga, Arthur Lorenzon, and Philippe O. A. Navaux. Harnessing data movement strategies tonbsp;optimize performance-energy efficiency ofnbsp;oil & gas simulations innbsp;hpc. In *Euro-Par 2024: Parallel Processing: 30th European Conference on Parallel and Distributed Processing, Madrid, Spain, August 26–30, 2024, Proceedings, Part II*, page 211–225, Berlin, Heidelberg, 2024. Springer-Verlag.

[61] Jingcheng Shen, Linbo Long, Xin Deng, Masao Okita, and Fumihiko Ino. A compression-based memory-efficient optimization for out-of-core gpu stencil computation. *The Journal of Supercomputing*, Feb 2023.

[62] Jingcheng Shen, Yifan Wu, Masao Okita, and Fumihiko Ino. Accelerating gpu-based out-of-core stencil computation with on-the-fly compression. In *PDCAT'22*, pages 3–14, Cham, 2022. Springer.

[63] Cristina Silvano, Daniele Ielmini, Fabrizio Ferrandi, Leandro Fiorin, Serena Curzel, Luca Benini, Francesco Conti, Angelo Garofalo, Cristian Zambelli, Enrico Calore, et al. A survey on deep learning hardware accelerators for heterogeneous hpc platforms. *ACM Computing Surveys*, 2023.

[64] William W. Symes. Reverse time migration with optimal checkpointing. *Geophysics*, 72, 2007.

[65] TOP500. Ogbon cimatec/petrobras - bull sequana x1000, xeon gold 6240 18c 2.6ghz, mellanox infiniband edr, nvidia tesla v100 sxm2. https://top500.org/system/179703/, 2024. Accessed: May 26, 2025.

[66] TOP500. Santos dumont (sdumont) - bull sequana x1000, xeon gold 6252 24c 2.1ghz, mellanox infiniband edr, nvidia tesla v100 sxm2. https://top500.org/system/179704/, 2024. Accessed: May 26, 2025.

[67] TOP500. Top 500 list from november 2024. https://top500.org/lists/top500/2024/11/, 2024. Accessed: May 26, 2025.

[68] under review. zcut: An optimization algorithm for checkpointing computional graphs. Under review, 2024.

[69] Steven P. Vanderwiel and David J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32:174–199, 6 2000.

[70] Wu Wang, Wei Zhang, and Tao Lei. Compression of seismic forward modeling wavefield using tuckermpi. *Computers Geosciences*, 172:105298, 3 2023.

[71] Hervé Yviquel, Marcio Pereira, Emílio Francesquini, Guilherme Valarini, Gustavo Leite, Pedro Rosso, Rodrigo Ceccato, Carla Cusihualpa, Vitoria Dias, Sandro Rigo, Alan Souza, and Guido Araujo. The openmp cluster programming model. 2023.

[72] Boyuan Zhang, Jiannan Tian, Sheng Di, Xiaodong Yu, Yunhe Feng, Xin Liang, Dingwen Tao, and Franck Cappello. Fz-gpu: A fast and high-ratio lossy compressor for scientific computing applications on gpus. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 129–142. ACM, 8 2023.

Appendix A

Itegrating GPUZIP in a C++/CUDA Project

This appendix provides detailed instructions for integrating GPUZIP with C++ and CUDA projects. It describes the required dependencies, how to configure GPUZIP within a `CMakeLists.txt` file, and how to manage GPUZIP's compilation flags. Furthermore, it includes a practical example demonstrating GPUZIP's usage in a multi-GPU adjoint computation scenario.

The following versions were tested for compatibility:

- CMake 3.22
- Make 4.2
- NVCC 10.1
- CUDA 12.2
- cuZFP (refer to the *Installing cuZFP* tutorial)

Alternatively, the Docker image `maltempi/awave-dev:ompc` can be used to simplify the environment setup.

