Chapter 6

GPU Implementation and Adaptation

The majority of Graphics Processing Unit (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, puts great demands on hardware resources, especially when the industry is now moving toward multi-parameter imaging 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 using as many time steps as GPU memory allows, the host-device communication can be completely overlapped by 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 and slow 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 it to 3D acoustic finite difference modeling.

I adopt a pipeline approach to accelerate 3D time-domain finite-difference waveform inversion codes using graphics cards (GPUs), thus avoiding 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 users flexibility to adapt to available resources at hand. The most significant advantage that the pipeline approach offers is the ability to compute subsurface offset gathers on GPUs. In this chapter I describe my implementation on the pseudo-acoustic anisotropic wave equations and show that the pipeline technique achieves nearly linear scaling with number of GPUs.

PIPELINE ALGORITHM

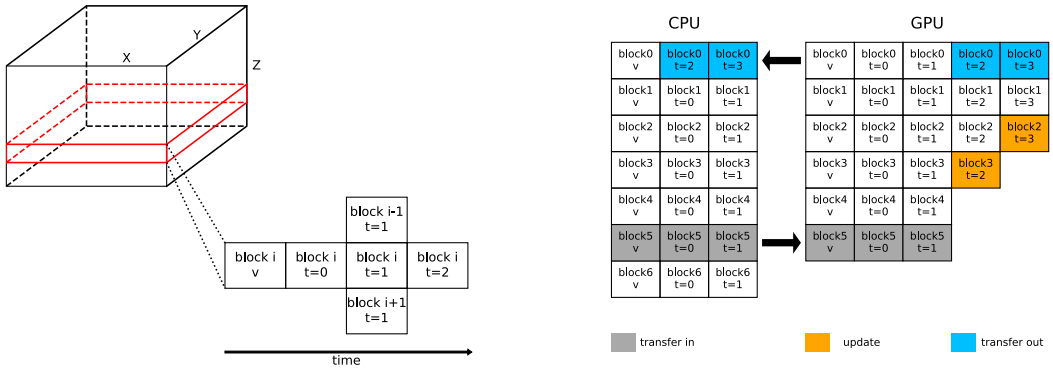
Single GPU implementation

Figure 6.1(a) 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 6.1(b) 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 through 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 filled up, the wavefield blocks at the two most current time steps are transfered 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 I assume zeros.

Figure 6.4(a) shows a performance comparison of this pipeline approach and a CPU code on the scalar acoustic wave equation with second order in time and 8^{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, parallelized with Intel Thread Building Blocks (TBB) library, and vectorized with Intel SIMD Program Compiler (ISPC). I 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 $\text{NBLOCK} = \frac{\text{NZ}}{\text{BLOCK\_SIZE}}$ is the number of blocks. In my 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 6.1: (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 stencil's 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, $\frac{k+BLOCK\_SIZE}{BLOCK\_SIZE}$. With k = 8, the redundancy is 3, 2, and 1.5 respectively for BLOCK_SIZE $= 4, 8,$ and 16. For this reason, I almost always choose BLOCK_SIZE = 4.

The number of updates, NUPDATE, is bounded below by the cost of host-device transferring and is bounded above by GPU memory. I have implemented the pipeline algorithm for the pseudo-acoustic anisotropic wave equations and experimented with diferent numbers of updates. Table 6.1 shows GPU memory usage. Figure 6.4(b) shows the performance result. The algorithm's performance improves as NUPDATE increases and approaches an asymptote after 8 updates. This is when the compute

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

time completely overlaps host-device IO.

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 6.2 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 6.4(c) shows that the pipeline algorithm achieves nearly linear scaling with number of GPUs.

