Combining checkpointing and data compression to accelerate adjoint-based optimization problems

Anonymous Author(s)

CCS CONCEPTS

• **Mathematics of computing → Mathematical software performance**;

KEYWORDS

Checkpointing, compression, adjoints, inversion, seismic

ACM Reference format:
Anonymous Author(s). 2019. Combining checkpointing and data compression to accelerate adjoint-based optimization problems. In *Proceedings of Platform for Advanced Scientific Computing, 2019,*
, Zurich, Switzerland (PASC '19), 9 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 Adjoint-based optimization

Adjoint-based optimization problems typically consist of a simulation that is run forward in simulation time, producing data that is used in reverse order by a subsequent adjoint computation that is run backwards in simulation time. Figure 1 shows the resulting data flow. Many important numerical problems in science and engineering use adjoints and follow this pattern.

Since the data for each of the computed timesteps in the forward simulation will be used later in the adjoint computation, it would be prudent to store it in memory until it is required again, if the required amount of memory is indeed available. However, the total size of this data can often run into tens of terabytes and the management of this data becomes a problem in itself. A variety of strategies exist for this problem - some involve storing this data, sometimes with some preprocessing, while others work around the problem of storage by recomputing the discarded data instead. In this paper we present a new strategy that combines some previously used ones.

1.2 Example adjoint problem: Seismic inversion

Seismic inversion typically involves the simulation of the propagation of seismic waves through the earth's subsurface, followed by a comparison with data from field measurements. The model of the subsurface is iteratively improved by minimizing the misfit between simulated data and field measurement in an adjoint optimization problem [Plessix 2006].

PASC '19, Zurich, Switzerland,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

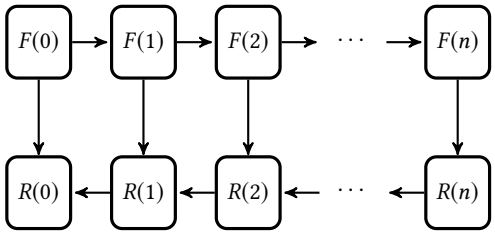

Figure 1: The data flow pattern that is typical of adjoint-based optimization problems

The data collected in an offshore survey typically consists of a number of "shots" - each of these shots corresponding to different locations of sources and receivers. As a loose analogy with machine learning, these correspond to different data points. Since the gradient computation over a single shot is complex enough that a single shot can occupy a complete node for $10^1 - 10^2$ minutes, the gradient is computed for each of these shots independently and then collated across all the shots to form a single update that is used to update the model. It might be evident here that the processing across shots is easy to parallelize since it requires a small amount of communication followed by a relatively long period of independent computation as part of a single iteration of the optimization. Since the number of shots is typically of the order of 10^4, this offers ample opportunity to fill up a large cluster with computation, even if an individual shot is only processed on a single node of the cluster at a time. Hence, it is worthy to note that even though this paper focuses on a single gradient evaluation for a single shot, the overall problem involves carrying out 10^5 such evaluations and is typically run on large clusters for non-trivial amounts of time.

The first part of seismic inversion, i.e. the simulation of seismic wave propagation through the earth's subsurface, is typically done using a finite-difference solver and is called the "forward problem" in the context of inversion. Looking at this part in isolation, the data flow here looks like the one shown in figure 2. This illustration assumes a first-order time-stepper, i.e. the computation of each timestep only depends on the previous timestep. In such a scenario, only two timesteps need to be kept in memory - the last computed step and the one currently being computed. In case of an n-th order time-stepper, $(n + 1)$ timesteps need to be kept in memory at any one time. Hence, the memory requirements of the forward problem can remain constant, regardless of the number of timesteps the simulation may be run for.

1.3 Memory requirements

A number of strategies are regularly employed to deal with this enormous volume of data - the simplest of these being to store it to a disk, to be read later by the adjoint pass in reverse order. However, typically the computation to be done on this data in the

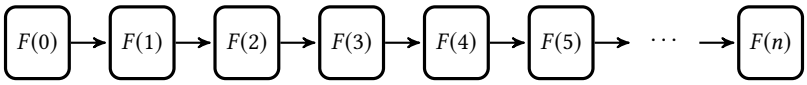

Figure 2: The dataflow pattern typical of a forward-only simulation. Boxes represent data and arrows represent computation.

adjoint phase takes much less time than the time taken to read it from the disk. Hence, reading from the disk becomes the bottleneck for most practical cases. While this may be one of the simpler strategies, it leaves a lot of room for improvement in computational performance. Seeing this from the perspective where thousands of such computations might be running in parallel on a single cluster (for different shots), the network bandwidth might restrict the use of a network storage further, hence only node-local disks may be suitable for this strategy.

Domain decomposition, where a single shot may be distributed across more than one node, is often used not only to distribute the computational workload across more processors, but also to utilize the large amount of memory available in distributed systems. While this strategy is very powerful, the number of compute nodes and therefore the amount of memory that can be used efficiently is limited, for example by communication overheads that start to dominate as the domain is split into increasingly small pieces [Virieux et al. 2009]. Secondly, this strategy can lead to a wastage of resources within the nodes, i.e. using more nodes only for their memory implies that their CPUs are incompletely utilized. At the scale of the entire inversion problem, this can sometimes even lead to a longer time-to-solution, especially when the number of nodes is less than the number of shots. For example, a problem setup that requires only 10% more memory than is available on a single node might not be a good candidate for domain decomposition over multiple nodes. Lastly, this method is even less applicable on cloud-based setups since it can be drastically more complicated to setup and slower due to the communication.

Checkpointing is yet another strategy to reduce the memory overhead. Only a subset of the timesteps during the forward pass is stored (and the rest discarded). The discarded data is recomputed when needed by restarting the forward pass from the last available stored state. We discuss this strategy in section 3.

Another strategy commonly employed to reduce the memory footprint of such applications is data compression. This is discussed in section 2.