A.1 Example of `CMakeLists.txt` Configuration

```
set(CMAKE_CXX_STANDARD 17)

include_directories(./GPUZIP/src/Prefetch/include)
include_directories(./GPUZIP/src/Compressor/include)
add_library(cuda_utils ./GPUZIP/src/Compressor/cuda_utils.cu)

option(USE_NVTX "GPUZIP use NVTX (nsight) tracing." OFF)
if (USE_NVTX)
    add_definitions(-DUSE_NVTX)
endif()

```

```
# Enable ZFP compression
option(ZFP "Includes cuZFP as an available compressor." ON)
if (ZFP)
    add_definitions(-DZFP)
    include_directories(/opt/zfp/include)

    if (NOT DEFINED CMAKE_CUDA_FLAGS)
        set(CMAKE_CUDA_FLAGS "")
    endif()
    set(CMAKE_CUDA_FLAGS "${CMAKE_CUDA_FLAGS} --extended-lambda --expt-relaxed-constexpr -Wno-deprecated-declarations")

    target_link_libraries(awave3d-decom
        /opt/zfp/lib/libzfp.so
        /opt/zfp/lib/libzfp.so.1
        /opt/zfp/lib/libzfp.so.1.0.0
        -lnvToolsExt
        -lcuda
        -lcusparse)
endif()

# Enable NVCOMP BITCOMP
option(NVCOMP_BITCOMP "Includes NVIDIA Bitcomp as an available compressor." ON)
if (NVCOMP_BITCOMP)
    add_definitions(-DNVCOMP_BITCOMP)
    message(STATUS "Using NVCOMP BITCOMP")

    include(FetchContent)

    FetchContent_Declare(
        NVCOMP_BITCOMP
        DOWNLOAD_EXTRACT_TIMESTAMP false
        URL https://developer.download.nvidia.com/compute/nvcomp/2.6.1/local_installers/nvcomp_2.6.1_x86_64_12.x.tgz
        URL_HASH SHA256=ac4834397291f245578af959694e816d96f80036eac50b5f24b113dee5b54225
        TLS_VERIFY false
    )
    FetchContent_MakeAvailable(NVCOMP_BITCOMP)

    FetchContent_GetProperties(NVCOMP_BITCOMP SOURCE_DIR NVCOMP_SRC_DIR)
    message(STATUS "NVCOMP_SRC_DIR: ${NVCOMP_SRC_DIR}")

    list(APPEND CMAKE_PREFIX_PATH "${NVCOMP_SRC_DIR}/lib/cmake/nvcomp")

    message(STATUS "CMAKE_PREFIX_PATH: ${CMAKE_PREFIX_PATH}")

    find_package(nvcomp REQUIRED CONFIG PATHS "${NVCOMP_SRC_DIR}/lib/cmake/nvcomp")

    target_link_libraries(awave3d-decom
        nvcomp::nvcomp_bitcomp
```

```
        cuda_utils
        -lnvToolsExt
        -lcuda
        -lcusparse)
endif()

# Enable cuSZp compression
option(CUSZP "Includes cuSZp as an available compressor." ON)

if (CUSZP)
    add_definitions(-DCUSZP)
    message("Using CUSZP")
    include(FetchContent)

    FetchContent_Declare(
        cuszp
        GIT_REPOSITORY https://github.com/szcompressor/cuSZp.git
        GIT_TAG cuSZp-V1.1
    )
    FetchContent_MakeAvailable(cuszp)
    target_link_libraries(awave3d-decom
      cuSZp
      cuda_utils
      -lnvToolsExt
      -lcuda
      -lcusparse)
endif()
```

Listing A.1: Example of CMake configuration to include GPUZIP

A.2 Managing GPUZIP Compilation Flags

By default, GPUZIP enables all available compressors. For production environments, it is recommended to disable unused compressors to reduce binary size and simplify deployment. The flags `CUZFP`, `CUSZP`, and `BITCOMP` control whether the respective compressors are enabled. Additionally, the `USE_NVTX` flag enables internal instrumentation with NVIDIA NSight.

```
## Default configuration (all compressors enabled)
cmake

## Disabling cuZFP and cuSZP
cmake -DCUZFP=0 -DCUSZP=0

## Disabling Bitcomp only
cmake -DBITCOMP=0

## Enabling NVTX tracing
cmake -DUSE_NVTX=1
```

Listing A.2: Example of CMake flags usage

A.3 Example: Multi-GPU Adjoint Computing with Prefetch and Compression

The following example illustrates the usage of GPUZIP in a multi-GPU adjoint computation setup. The implementation is agnostic to the specific compressor or checkpointing algorithm used.

The variable `your_data_t` symbolizes user-defined data structures representing current and previous fields. The fields are partitioned across multiple GPUs.