In multiple GPU implementation, the number of GPUs, NGPU, is another tuning parameter. Except in initialization and drainage, updates on different GPUs are performed in parallel while updates in one GPU have to be sequential. One, therefore, would like to extend the pipeline to as many GPUs as possible. On the other hand, one also would like to have more blocks than the pipeline's length, $\text{NBLOCK} \geq \text{NGPU} \times \text{NUPDATE}$, to keep all GPUs running at maximum capacity at all time. In industry-scale applications, this is usually the case, for NZ being very large. For small to medium-sized models or when NUPDATE needs to be large to overcome host-device transfer, this, however, might not be true. In that case, one might not want to extend the pipeline to all available GPUs, but break them up into groups

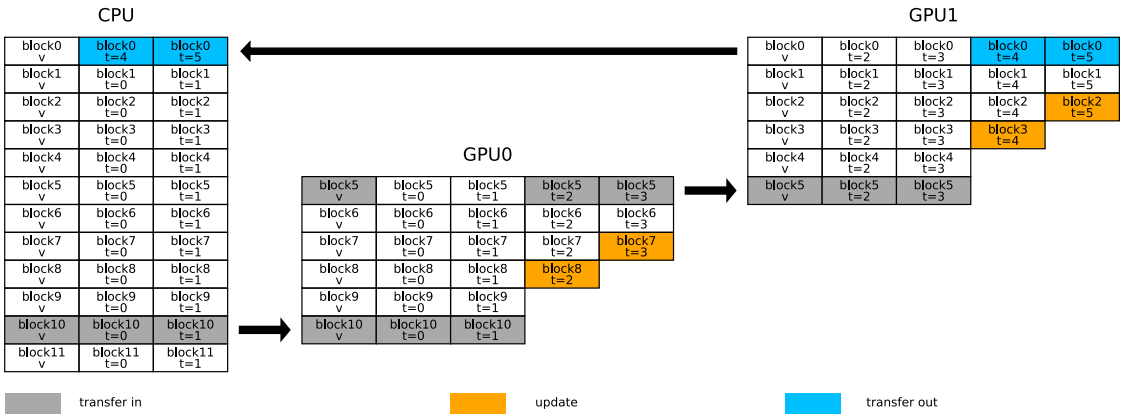


Figure 6.2: Pipeline algorithm for 2 GPUs. [**NR**]

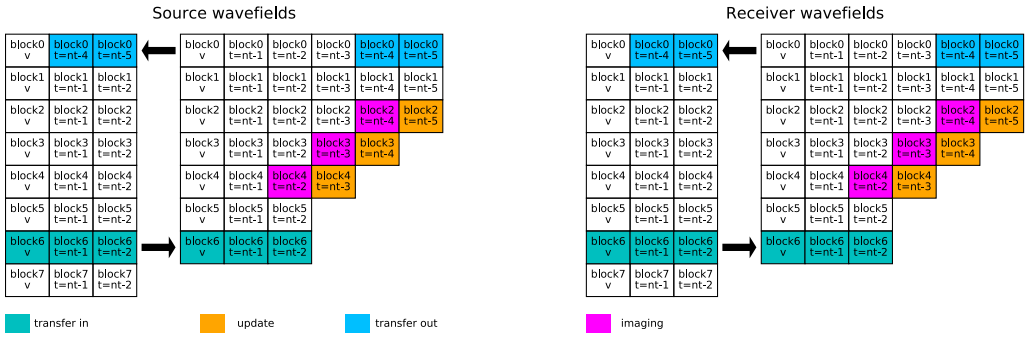
of $\frac{\text{NBLOCK}}{\text{NUPDATE}}$ GPUs, with minimal NUPDATE, i.e. just enough to overlap the transfer. Doing so will open up another level of parallelization, in which each GPU group processes one seismic shot. This parallelization can easily be achieved with OpenMP or the standard C++11 thread library.

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, I adopt random boundary condition (Shen and Clapp, 2015). This adds 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 6.3).

