Multi-parameter waveform inversion on GPUs: A pipeline approach

Huy Le, Robert G. Clapp, and Stewart A. Levin

ABSTRACT

We adopt a pipeline approach to accelerate 3D time-domain finite-difference waveform inversion codes using graphics cards (GPUs), without having to do domain decomposition. The key designs include streaming through the volume one block at a time and propagating this block as many time steps as possible while it is on the device. This approach allows us to process an arbitrarily large volume with a single GPU, which is particularly suitable in a cloud environment where fast inter-nodal connection is not guaranteed. Moreover, two parameters, block size and number of updates, give developers flexibility to adapt to available resources at hand. The most significant advantage that the pipeline approach offers, in our opinion, is the ability to compute subsurface offset gathers on GPUs. In this paper we describe our implementation on the pseudo-acoustic anisotropic wave equations and show that the pipeline technique achieves nearly linear scaling with number of GPUs.

INTRODUCTION

The majority of GPU implementations of seismic modeling and migration codes are based on the conventional domain decomposition technique when memory requirement exceeds GPU's limit. This technique, though relatively easy to implement, put great demands on hardware resources, especially when the industry is now moving toward multi-parameter settings such as elasticity and anisotropy. Extended domains used in waveform inversion methods such as Wave Equation Migration Velocity Analysis (WEMVA) stress this problem even more.

Johnsen and Loddoch (2014) present a particularly interesting approach that can overcome the problem of GPU memory limit. In their design, the wave propagation process is implemented as a pipeline that streams through the computational volume. By dividing this volume into small blocks and taking advantage of the finite difference stencil structure to update these blocks as many time steps as GPU memory allows, the host-device communication can be completely overlapped with computation. This makes it possible for one single GPU to process an arbitrarily large volume. In a way, the pipeline approach can be considered as an out-of-core algorithm in which CPU memory is used as bulk memory.

Outside of GPU context, similar algorithms have been applied to array processors and vector computers when CPU memory was not enough for large-scale problems. Levin (1993) introduces this algorithm in solving Laplace's equation. Graves (1996) uses it in simulations of earthquakes. Etgen and O'Brien (2007) apply to 3D acoustic finite difference modeling. Here we implement the pipeline algorithm to the pseudo-acoustic anisotropic wave equations and determine that it not only scales well with number of GPUs but also allows us to compute extended images on device.

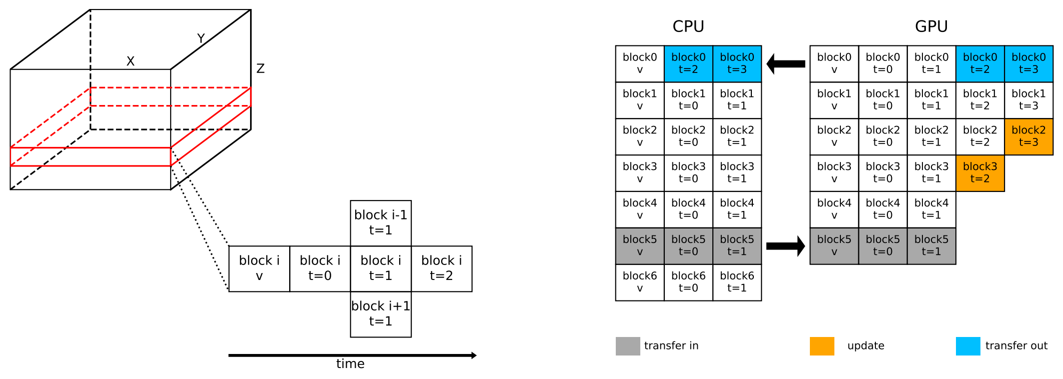
PIPELINE ALGORITHM

Single GPU implementation

Figure 1a depicts how a computational volume is divided into small blocks along an axis (z in this case). The number of depth slices in each block is at least equal to half of the stencil length so that three consecutive blocks contain all necessary data to compute spatial derivatives in the middle block. For finite difference schemes that are second-order in time, updating this middle block also requires a velocity block and a block of wavefield at the previous time step.