Another common strategy in seismic inversion is to only store values at the boundaries of the domain at each timestep, and reconstruct the rest of the wavefield when required [Clapp 2009; Yang et al. 2014] with time reversal of the wave equation. However, this method is not applicable for wave equations that are not time reversible when for example physical attenuation is included.

In this paper, we extend the previous studies by *combining* checkpointing and compression. This is obviously useful when the data does not fit in the available memory even after compression, for example for very large adjoint problems, or for problems where the required accuracy limits the achievable compression ratios.

Compared to the use of only checkpointing without compression, this combined method often improves performance. This is a consequence of the reduced size of stored timesteps, allowing more timesteps to be stored during the forward computation. This in turn reduces the amount of recomputation that needs to be performed. On the other hand, the compression and decompression itself takes time. The answer to the question "does compression pay off?", depends on a number of factors including - available memory, the required precision, the time taken to compress and decompress, and the achieved compression factors, and various problem specific parameters like computational intensity of the kernel involved in the forward and adjoint computations, and the number of timesteps.

Hence, the answer to the compression question depends not only on the problem one is solving (within seismic inversion, there are numerous variations of the wave equation that may be solved), but also the hardware specifics of the machine on which it is being solved. In fact, as we will see in section 5, the answer might even change during the solution process of an individual problem. This brings up the need to be able to predict whether compression would pay off in a given scenario, without incurring significant overheads in answering this question. In this paper, we present the use of a performance model to answer that question.

1.4 Summary of contributions

In this paper, we study

- the use of different compression algorithms to seismic data including six lossless and the two most popular lossy compression algorithms for floating point data,
- a performance model for Revolve alone, taking into account the time taken to read and write checkpoints, and
- an online performance model to predict whether compression would speed up an optimization problem.

2 COMPRESSION ALGORITHMS

Data compression is increasingly used to reduce the memory footprint of scientific applications. General purpose data compression algorithms like Zlib (which is a part of gzip) [Deutsch and Gailly 1996], and compression algorithms for video and image data such as JPEG-2000 [Skodras et al. 2001] have been presented in previous work. More recently, special purpose compression algorithms for floating-point scientific data have been developed, such as ZFP or SZ [Di et al. 2018; Lindstrom 2014].

Lossless algorithms guarantee that the exact original data can be recovered during decompression, whereas lossy algorithms introduce an error, but often guarantee that the error does not exceed certain absolute or relative error metrics. Typically, lossy compression is more effective in reducing the data size. Most popular compression packages offer various settings that allow a tradeoff between compression ratio, accuracy, and compression and decompression time.

Another comonly-observed difference between lossless and lossy compression algorithms is that lossless compression algorithms

tend to interpret all data as one-dimensional series only while SZ and ZFP, being designed for scientific data, tend to take the dimensionality into account directly. This makes a difference in the case of a wavefield, for example, where the data to be compressed corresponds to a smoothly varying function in (two or) three dimensions and interpreting this three-dimensional data as one-dimensional would completely miss the smoothness and predictability of the data values.

It is worth noting that another data reduction strategy is to typecast values into a lower precision format, for example, from double precision to single precision. This can be seen as a computationally cheap lossy compression algorithm with a compression ratio of 2.

Perhaps counterintuitively, compression can not only reduce the memory footprint, but also speed up an application. Previous work has observed that the compression and decompression time can be less than the time saved from the reduction in data that needs to be communicated across MPI nodes or between a GPU and a host computer [O'Neil and Burtscher 2011].

One way of using compression in adjoint-based methods is to compress all the timesteps during the forward pass. If the compression ratio is sufficient to fit the entire data in memory, this enables solving an adjoint-based optimization problem without resorting to any of the other techniques previously discussed here. Specifically, compression serves as an *alternate strategy* to checkpointing in this scenario. Previous work has discussed this in the context of computational fluid dynamics [Cyr et al. 2015; Marin et al. 2016] and seismic inversion using compression algorithms specifically designed for the respective applications [Boehm et al. 2016; Dalmau et al. 2014].

Since the time spent on compressing and decompressing data is often non-negligible, this raises the question whether the computational time is better spent on this compression and decompression, or on the recomputation involved in the more traditional checkpointing approach. This question was previously answered to a limited extent for the above scenario where compression is an alternative to checkpointing, in a specific application [Cyr et al. 2015]. We discuss that in section 4.

2.1 Lossless

We use the python package *blosc* [blo [n. d.]], which includes implementations for six different lossless compression algorithms, namely ZLIB, ZSTD, BLOSCLZ, LZ4, LZ4HC and Snappy. All these algorithms look at the data as a one-dimensional stream of bits and at least the blosc implementations have a limit on the size of the one-dimensional array that can be compressed in one call. Therefore we use the python package *blosc-pack*, which is a wrapper over the blosc library, to implement *chunking*, i.e. breaking up the stream into chunks of a chosen size, which are compressed one at a time.

2.2 Lossy

2.2.1 ZFP. We use the lossy compression package ZFP [Lindstrom 2014] developed in C. To use ZFP from python, we developed a python wrapper for the reference implementation of ZFP [1].

ZFP supports three compression modes, namely fixed-tolerance, fixed-precision and fixed-rate. The fixed-tolerance mode limits the absolute error, while the fixed-precision mode limits the error as a ratio of the range of values in the array to be compressed. The fixed-rate mode achieves a guaranteed compression ratio requested by the user, but does not provide any bounds on accuracy loss.

2.2.2 SZ. SZ [Di et al. 2018] is a more recently developed compression library, also focussed on lossy compression of floating-point scientific data, also developed in C. We also wrote a python wrapper for the reference implementation of SZ to use it as part of our benchmark suite. [2]

SZ supports four compression modes, namely absolute error mode, which, similar to ZFP's fixed-tolerance mode, allows the user to control the maximum pointwise error in absolute values. The relative ratio mode of SZ allows the user to specify a maximum error as a ratio of the range of values in the array, which is effectively similar to ZFP's fixed-precision mode but not exactly. SZ has two other modes that are missing in ZFP, namely pointwise relative error and pointwise SNR mode. In the pointwise relative error mode, the user can provide a relative error ratio and SZ will ensure that the error at each point is within that ratio, considering its absolute value. In the pointwise SNR mode, the user provides a signal-to-noise ratio value that SZ respects at each point.

