Fast Seismic Modeling and Reverse Time Migration on a GPU Cluster

Rached Abdelkhalek
Henri Calandra
TOTAL, avenue Larribau
F-64000 Pau
E-mail: name.surname@total.com

Olivier Coulaud
Jean Roman
INRIA/HiePACS project,
351 cours Libération,
F-33405 Talence

Guillaume Latu
Strasbourg University
& INRIA/Calvi project, LSIIT,
bd Seb. Brant,
F-67400 Illkirch

ABSTRACT

We have designed a fast parallel simulator that solves the acoustic wave equation on a GPU cluster. Solving the acoustic wave equation in an oil exploration industrial context aims at speeding up seismic modeling and Reverse Time Migration. We consider a finite difference approach on a regular mesh, in both 2D and 3D cases. The acoustic wave equation is solved in either a constant density or a variable density domain. All the computations are done in single precision, since double precision is not required in our context. We use CUDA to take advantage of the GPUs computational power. We study different implementations and their impact on the application performance. We obtain a speed up of 10 for Reverse Time Migration and up to 30 for the modeling application over a sequential code running on general purpose CPU.

KEYWORDS: Seismic modeling, Finite Difference, Reverse Time Migration, GPGPU, CUDA.

1. INTRODUCTION

The extraordinary challenge that the oil and gas industry must face for hydrocarbon exploration requires the development of leading edge technologies to recover an accurate representation of the subsurface. Seismic modeling and Reverse Time Migration (RTM) based on the full wave equation discretization, are tools of major importance since they give an accurate representation of complex wave propagation areas. Unfortunately, they are highly compute intensive. Advances in High Performance Computing technologies resulted in renewed attention from the seismic community to these techniques.

The recent development in GPU technologies with unified architecture and general-purpose languages coupled with the high and rapidly increasing performance throughput of these components made General Purpose Processing on Graphics Processing Units (GPGPU) an attractive solution to speed up diverse applications [1].

In this work, we will discuss how seismic modeling and RTM can take advantage of GPUs to achieve massive computation capacity. The remainder of this paper is organized as follows: in Section 2 we begin by describing the context, the sequential algorithm and the CPU implementation of the modeling and RTM applications. We also introduce the GPU architecture and programming model. The GPU implementations and optimizations are described in Section 3. In Section 4 we provide and analyse performance results. We conclude in Section 5 by validating the numerical results.

2. BACKGROUND

2.1. Context

2.1.1. Seismic Surveys

Seismic surveys are the first step in oil and gas exploration and production projects. A typical seismic survey consists in generating a seismic wave by sources placed at the surface such as air guns in marine surveys or dynamite in land surveys. The seismic wave which propagates downward in the subsurface is refracted and/or reflected at the interfaces of the geological layers. The refracted and/or reflected wave propagates upward and is recorded by sensors called geophones. Each geophone records time series of events which are called seismic traces or seismograms. The recorded wave field is used as an initial condition or boundary condition to determine the earth's subsurface structure. The refraction and the reflection of seismic energy at geological interfaces exactly follow the same laws that govern the refraction and reflection of light through prisms.

978-1-4244-4907-1/09/$25.00 ©2009 IEEE

2.1.2. Seismic Modeling

Numerical seismic modeling [2] aims at simulating seismic wave propagation in the earth in order to generate synthetic seismograms that are the seismograms that a set of sensors would record, given an assumed structure of the subsurface. This technique is a valuable tool for seismic interpretation and an essential part of seismic inversion algorithms. Seismic modeling is also of major importance in the evaluation and the design of seismic surveys.

Among the numerous approaches to seismic modeling, direct methods based on approximating the geological model by a numerical mesh is of particular interest. These techniques are also called grid methods and full-wave equation methods, the latter since the solution implicitly gives the full wave field. Direct methods can be very accurate when a sufficiently fine grid is used. Furthermore, full wave methods are well suited for the generation of snapshots which can be an important aid in the interpretation of the results. A disadvantage of these general methods, however, is that they can be more expensive than analytical and ray methods in terms of computational demand.