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

HETEROGENEOUS CLUSTER ADAPTATION

In order to cope with the large number of shots in 3D seismic field data, I take avantage of all computational resources available at hand: the XStream GPU cluster, CEES Mazama CPU cluster, and SEP GPU machines. To adapt to different cluster environments, I developed a job distributor, whose purpose is to create job descriptions, submit jobs, check job statuses, and gather results. There have been a number of implementations of a job distributor by SEP colleagues (most well-known are Bob Clapp's and Zhang Yang's, both in Python), but my C++ implementation

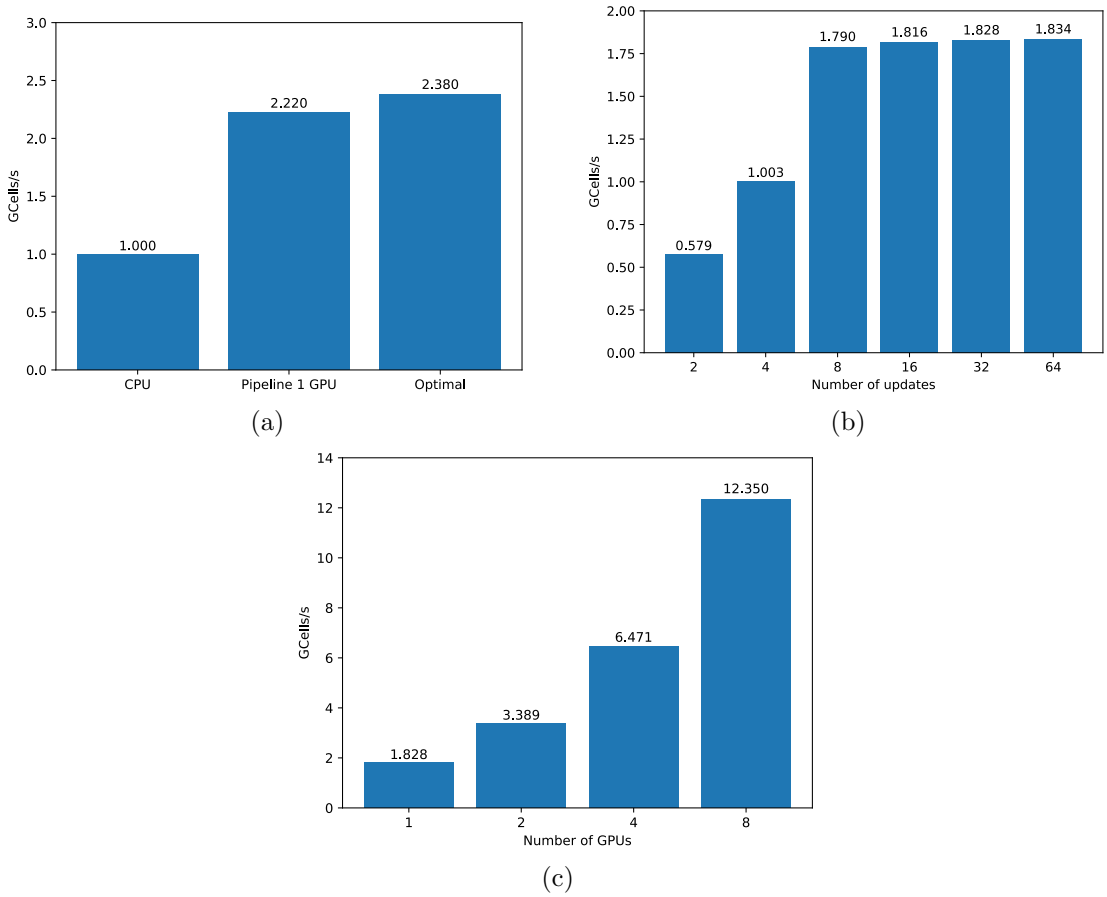


Figure 6.4: Performance reports. (a) Comparison of CPU, pipeline, and optimal codes for isotropic scalar wave equation. (b) Performance of pipeline algorithm with different numbers of update for 1 GPU. (c) Performance of pipeline algorithm with different numbers of GPUs. [**CR**]

overlaps the process of stacking over shots with the propagation by constantly checking and adding the gradients as soon as a job completes (Algorithm 2). There are two parameters. MAX_SHOT_PER_JOB is to control how many shots to assign to one job. Usually one shot is assigned to one job, but this number might increase if I want to reduce number of outputs to avoid storage overflow or to leverage between propagation time and stacking time. The second parameter is MAX_FAIL, which is the maximum number of allowed fails per job.