ZFP's fixed-rate mode, that guarantees compression ratios, could make an implementation quite straightforward, but with no error guarantees, it might affect the numerical properties of the problem too much. ZFP's fixed-precision mode can be compared to SZ's relative error mode although not directly. ZFP claims to achieve the best "compression efficiency" in the fixed-tolerance mode, and since this is the mode most readily comparable with SZ in its absolute error mode, we chose to do all the tests using this mode.

2.3 Combining lossy and lossless compression

Another approach we attempted is a combination of lossy and lossless compression schemes to achieve an overall lossless compression scheme. Here, the checkpoint is first compressed using a lossy compression scheme, following which the errors incurred by this lossy scheme are passed on to a lossless scheme for compression. The idea is that the distribution of errors incurred by a lossy compression algorithm might make it more favourable for lossless compression than the original array.

3 REVOLVE: PERFORMANCE MODEL

Checkpointing is a commonly used strategy to reduce the memory footprint of adjoint problems. Here, depending on the memory available, some timesteps computed in the forward pass are stored, while others are discarded. The ones that were discarded are later recomputed by rerunning the forward pass from the last stored checkpoint. The Revolve algorithm [Griewank and Walther 2000] provides an answer to the question of which timesteps should be stored and which states should be recomputed to minimize the total amount of recomputation work. Other authors have subsequently developed extensions to Revolve that are optimal under different assumptions [Aupy and Herrmann 2017; Aupy et al. 2016; Schanen

[1] To be released open source on publication

[2] Also to be released open source upon publication

et al. 2016; Stumm and Walther 2009; Wang et al. 2009]. Previous work has applied checkpointing to seismic imaging and inversion problems [Datta et al. 2018; Symes 2007].

In this section, we build on the ideas introduced in [Stumm and Walther 2009] to build a performance model that can be used to predict the runtime of an adjoint computation that uses the Revolve checkpointing strategy. We call the time taken by a single forward computational step C_F and correspondingly, the time taken by a single backward step C_R. For a simulation with $\mathbf{N}$ timesteps, the minimum wall time required for the full forward-adjoint evaluation is given by

$$T_N = \mathbf{C_F} \cdot \mathbf{N} + \mathbf{C_R} \cdot \mathbf{N} \tag{1}$$

If the size of a single timestep in memory is given by $\mathbf{S}$, this requires a memory of at least size $\mathbf{S} \cdot \mathbf{N}$. If sufficient memory is available, no checkpointing or compression is needed.

If the memory is smaller than $\mathbf{S} \cdot \mathbf{N}$, Revolve provides a strategy to solve for the adjoint field by storing a subset of the $\mathbf{N}$ total checkpoints and recompute the remaining ones. The overhead introduced by this method can be broken down into the recomputation overhead $\mathbf{O}_R$ and the storage overhead $\mathbf{O}_S$. The recomputation overhead is the amount of time spent in recomputation, given by

$$\mathbf{O}_R(N, M) = p(N, M) \cdot \mathbf{C_F}, \tag{2}$$

where $p(N, M)$ is the minimum number of recomputed steps from [Griewank and Walther 2000], reproduced here in equation 3. In equation 3, M is the number of checkpoints that can be stored in memory. Note that for $M >= N$, $\mathbf{O}_R$ would be zero. For $M < N$, $\mathbf{O}_R$ grows rapidly as M is reduced relative to N.

In an ideal implementation, the storage overhead $\mathbf{O}_S$ might be zero, since the computation could be done "in-place", but in practice, checkpoints are generally stored in a separate section of memory and they need to be transferred to a "computational" section of the memory where the computation is performed, and then the results copied back to the checkpointing memory. This copying is a common feature of checkpointing implementations, and might pose a non-trivial overhead when the computation involved in a single timestep is not very large. This storage overhead is given by:

$$\mathbf{O}_{SR}(N, M) = \mathbf{W}(N, M) \cdot \frac{\mathbf{S}}{\mathbf{B}} + \mathbf{N} \cdot \frac{\mathbf{S}}{\mathbf{B}} \tag{4}$$

where $\mathbf{W}$ is the total number of times Revolve writes checkpoints for a single run, $\mathbf{N}$ is the number of times checkpoints are read, and $\mathbf{B}$ is the bandwidth at which these copies happen. The total time to solution becomes

$$T_R = \mathbf{C_F} \cdot \mathbf{N} + \mathbf{C_R} \cdot \mathbf{N} + \mathbf{O}_R(N, M) + \mathbf{O}_{SR}(N, M) \tag{5}$$

4 PERFORMANCE MODEL INCLUDING COMPRESSION

By using compression, the size of each checkpoint is reduced and therefore the number of checkpoints available is increased (M in equation 3). This reduces the recomputation overhead $\mathbf{O}_R$, while at the same time adding overheads related to compression and decompression in $\mathbf{O}_S$. To be beneficial, the reduction in $\mathbf{O}_R$ must offset the increase in $\mathbf{O}_{SR}$, leading to an overall decrease in the time to solution T.

Our performance model assumes that the compression algorithm behaves uniformly across the different time steps of the simulation, i.e. that we get the same compression ratio, compression time and decompression time, no matter which of the N possible checkpoints we try to compress/decompress. The storage overhead now becomes

$$\begin{aligned}\mathbf{O}_{SR}(N, M) =& \mathbf{W}(N, M \cdot F) \cdot \left(\frac{\mathbf{S}}{\mathbf{F} \cdot \mathbf{B}} + t_c\right) + \\ & \mathbf{N} \cdot \left(\frac{\mathbf{S}}{\mathbf{F} \cdot \mathbf{B}} + t_d\right)\end{aligned} \tag{6}$$

where $\mathbf{F}$ is the compression ratio (i.e. the ratio between the uncompressed and compressed checkpoint), and t_c and t_d are compression and decompression times, respectively. At the same time, the recomputation overhead decreases because $\mathbf{F}$ times more checkpoints are now available.