2.1.3. Reverse Time Migration

Reverse Time Migration (RTM) is a technique for creating seismic images in areas of complex wave propagation, providing imaging of so called turning and prismatic waves. RTM was introduced in the 1980's [3] but despite showing promising imaging capabilities, its high computational cost prevented it from being used in practice, until recent advances in HPC technologies. RTM is based on the simulation of waves propagation: both source and receivers wave fields are propagated respectively forward and backward in time. These wave fields are then compared using an imaging condition for corresponding time steps in order to form the subsoil image. Figure 1 (extract from[4]) depicts the RTM backward sweep, for times $t = 1.20s$ (a), $0.75s$ (b) and $0.30s$ (c): source wave field (left) and receiver wave field (center) are propagated backward in time. the imaging condition is then applied to form the subsurface image (right).

2.2. Governing Equations

The three dimensional acoustic wave equation (1) links the pressure field $u(x, y, z, t)$ to the density $\rho(x, y, z)$ and the velocity $c(x, y, z)$[5].

$$\frac{1}{c^2\rho}\frac{\partial^2 u}{\partial t^2} = \nabla.(\frac{1}{\rho}\nabla u) \tag{1}$$

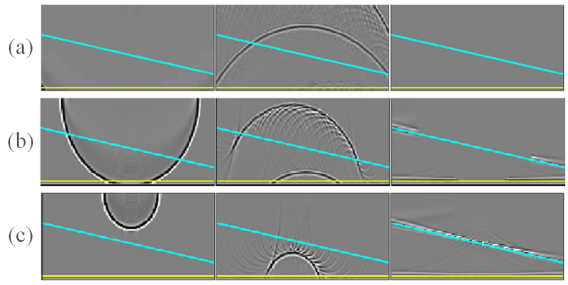

Figure 1. Reverse Time Migration Backward Sweep.

Using finite difference (FD) methods to solve the wave equation is one way among others to tackle direct methods. The way this equation is derived, among a regularly meshed domain, is described in [6]: we write a cascaded first order spatial difference expression to compute the second time difference of the wave field:

$$\frac{\partial^2 u}{\partial t^2} = K[\frac{\partial_-}{\partial x}(\frac{1}{\rho}\frac{\partial_+ u}{\partial x}) + \frac{\partial_-}{\partial y}(\frac{1}{\rho}\frac{\partial_+ u}{\partial y}) + \frac{\partial_-}{\partial z}(\frac{1}{\rho}\frac{\partial_+ u}{\partial z})] \tag{2}$$

where $K = c^2\rho$ is the bulk modulus. The ∂_- and ∂_+ symbols denote the spatial difference operators that are centered halfway between grid points either forward or backward in the direction of the spatial difference. We use operators with a 3D stencil width of 8. So for example, the first derivative of u with respect to x evaluated at $(i+\frac{1}{2})\Delta x$ is written

$$\frac{\partial_+}{\partial x}u((i+\frac{1}{2})\Delta x) = \sum_{n=0}^{3} a_n[u((i+1+n)\Delta x) - u((i-n)\Delta x)] \tag{3}$$

The a_n coefficients are the 8th order finite difference operator optimized coefficients [7].

When the density is considered to be constant in all the domain, equation (1) is simplified to:

$$\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} = \Delta u \tag{4}$$

This approximation is especially done during migration process. The discretization of this equation is done with a second order in time leap-frog scheme and an 8th-order centred difference scheme in space with either Taylor or optimized coefficients [8], which leads in 2D to:

$$U_{i,j}^{n+1} = 2U_{i,j}^{n} - U_{i,j}^{n-1} + c^2\Delta t^2\Big[\frac{1}{\Delta x^2}\sum_{m=-4}^{4} b_m U_{i+m,j}^{n} + \frac{1}{\Delta y^2}\sum_{m=-4}^{4} b_m U_{i,j+m}^{n}\Big] \tag{5}$$

with $U_{i,j}^{n} = u(i\Delta x, j\Delta y, n\Delta t)$.