All configurable parameters for GPUZIP are encapsulated within the `gpuzip_config_t` structure, defined in `Prefetch/include/common/GPUZIPBuilders.cpp`.

```
#include "prefetch/Prefetch.cuh"
#include "prefetch/Checkpointing.hpp"
#include "common/GPUZIPBuilders.cpp"
#include "common/GPUZIPConfig.h"

void adjoint(gpuzip_config_t *gpuzip_config, your_data_t *data, int
    num_gpus) {
     GPUZIPLogger::SetLevel(gpuzip_config->log_level);
     GPUZIPLogger::PerfTraceSwitch(gpuzip_config->enable_performance_log)
    ;

     int steps = afd->nt;
     bool useCompression = gpuzip_config->compressor > 0;

     Checkpointing *chkpt = CheckpointingBuilder(gpuzip_config, steps);
     int snaps = chkpt->GetNumberOfCheckpoints();

     Prefetch *prefetch[num_gpus];
     for (int d = 0; d < num_gpus; d++) {
         size_t n1 = std::max(data->curr->devices[d].n1, data->prev->
    devices[d].n1);
         size_t n2 = std::max(data->curr->devices[d].n2, data->prev->
    devices[d].n2);
         size_t n3 = std::max(data->curr->devices[d].n3, data->prev->
    devices[d].n3);
         prefetch[d] = PrefetchBuilder(gpuzip_config, steps, chkpt);
     }

     for (int shot = 0; shot <= data->shots; shot++) {
         for (int d = 0; d < num_gpus; d++) {
             cudaSetDevice(d);
             prefetch[d]->Setup();
         }

         chkpt->Reset();
         bool terminate = false;

         do {
             Action action = chkpt->GetAction();
```

```
35
36          for (int d = 0; d < num_gpus; d++) {
37              cudaSetDevice(d);
38              prefetch[d]->Dispatch(chkpt->GetIt());
39          }
40
41          if (action.actionType == ACTION_SAVE) {
42              for (int d = 0; d < num_gpus; d++) {
43                  cudaSetDevice(d);
44
45                  Field_t curr = {
46                      .n1 = data->curr->devices[d].n1,
47                      .n2 = data->curr->devices[d].n2,
48                      .n3 = data->curr->devices[d].n3,
49                      .data = data->curr->devices[d].data,
50                      .size = data->curr->devices[d].n1 * data->curr->
devices[d].n2 * data->curr->devices[d].n3 * sizeof(float)
51                  };
52
53                  Field_t prev = {
54                      .n1 = data->prev->devices[d].n1,
55                      .n2 = data->prev->devices[d].n2,
56                      .n3 = data->prev->devices[d].n3,
57                      .data = data->prev->devices[d].data,
58                      .size = data->prev->devices[d].n1 * data->prev->
devices[d].n2 * data->prev->devices[d].n3 * sizeof(float)
59                  };
60
61                  if (useCompression) {
62                      auto comp = CompressorBuilder(gpuzip_config,
curr.n1, curr.n2, curr.n3);
63                      auto compprev = CompressorBuilder(gpuzip_config,
 prev.n1, prev.n2, prev.n3);
64                      prefetch[d]->Save(action.ts, &curr, &prev, comp.
get(), compprev.get());
65                  } else {
66                      prefetch[d]->Save(action.ts, &curr, &prev);
67                  }
68              }
69          }
70
71          if (action.actionType == ACTION_FORWARD) {
72              forward_computation(action.ts, data);
73          }
74
75          if (action.actionType == ACTION_BACKWARD) {
76              backward_computation(action.ts, data);
77          }
78
79          if (action.actionType == ACTION_RESTORE) {
80              for (int d = 0; d < num_gpus; d++) {
81                  Field_t curr = {
82                      .data = data->curr->devices[d].data,
```

```
83                          .n1 = data->curr->devices[d].n1,
84                          .n2 = data->curr->devices[d].n2,
85                          .n3 = data->curr->devices[d].n3
86                      };
87
88                      Field_t prev = {
89                          .data = data->prev->devices[d].data,
90                          .n1 = data->prev->devices[d].n1,
91                          .n2 = data->prev->devices[d].n2,
92                          .n3 = data->prev->devices[d].n3
93                      };
94
95                      if (useCompression) {
96                          auto comp = CompressorBuilder(gpuzip_config,
    curr.n1, curr.n2, curr.n3);
97                          auto compprev = CompressorBuilder(gpuzip_config,
     prev.n1, prev.n2, prev.n3);
98                          prefetch[d]->Retrieve(action.ts, &curr, &prev,
    comp.get(), compprev.get());
99                      } else {
100                         prefetch[d]->Retrieve(action.ts, &curr, &prev);
101                     }
102                 }
103             }
104         } while (!terminate);
105     }
106 }
```

Listing A.3: Example of multi-GPU adjoint computation with GPUZIP