5 ACCEPTABLE ERRORS AND CONVERGENCE

Our performance model is designed to be agnostic of the specific adjoint-based optimization problem being solved. This is because we envision its use in a generic checkpointing runtime that manages the checkpointed execution of the optimization problem that accepts an acceptable error tolerance as an input parameter for each gradient evaluation and determines whether or not compression can pay off for that iteration, and if yes, which of the available strategies is to be used. This last question has previously been addressed previously in literature but in more specific contexts [Kunkel et al. 2017; Tao et al. 2018].

One question that arises in evaluating derivatives on grids compressed (and decompressed) using lossy compression is the numerical stability of the computed derivatives, since errors in neighbouring points can accumulate in the derivative rather quickly, rendering the derivatives unusable. This question was addressed for ZFP [zfp [n. d.]] and SZ[Tao et al. 2017] separately.

In the context of seismic inversion, it has been shown before that the precision required in the gradient evaluation is very low in the beginning of the optimization and accurate gradients are not needed until the optimization is close to a minimum [Boehm et al. 2016; van Leeuwen and Herrmann 2014]. This is perhaps quite intuitive since, being far from a minimum in the beginning, a gradient pointing in the approximate direction of the relevant minimum is sufficient to make progress. These initial iterations could use a more aggressive lossy compression strategy to accelerate (through compression) the progress towards the minimum. Once the optimization is within the vicinity of the minimum, the gradient is required at a higher accuracy to make any progress, and this performance model can then dynamically decide to disable compression for those iterations where a more accurate, albeit slower, gradient evaluation is preferred.

There is also a body of work that addresses convergence guarantees of trust-region based optimization methods in the presence of unreliable gradients. This was primarily done for the scenario where the gradient (and sometimes the functional itself) is known with a probability p. [Blanchet et al. 2016; Cartis and Scheinberg 2017; Chen et al. 2018] It was shown here that the convergence rate is only affected by a factor that is a linear function of p. This analytical framework could be extended to provide bounds on the

$$p(N, M) = \begin{cases} N(N-1)/2, & \text{if } M = 1 \\ \min\limits_{1<=\widetilde{N}<=N} \{\widetilde{N} + p(\widetilde{N}, M) + p(N - \widetilde{N}, M-1)\}, & \text{if } M > 1 \end{cases} \tag{3}$$

accuracy required in a particular gradient evaluation in order to guarantee a certain convergence rate.

Both these analyses stop at the required accuracy in the gradient evaluation. This needs to be extended to derive acceptable error tolerances in individual grid points corresponding to a specific error bound in the overall gradient evaluation.

6 PROBLEM AND TEST CASE

We use Devito [Louboutin et al. 2018; Luporini et al. 2018] to solve forward and adjoint wave equation problems. Devito is a domain-specific language that enables the rapid development of finite-difference solvers from a high-level description of partial differential equations. The simplest version of the seismic wave equation is the acoustic isotropic wave equation defined as:

$$m(x)\frac{\partial^2 u(t,x)}{\partial t^2} - \nabla^2 u(t,x) = q(t,x), \tag{7}$$

where $m(x) = \frac{1}{c^2(x)}$ is the squared slowness, $c(x)$ the spatially dependent speed of sound, $u(t,x)$ is the pressure wavefield, $\nabla^2 u(t,x)$ denotes the laplacian of the wavefield and $q(t,x)$ is a source term.

The solution to equation 7 forms the forward problem. The seismic inversion problem minimizes the misfit between simulated and observed signal given by:

$$\min_m \phi_s(m) = \frac{1}{2} \left\| d_{sim} - d_{obs} \right\|_2^2 . \tag{8}$$

We call the kernel derived from a basic finite difference formulation of Equation 7, the OT2 kernel because it is second-order accurate in time. We also use another formulation from Louboutin et al. [2018], which is 4th-order accurate in time. We call this the OT4 kernel.

This optimization problem is usually solved using gradient based methods such as steepest descent, where the gradient is computed using the adjoint-state method that involves the data-flow pattern from Figure 1.

The values of $m(x)$ used in this work are derived from the Overthrust model [Aminzadeh et al. 1996] over a grid of $287 \times 881 \times 881$ points, including an absorbing layer of 40 points on each side. The grid spacing is $25m$ in space. The propagation time is $4sec$ that corresponds to 2500 timesteps. The wave field at the final time is shown in Figure 3. The uncompressed size of this single time step field is just under 900MB. If one were to store all the timesteps, this would require 2.3TB of memory.

To implement Revolve with Devito, we use pyRevolve [Kukreja et al. 2018] which is a python library to manage the execution of checkpointed adjoint computations. The performance model in section 3 assumes that the implementation is similar to pyRevolve, which stores a checkpoint by copying a portion of the operator's working memory to the checkpointing memory and similarly loads a checkpoint by copying from the checkpointing memory to the operator's working memory. Although a perfect implementation of checkpointing may be able to avoid these copies, the overhead attached to these copies can be ignored for an operator that is sufficiently computationally intensive. However, we include the overheads in the model to verify this assumption.

For benchmarking we used a dual-socket Intel(R) Xeon(R) Platinum 8180M @ 2.50 Ghz (28 cores each) (skylake).