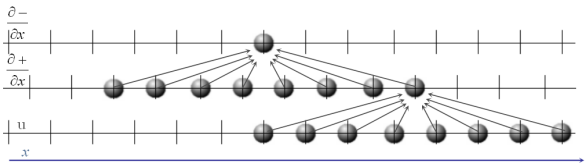

Figure 2. Data Dependency along X Axis.

2.3. Modeling and RTM on Homogeneous Processors

We describe in this section our CPU code translation of discretizations of (1) and (4). The latter is easily inferred from (5). At each grid point the stencil is applied to compute the Laplacian value. Thus, to update one grid point, 25 data reads are performed from the wave field's last update. Data are contiguous along X axis. When a domain is too large to fit into the CPU cache size, which is the case almost all the time, data accesses along Y and Z axes become expensive. Cache blocking [9], along Y axis is used to increase data reuse and to exploit the memory hierarchy.

No cache strategy is implemented for variable density domains. In this case, we need to loop over all the grid points twice as depicted in the algorithm below. During first sweep, three 3D arrays are filled with the forward first derivatives along each axis. Then, backward first derivatives are computed using these arrays and the density array. The stencil is then twice as large as in the density case. Figure 2 depicts the data dependency along X axis.

Algorithm 1 Sequential variable density seismic modeling.

```
1: for time = t_start to t_end do
2:     add source term
3:     for all Domain grid points do
4:         compute forward first derivatives
5:     for all Domain grid points do
6:         compute backward first derivatives
7:         update wave field
8:     save seismogram
```

Algorithm 1 describes the sequential variable density seismic modeling CPU implementation. The reference CPU implementation that we use to evaluate our GPU accelerated solution is parallel. It is based on subdomain decomposition. Thus, if a domain is too large to fit into one host memory, it is divided into subdomains mapped onto several hosts. Ghost nodes are exchanged via asynchronous MPI communications. Ghost node thickness is determined by the stencil used to solve the wave equation: 4 planes in the constant density case, 8 planes in the variable density case.

In practice, since the simulated domain cannot extend infinitely, damping zones are added at the borders of the domain to avoid reflections. In this zones we use Perfectly Matched Layers [10] to simulate non reflecting boundaries.

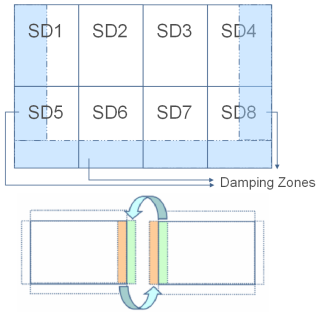

Figure 3. 2D Domain Decomposition and Ghost Node Exchanges.

Algorithm 2 Reverse Time Migration algorithm

```
1: for time = t_start to t_end do
2:     add source term
3:     for all domain grid points do
4:         take forward time step
5:     save boundaries
6: for time = t_end to t_start do
7:     read saved boundary
8:     for all domain grid points do
9:         backward time step source wave field
10:    add receiver term
11:    for all domain grid points do
12:        backward time step receiver wave field
13:    for all domain grid points do
14:        apply imaging condition
```

The Reverse Time Migration algorithm is listed in Algorithm 2. The imaging condition applied during backward recursion consists in the accumulation of the cross-correlation between source and receiver wave fields over the time iterations.

2.4. GPGPU: Architecture and Programming Model

In this section, we briefly describe the S1070 GPUs architecture and programming model. Even though other models exist, we focus on the Compute Unified Device Architecture (CUDA) that we use for our implementations. We also focus on architectural features that are most relevant to our use. We invite the reader to refer to [11] for a more detailed description.