The cluster job distributor (Algorithm 2) is then combined with SSH tunneling (I use the open-source libssh library, `https://www.libssh.org/`) to spread the workload and communicate among the XStream cluster, CEES cluster, and SEP machines (Algorithm 3). This algorithm is used in the subroutine that computes the FWI objective function and gradients. Reverse-communication design of the LBFGS solver (Nocedal (1980)) allows great flexibility in implementing this subroutine. If these clusters and machines were not separated by firewalls, socket programming could have been used to keep all data transfers in memory, instead of going through disks.

Having multiple clusters raises the question of how to efficently and intelligently partition the workload to avoid being slowed down by one single crowded cluster. My simple solution is to time the execution on each cluster, re-evaluate their usage, and adaptively partition the workload at each iteration. This solution works when the cluster usage fluctuates at a greater interval than the time one iteration takes. For the two clusters at hand, XStream and CEES, this is usually the case. Another SEP colleague of mine, Yinbin Ma, recently implements an even finer-grain (at level of each shot/job instead of each iteration) adaptive distribution mechanism. Though being more efficently making use of resources, his implementation uses more memory and makes more cross-network transfers.

CONCLUSIONS

I have implemented a pipeline algorithm for 3D time-domain finite-difference waveform inversion on GPUs and have shown that this algorithm scales well with number

Algorithm 2 A pseudo-code for a cluster job distributor.

```
1: nShotLeft ← nShotTotal
2: while nShotLeft > 0 do
3:     nShotForThisJob = min (nShotLeft, MAX_SHOT_PER_JOB)
4:     Create script with job description
5:     Submit job script
6:     nFail ← 0
7:     while Submission fails and nFail ≤ MAX_FAIL do
8:         Sleep some seconds
9:         Resubmit job script
10:        nFail ← nFail + 1
11:    end while
12:    if Submission succeeds then
13:        Record job id and add job to jobList
14:    end if
15:    nShotLeft ← nShotLeft − nShotForThisJob
16: end while
17: nJobComplete ← 0
18: while nJobComplete < jobList.size() do
19:    for job in jobList that has not completed or failed do
20:        Check job status
21:        if Job completed then
22:            Gather output from this job
23:            Set this job's state to be completed
24:            nJobComplete ← nJobComplete + 1
25:        else if Job fails then
26:            nFail ← nFail + 1
27:            if nFail > MAX_FAIL then
28:                Set this job's state to be failed
29:                nJobComplete ← nJobComplete + 1
30:            else
31:                Resubmit this job and record new id
32:            end if
33:        end if
34:    end for
35: end while
```

Algorithm 3 A pseudo-code to spread workload and communicate among different clusters and machines.

```
Establish SSH and SFTP connections to CEES cluster
Establish SSH and SFTP connections to SEP machine
for k = 0 to N do:
    Partition workload into three portions based on initial guess of cluster usage
    Spawn two new threads (three including the master thread) to work in parallel
    Thread number 1 sends data and necessary parameters to CEES cluster and processes one portion of the workload
    Thread number 2 sends data and necessary parameters to SEP machine and processes another portion of the workload
    The master thread processes the third portion on XStream cluster
    Wait for all threads to finish and gather outputs
    Re-evaluate cluster usage for better partition in the next iteration based on execution times of each cluster in this iteration
    Call LBFGS solver to update model
end for
Close all connections
```

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. I have also developed two algorithms to distribute jobs and communicate across heterogeneous clusters and machines.