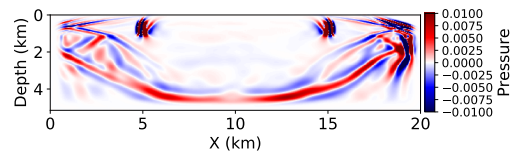

Figure 3: Cross-section of the wavefield used as a reference sample for compression and decompression. This field was formed after a Ricker wavelet source was placed at the surface of the model and the wave propagated for 2500 timesteps. This is a vertical (x-z) cross-section of a 3D field, taken at the y source location

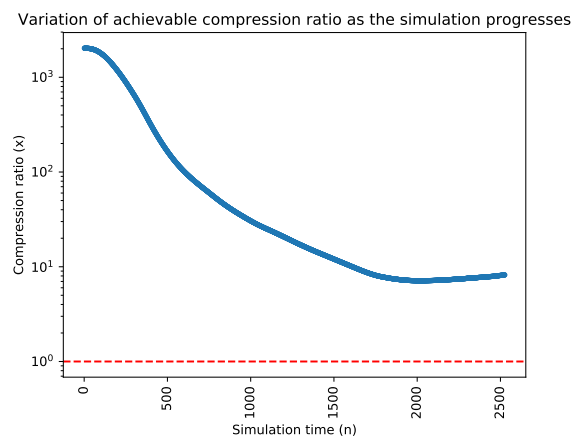

Figure 4: Compression ratios achieved on compressing different time steps. Every timestep from 1 to 2526 was compressed and plotted.

7 RESULTS AND DISCUSSION

7.1 Evolution of compressibility

To understand the compressibility of the data produced in a typical wave-propagation simulation, we ran a simulation as per the setup described in section 6, and tried to compress every single timestep. For this we chose ZFP in fixed tolerance mode at some arbitrary tolerance level. We noted the compression ratios achieved at every timestep. As figure 4 shows, the initial timesteps are much easier

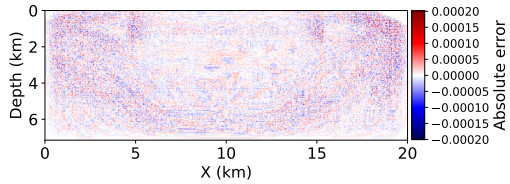

Figure 5: Cross-section of the field that shows errors introduced during compression and decompression using the fixed-tolerance mode. It is interesting to note that the errors are more or less evenly distributed across the domain with only slight variations corresponding to the wave amplitude (from the field plot in Figure 3. A small block-like structure characteristic of ZFP can be seen.

to compress than the later ones. This is not surprising since most wave simulations start with the field at rest, i.e. filled with zeros. As the wave reaches more parts of the domain, the field becomes less compressible until it achieves a stable state when the wave has reached most of the domain.

If the simulation had started with the field already oscillating in a wave, it is likely that the compressibility curve for that simulation would be flat.

This tells us that the compressibility of the last timestep of the solution is representative of the worst-case compressibility and hence we used the last timestep as our reference for comparison of compression in the following sections.

7.2 Lossless compression

Table 1 shows the compression ratios and times for a few different lossless compressors and their corresponding settings. As can be seen, the compression factors achieved, and the time taken to compress and decompress can vary significantly, but it is hard to say whether this compression could be used to speed up the inversion problem.

7.3 Lossy Compression

Figure 6 shows compression ratios for different tolerance settings for the fixed-tolerance mode of ZFP. The point highlighted here was the setting used to compress all timesteps in section 7.1. Figure 5 shows the spatial distribution of the errors after compression and decompression, compared to the original field, for this setting.

7.4 Performance Model

Using the results so far, it was evident that the use of compression can sometimes speed up an adjoint problem and may sometimes slow it down. For the generic and effective use of compression in adjoint problems, we need to be able to predict and choose between various kinds of compression and their settings, or even to switch off compression when appropriate. For this purpose, the performance model in section 4 was developed.

To study the performance model, we first visualize it along the axis of available memory, comparing the predicted performance of the chosen compression scheme with the predicted performance of a Revolve-only adjoint implementation. This is shown in Figure 7

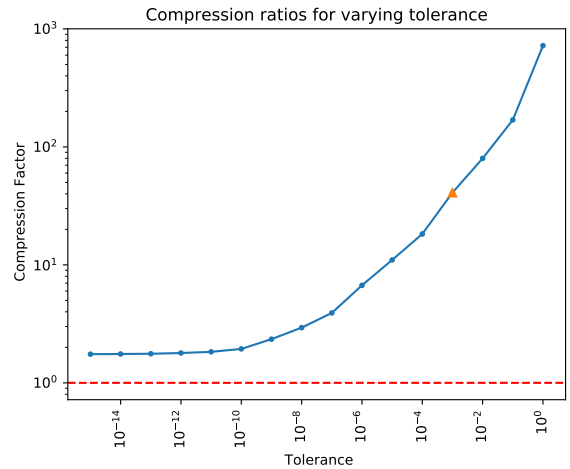

Figure 6: Compression ratios achieved on compressing the wavefield. We define compression ratio as the ratio between the size of the uncompressed data and the compressed data. The dashed line represents no compression. The highlighted point corresponds to the setting used for the other results here unless otherwise specified.

where we can distinguish three different scenarios, depending on the amount of available memory.

(1) If the memory is insufficient even with compression to store the entire trajectory, one can either use checkpointing only, or combine checkpointing with compression. This is the left section of the figure.
(2) If the available memory is not sufficient to store the uncompressed trajectory, but large enough to store the entire compressed trajectory, we compare two possible strategies: Either use compression only, or use checkpointing only. This is the middle section of the figure.
(3) If the available system memory is large enough to hold the entire uncompressed trajectory, neither compression nor checkpointing is necessary. This is the right section of the figure.

The second scenario was studied in previous work [Cyr et al. 2015], while the combined method is also applicable to the first scenario, for which previous work has only used checkpointing without compression.

We can identify a number of factors that make compression more likely to be beneficial compared to pure checkpointing: A very small system memory size and a large number of time steps lead to a rapidly increasing recompute factor, and compression can substantially reduce this recompute factor. This can be seen in Figures 7 and 9.