Figure 1b schematically shows how the pipeline algorithms works. At each iteration of the pipeline, one velocity block and two wavefields blocks at two consecutive time steps are transfered from CPU to GPU. The compute kernel updates the wavefield blocks as many time steps as possible using the blocks that already exist on GPU. This kernel is implemented in a similar fashion to Micikevicius (2009) using a 2D front of thread blocks that advances throuh depth slices to process derivatives in horizontal directions while derivative in the vertical direction is handled by an array of registers local to each thread. Once GPU memory is exhausted, the wavefield blocks at the two most current time steps are tranfered back to CPU. When it reaches the bottom blocks, the pipeline continuously feeds in the updated blocks for another round of time steppping. Note that spatial derivatives of the top and bottom blocks require data that is outside of the computational domain, where we assume zeros.

Figure 4a shows a performance comparison of pipeline approach and a CPU code on the scalar acoustic wave equation with second order in time and 8$^{\text{th}}$ order in space. The computational volume is $1000 \times 1000 \times 500$. The optimal performance is measured when the whole volume fits in a single K80 GPU (12 GB memory) so that no domain decomposition or host-device transfer is needed. The CPU code is optimized by explicit blocking, parallized with Intel Thread Building Blocks (TBB) library, and vectorized with Intel SIMD Program Compiler (ISPC). We observe that the pipeline algorithm achieves more than double the performance of the CPU code and is very close to the optimal performance. The reason for sub-optimal performance, besides overheads in host-device communication, is the fact that the pipeline takes a finite number of iterations to initialize and drain. The total number of iterations is $\frac{\text{NT}}{\text{NUPDATE}} \times \text{NBLOCK}$, where NT is the number of time steps, NUPDATE is the number of updates per host-device transfer, and NBLOCK $= \frac{\text{NZ}}{\text{BLOCK\_SIZE}}$ is the number of blocks. In our implementation, it takes NUPDATE + 5 iterations to initialize and drain. Whether the initialization and drainage times are negligible depends on volume's size, NZ, and number of time steps, NT.



(a) (b)

Figure 1: (a) Division of the computational volume in blocks of half-stencil-length size. (b) Pipeline algorithm works by streaming through the volume block by block and update as many time steps as possible. Two updates are shown in this figure. [**NR**]

Parameter tuning

Two adjustable parameters are the number of depth slices per block, BLOCK_SIZE, and number of update, NUPDATE. The number of depth slices per block has to be at least half the length of the stencil, BLOCK_SIZE $\geq \frac{k}{2}$, where k is the order. Increasing BLOCK_SIZE reduces the redundancy in the computation of vertical derivative and the host-device transfer overheads, at the expense of GPU memory. The redundancy is the ratio between the number of grid points accessed and the number of grid points processed, $\frac{\text{k+BLOCK\_SIZE}}{\text{BLOCK\_SIZE}}$. With k = 8, the redundancy is 3, 2, and 1.5 respectively for BLOCK_SIZE = 4, 8, 16.

The number of updates, NUPDATE, is bounded below by the cost of host-device transferring and is bounded above by GPU memory. We have implemented the pipeline algorithm for the pseudo-acoustic wave equations and experimented with diferent numbers of updates. Table 1 shows GPU memory usage. Figure 4b shows the performance result. The algorithm's performance improves as NUPDATE increases and approaches an asymptote after 8 updates. This is when the compute time completely overlaps host-device IO.

Number of updates	GPU Memory (GBs)
2	0.736
4	1.024
8	1.6
16	2.752
32	5.056
64	9.664

Table 1: GPU Memory for the pseudo-acoustic wave equations for different number of updates.

Multiple GPU implementation

The easiest and most efficient way to extend the algorithm to multiple GPUs is to replicate it on all devices, in which the output wavefield and velocity blocks of one device are transfered to the next device for more updates. Figure 2 sketches an implementation on two GPUs. Note the additional transfer between GPUs. Now the total number of iterations is $\frac{\text{NT}}{\text{NUPDATE}\times\text{NGPU}} \times \text{NBLOCK}$ and the pipeline takes more iterations to initialize and drain, $\text{NGPU} \times (\text{NUPDATE} + 5)$. Figure 4c shows that the pipeline algorithm achieves nearly linear scaling with number of GPUs.