The extent to which the recompute factor affects the overall runtime also depends on the cost to compute each individual time step. If the compute cost per time step is large compared to the compression and decompression cost, then compression is also likely to be beneficial, as shown in Figure 8. As the time per time step increases and the compression cost becomes negligible, we

Compressor	Chunk size(bytes)	Shuffle Mode	Setting	Compression time(ms)	Decompression time(ms)	Compression Ratio
BloscLZ	1048576	SHUFFLE	6	4249.44	1288.86	1.188
LZ4	2965280	SHUFFLE	4	1371.26	920.98	1.199
LZ4HC	2097152	SHUFFLE	8	31245.16	926.69	1.265
ZLib	524288	SHUFFLE	7	30218.81	2470.04	1.291
ZStd	524288	SHUFFLE	9	117238.76	1477.34	1.312

Table 1: Some results from trying out all possible compressors and settings in blosc. We selected the best compression ratio seen for each compressor. "Setting" here is the choice between speed and compression, where 0 is fastest and 9 is highest compression.

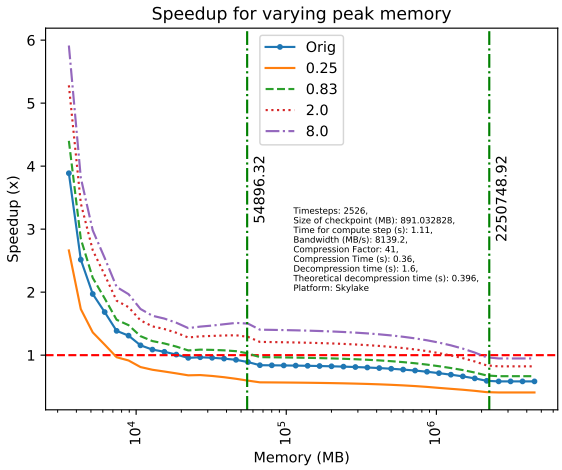

Figure 7: The speedups predicted by the performance model for varying memory. The baseline (1.0) is the performance of a Revolve-only implementation under the same conditions. The different curves represent kernels with differing compute times (represented here as a factor of the sum of compression and decompression times). The first vertical line at 53GB marks the spot where the compressed wavefield can completely fit in memory and Revolve is unnecessary if using compression. The second vertical line at 2.2 TB marks the spot where the entire uncompressed wavefield can fit in memory and neither Revolve nor compression is necessary. The region to the right is where these optimizations are not necessary or relevant. The middle region has been the subject of past studies using compression in adjoint problems. The region to the left is the focus of this paper.

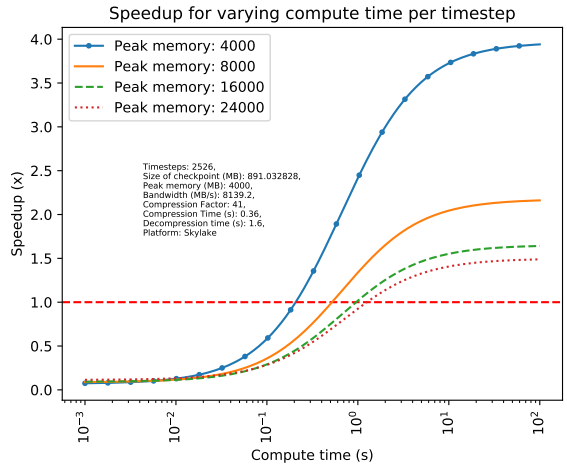

Figure 8: The speedups predicted by the performance model for varying compute cost. The baseline (1.0) is the performance of a Revolve-only implementation under the same conditions. The benefits of compression drop rapidly if the computational cost of the kernel that generated the data is much lower than the cost of compressing the data. For increasing computational costs, the benefits are bounded.

observe that the ratio between the runtime of the combined method and that of pure checkpointing is only determined by the difference in recompute factors.

7.5 Validation of model

To validate the revolve-only performance model, figure 10 shows the predicted runtime for a variety of peak memory constraints along with measured runtime for the same scenario. Next, we test the performance model with compression. Figure 11 shows a comparison of predicted and measured runtimes for the OT2 kernel, which has a relatively lower computational complexity.

Figure 12 repeats this experiment for the OT4 kernel which has a higher computational complexity. It can be seen that compression pays off more when the computational complexity of the kernel is higher, as can be expected.

7.6 Accumulation of errors

Table 2 shows the effect of different levels of pointwise absolute error on the overall error in the gradient evaluation.

8 CONCLUSIONS AND FUTURE WORK

We use lossless and lossy compression to reduce the computational overhead of checkpointing in an adjoint computation used in seismic inversion, a common method in seismic imaging applications whose memory footprint commonly exceeds the available memory size in high performance computing systems. We saw that the compression ratios achieved and the time taken to compress and decompress can vary a lot based on the choice of compressor and whether and how much error it is possible to tolerate in the gradient

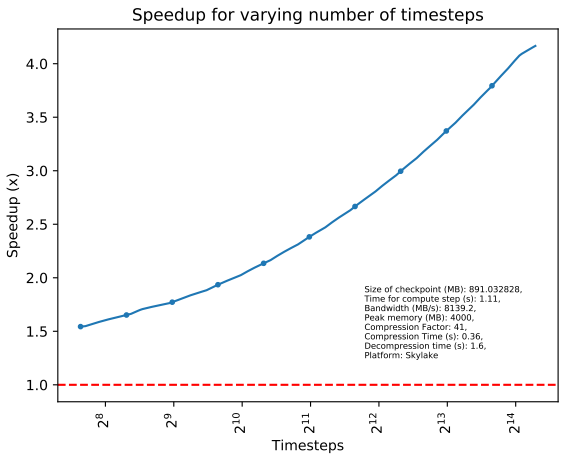

Figure 9: The speedups predicted by the performance model for varying number of timesteps to be reversed. The baseline (1.0) is the performance of a Revolve-only implementation under the same conditions. It can be seen that compression becomes more beneficial as the number of timesteps is increased.

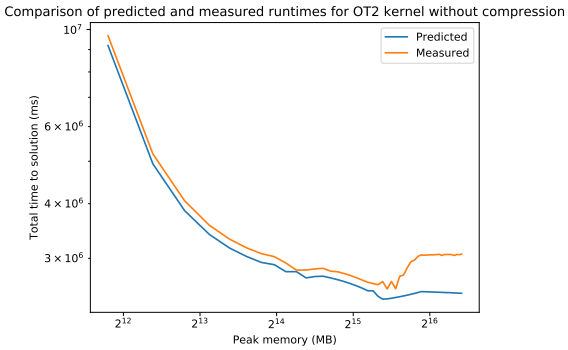

Figure 10: Predicted vs measured runtimes for OT2 kernel and no compression. This tests the performance model for checkpointing alone

evaluation. We saw that it is possible to compute the gradient after lossy checkpointing without affecting the stability of the numerical algorithm. We also saw that the tolerable error in the gradient evaluation can vary from iteration to iteration within the same optimization problem. Hence, the question, "Will compression speed up the gradient evaluation?" can not be answered at a general level and has to be evaluated at the problem and iteration level. To this end, we developed a performance model that computes whether or not the combination of compression and checkpointing will outperform pure checkpointing or pure compression in a variety of scenarios, depending on the available memory size, computational intensity of the application, and compression ratio and throughput of the compression algorithm. In an ideal implementation, the evaluation of various compressors and settings would be distributed across

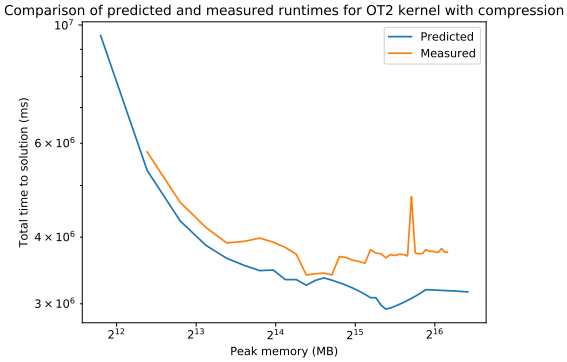

Figure 11: Predicted vs measured runtimes for OT2 kernel and ZFP compression enabled with absolute error tolerance set to 10^{-6}. This tests the performance model for checkpointing combined with compression

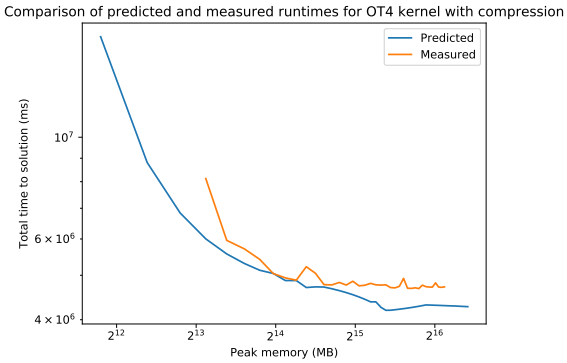

Figure 12: Predicted vs measured runtimes for OT4 kernel and ZFP compression enabled with absolute error tolerance set to 10^{-6}.

Absolute error setting	Gradient error
0.1	662.905
0.01	70.619
0.001	10.485
0.0001	0.763
10^{-5}	0.194
10^{-6}	0.154
10^{-7}	0.151

Table 2: The effect of pointwise checkpoint-(de)compression errors on the overall gradient computation errors. Absolute error was set in ZFP using the fixed-tolerance mode and the gradient error is the 2-norm of the error tensor in the gradient, as compared with an exact computation.

nodes, while doing the first few shots/iterations and the results collated to arrive at a central decision about whether to enable compression for successive shots/iterations. Our current result has several limitations that we plan to address in future work:

- The discussion in section 5 can be extended with an analysis that can derive pointwise bounds on checkpoint compression, given an error bound on the gradient evaluation.
- Our performance model is based on uniform compression ratios and times. However, many applications, including seismic inversion, are likely to have initial conditions that contain little information and are easily compressed, and the compression ratio gradually declines as the field becomes more complex. We based our experiments on the final wave field, which is presumably difficult to compress.
- In comparing pure compression with pure checkpointing, we assume that every checkpoint is compressed and decompressed. However, if the available memory is only slightly less than the required memory, an implementation that compresses only a subset of the checkpoints might outperform the expectations of our model.
- We do not discuss multi-level checkpointing, where some checkpoints are stored on a slower, larger device. We expect compression to be beneficial in these scenarios due to reduced data transfer sizes.
- To fully take advantage of checkpoint compression, the checkpointing scheduling algorithm needs to be aware of the size of each checkpoint after compression, since the compression ratio is different for each timestep. Since Revolve assumes uniform checkpoint sizes, it needs to be extended for the case where the checkpoint size is non-uniform and only known after compression.

REFERENCES

[n. d.]. ([n. d.]). https://computation.llnl.gov/projects/floating-point-compression/zfp-and-derivatives

[n. d.]. Blosc (http://blosc.org). ([n. d.]). http://blosc.org

Fred Aminzadeh, N Burkhard, J Long, Tim Kunz, and P Duclos. 1996. Three dimensional SEG/EAGE models—An update. *The Leading Edge* 15, 2 (1996), 131–134.

Guillaume Aupy and Julien Herrmann. 2017. Periodicity in optimal hierarchical checkpointing schemes for adjoint computations. *Optimization Methods and Software* 32, 3 (2017), 594–624.

Guillaume Aupy, Julien Herrmann, Paul Hovland, and Yves Robert. 2016. Optimal multistage algorithm for adjoint computation. *SIAM Journal on Scientific Computing* 38, 3 (2016), C232–C255.

Jose Blanchet, Coralia Cartis, Matt Menickelly, and Katya Scheinberg. 2016. Convergence rate analysis of a stochastic trust region method for nonconvex optimization. *arXiv preprint arXiv:1609.07428* (2016).