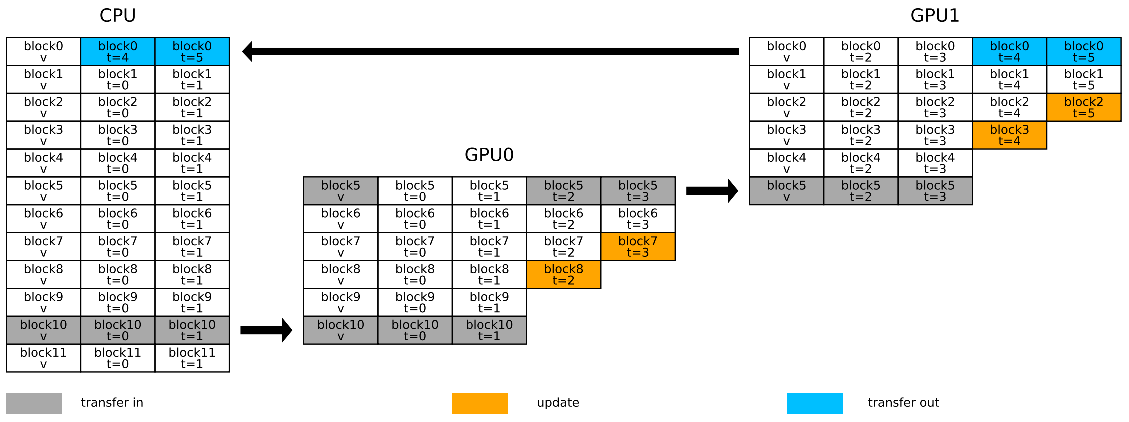
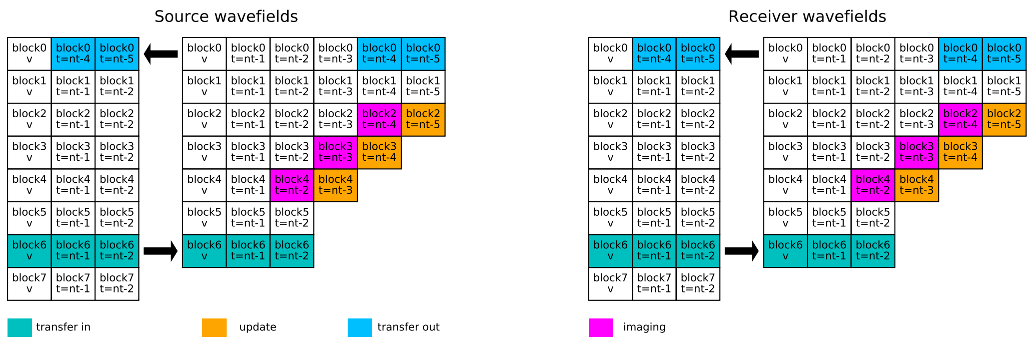


Figure 2: Pipeline algorithm for 2 GPUs. [**NR**]

Application to imaging and inversion

At the heart of any waveform inversion method is the computation of the objective function's gradients, which is the zero-lag cross-correlation of the source wavefields and the receiver wavefields. To avoid storing a 4D wavefield, we adopt random boundary condition (Shen and Clapp, 2015) by doing one extra propagation. As a result, the source and receiver wavefields can be propagated simutaneously on two pipelines and the gradients are formed on the fly (Figure 3).

The biggest benefit of the pipeline approach is in the computation of extended images. Due to the need for huge memory storage for these images, it is almost imposible for conventional domain decomposition methods to perform the extended imaging condition on device. Previous numerical experiments show that it takes 8 updates to overcome host-device IO, which means 1.6 GB of GPU memory for one propagation pipeline (i.e. 3.2 GB for both source and receiver wavefields). The K80 GPU is equiped with 12 GB memory. This opens the possibility to compute extended images on GPUs. In fact, our implementation with 32 subsurface offset lags for the pseudo-acoustic anisotropic wave equations requires 8 GB of GPU memory. Note that even though the host-device communication cost increases significantly to accommodate these extended images, the number of updates, NUPDATE, need not increase because each update now takes more time performing propagations of source and receiver wavefields and imaging condition.



(a) (b)

Figure 3: Pipeline algorithm for computing the gradients: (a) source wavefield pipeline and (b) receiver wavefield pipeline. [**NR**]

CONCLUSIONS

We have implemented a pipeline algorithm for 3D time-domain finite-difference waveform inversion on GPUs and showed that this algorithm scales well with number of GPUs. This algorithm also allows us to process a large volume using a small amount of memory, which makes possible to compute extended images on GPUs. Furthermore, the pipeline approach is advantageous to conventional domain decomposition techniques in cloud environments where inter-nodal connection might be slow.

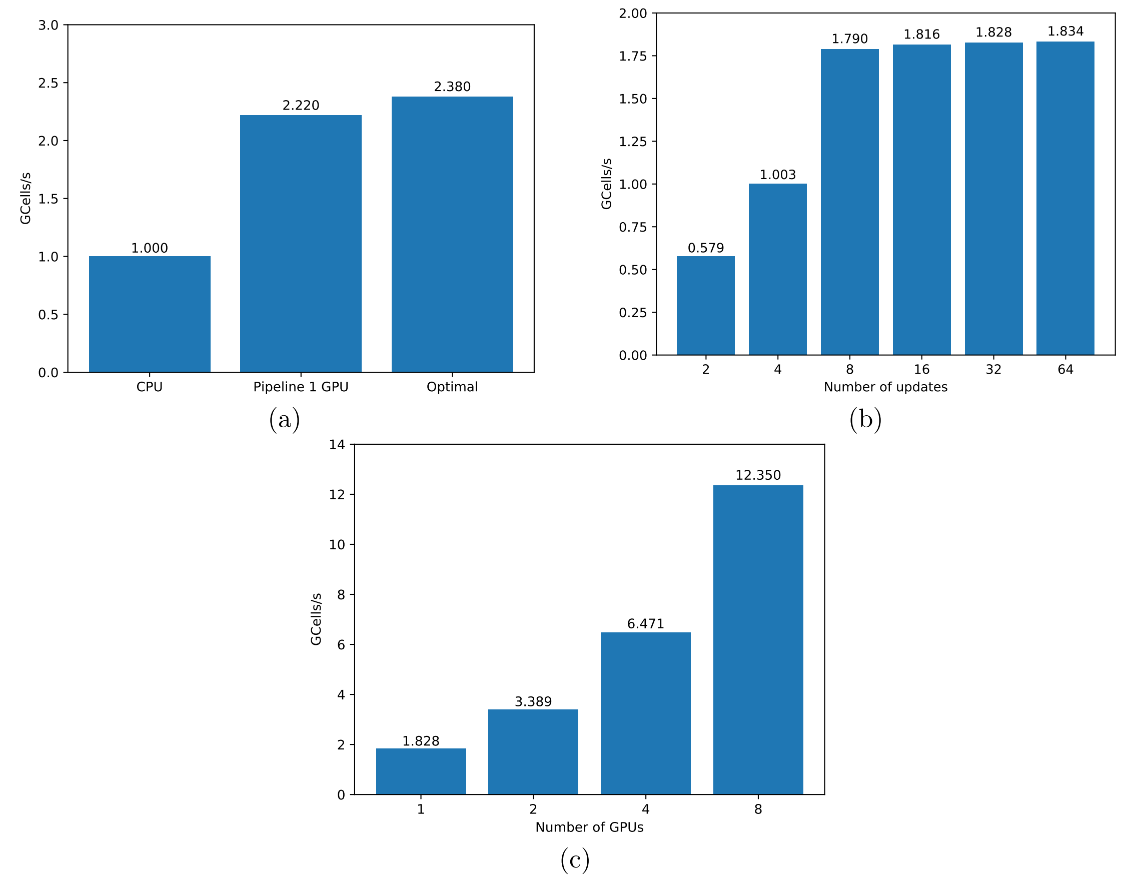


Figure 4: Performance reports. (a) Comparison of CPU, pipeline, and optimal codes for isotropic scalar wave equation. (b) Performance of pipeline algorithm with different numbers of update for 1 GPU. (c) Performance of pipeline algorithm with different numbers of GPUs. [**CR**]

REFERENCES

Etgen, J. T. and M. J. O'Brien, 2007, Computational methods for large-scale 3D acoustic finite-difference modeling: A tutorial: Geophysics, **72**, SM223–SM230.

Graves, R. W., 1996, Simulating seismic wave propagation in 3D elastic media using staggered-grid finite differences: Bulletin of the Seismological Society of America, **86**, 1091–1106.

Johnsen, T. and A. Loddoch, 2014, High frequency elastic seismic modeling on GPUs without domain decomposition: Presented at the GPU Technology Conference.

Levin, S. A., 1993, High performance computing: Scientific Computing and Computational Mathematics (SCCM) 240 Lecture Notes: Stanford University.

Micikevicius, P., 2009, 3D finite difference computation on GPUs using CUDA: The 2nd Workshop on General Purpose Processing on Graphics Processing Units, Expanded Abstracts, 79–84.

Shen, X. and R. G. Clapp, 2015, Random boundary condition for memory-efficient waveform inversion gradient computation: Geophysics, **80**, R351–R359.