Christian Boehm, Mauricio Hanzich, Josep de la Puente, and Andreas Fichtner. 2016. Wavefield compression for adjoint methods in full-waveform inversion. *Geophysics* 81, 6 (2016), R385–R397.

Coralia Cartis and Katya Scheinberg. 2017. Global convergence rate analysis of unconstrained optimization methods based on probabilistic models. *Mathematical Programming* (2017), 1–39.

Ruobing Chen, Matt Menickelly, and Katya Scheinberg. 2018. Stochastic optimization using a trust-region method and random models. *Mathematical Programming* 169, 2 (2018), 447–487.

Robert G Clapp. 2009. Reverse time migration with random boundaries. In *Seg technical program expanded abstracts 2009*. Society of Exploration Geophysicists, 2809–2813.

ERIC C Cyr, JN Shadid, and T Wildey. 2015. Towards efficient backward-in-time adjoint computations using data compression techniques. *Computer Methods in Applied Mechanics and Engineering* 288 (2015), 24–44.

F Rubio Dalmau, M Hanzich, J de la Puente, and N Gutiérrez. 2014. Lossy data compression with DCT transforms. In *EAGE Workshop on High Performance Computing for Upstream*.

Debanjan Datta, David Appelhans, Constantinos Evangelinos, and Kirk Jordan. 2018. An Asynchronous Two-Level Checkpointing Method to Solve Adjoint Problems on Hierarchical Memory Spaces. *Computing in Science & Engineering* 20, 4 (2018), 39–55.

Peter Deutsch and Jean-Loup Gailly. 1996. *Zlib compressed data format specification version 3.3*. Technical Report.

Sheng Di, Dingwen Tao, Xin Liang, and Franck Cappello. 2018. Efficient Lossy Compression for Scientific Data based on Pointwise Relative Error Bound. *IEEE Transactions on Parallel and Distributed Systems* (2018).

Andreas Griewank and Andrea Walther. 2000. Algorithm 799: Revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)* 26, 1 (2000), 19–45.

Navjot Kukreja, Jan Hückelheim, Michael Lange, Mathias Louboutin, Andrea Walther, Simon W Funke, and Gerard Gorman. 2018. High-level python abstractions for optimal checkpointing in inversion problems. *arXiv preprint arXiv:1802.02474* (2018).

Julian Kunkel, Anastasiia Novikova, Eugen Betke, and Armin Schaare. 2017. Toward decoupling the selection of compression algorithms from quality constraints. In *International Conference on High Performance Computing*. Springer, 3–14.

Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2674–2683.

M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman. 2018. Devito: an embedded domain-specific language for finite differences and geophysical exploration. *CoRR* abs/1808.01995 (Aug 2018). arXiv:1808.01995 https://arxiv.org/abs/1808.01995

F. Luporini, M. Lange, M. Louboutin, N. Kukreja, J. Hückelheim, C. Yount, P. Witte, P. H. J. Kelly, G. J. Gorman, and F. J. Herrmann. 2018. Architecture and performance of Devito, a system for automated stencil computation. *CoRR* abs/1807.03032 (jul 2018). arXiv:1807.03032 http://arxiv.org/abs/1807.03032

OANA Marin, MICHEL Schanen, and PF Fischer. 2016. *Large-scale lossy data compression based on an a priori error estimator in a spectral element code*. Technical Report. ANL/MCS-P6024-0616.

Molly A. O'Neil and Martin Burtscher. 2011. Floating-point Data Compression at 75 Gb/s on a GPU. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM. https://doi.org/10.1145/1964179.1964189

R-E Plessix. 2006. A review of the adjoint-state method for computing the gradient of a functional with geophysical applications. *Geophysical Journal International* 167, 2 (2006), 495–503.

Michel Schanen, Oana Marin, Hong Zhang, and Mihai Anitescu. 2016. Asynchronous Two-level Checkpointing Scheme for Large-scale Adjoints in the Spectral-element Solver Nek5000.. In *ICCS*. 1147–1158.

Athanassios Skodras, Charilaos Christopoulos, and Touradj Ebrahimi. 2001. The JPEG 2000 still image compression standard. *IEEE Signal processing magazine* 18, 5 (2001), 36–58.

Philipp Stumm and Andrea Walther. 2009. Multistage approaches for optimal offline checkpointing. *SIAM Journal on Scientific Computing* 31, 3 (2009), 1946–1967.

William W Symes. 2007. Reverse time migration with optimal checkpointing. *Geophysics* 72, 5 (2007), SM213–SM221.

Dingwen Tao, Sheng Di, Hanqi Guo, Zizhong Chen, and Franck Cappello. 2017. Z-checker: A framework for assessing lossy compression of scientific data. *The International Journal of High Performance Computing Applications* (2017), 1094342017737147.

Dingwen Tao, Sheng Di, Xin Liang, Zizhong Chen, and Franck Cappello. 2018. Optimizing Lossy Compression Rate-Distortion from Automatic Online Selection between SZ and ZFP. *arXiv preprint arXiv:1806.08901* (2018).

Tristan van Leeuwen and Felix J Herrmann. 2014. 3D frequency-domain seismic inversion with controlled sloppiness. *SIAM journal on scientific computing* 36, 5 (2014), S192–S217.

Jean Virieux, Stéphane Operto, Hafedh Ben-Hadj-Ali, Romain Brossier, Vincent Etienne, Florent Sourbier, Luc Giraud, and Azzam Haidar. 2009. Seismic wave modeling for seismic imaging. *The Leading Edge* 28, 5 (2009), 538–544.

Qiqi Wang, Parviz Moin, and Gianluca Iaccarino. 2009. Minimal repetition dynamic checkpointing algorithm for unsteady adjoint calculation. *SIAM Journal on Scientific Computing* 31, 4 (2009), 2549–2567.

Pengliang Yang, Jinghuai Gao, and Baoli Wang. 2014. RTM using effective boundary saving: A staggered grid GPU implementation. *Computers & Geosciences* 68 (2014), 64–72.