The S1070 blades are composed of 4 T10 graphics processing units (GPU) embedded with 4GB of memory per GPU. Each pair is connected to a host node via a PCIe 2.0 bus, acting as coprocessors. PCIe gen 2.0 connection delivers a theoretical peak throughput of 8 GB/s one way. In practice, we have been able to reach a sustained 5.6 GB/s bandwidth when using DMA transfers. Since 2 GPUs share the same PCIe bus, the data transfer rate is reduced to 3,3 GB/s for each GPU, when performing intensive memcopies between host and GPUs in the same direction. Writing to one GPU while reading from the other seems to give higher throughput (measured average of 4.3 GB/s).The limited PCIe Bandwidth -compared to the 102 GB/s memory bandwidth- and its latency (16 μs), makes it the main bottleneck when tackling small problems in terms of data sizes or problems of low computing intensity.

The T10 GPU can be seen as a set of 30 *multiprocessors*. Each multiprocessor is composed of 8 *streaming processors* running in a Single Instruction Multiple Data (SIMD) like way. These processors can execute 3 simple precision floating point operations per clock cycle. Our T10 GPUs have a clock rate of 1.44 GHz, providing thus a theoretical peak performance of $30 \times 8 \times 3 \times 1.44 = 1$ TFlops per GPU and 4 TFlops per S1070 blade.

The processors inside a multiprocessor have access to 16,384 32-bit registers (divided among processors), and to 16 kB shared memory space that can be seen as a cache memory with very low latency (4 clock cycles per read/write if no conflict). All the GPU's processors have read/write access to the off-ship global memory (nearly 4GB) but with a higher latency varying from 400 to 600 clock cycles.

To harness the GPU computing power and make it available for non graphics programmers, NVIDIA introduced CUDA. CUDA is based on a Single Program Multiple Data (SPMD)[1] paradigm: the programmer writes a portion of code (the *kernel*) to be executed by several *threads* in parallel on different data. CUDA defines a thread hierarchy depicted in Figure 4 in order to organize threads in a geometric topology. Thus threads are grouped into 1D, 2D or 3D *thread blocks*. These blocks are arranged into 1D or 2D *grids*. This topology matches the thread organization to the GPU physical structure. In fact, threads of the same block are run on the same multiprocessor, making them able to have access to the multiprocessor's shared resources (same shared memory space, texture and constant memory cache) and to coordinate their activities using a barrier synchronization function. Each thread is then identified according to its thread ID and the block ID of the block it belongs to. Threads of consecutive thread IDs are grouped into *warps* of 32 threads.

1. NVIDIA introduced the SIMT acronym standing for Single Instruction Multiple Thread.

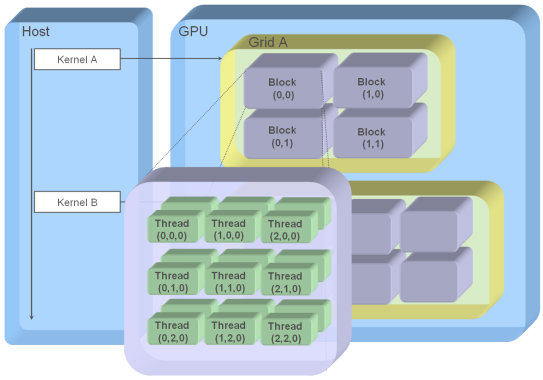

Figure 4. CUDA Programming Model: Grids of Thread Blocks.

3. GPU IMPLEMENTATIONS

To measure and to understand GPU code performance, CUDA introduces new terms and metrics. The concept of occupancy is of major importance when designing CUDA kernels. Occupancy is defined as the ratio of active warps per multiprocessor to the maximum number of active warps (32 for the T10 GPU). The number of active warps is defined by the availability of shared resources inside the multiprocessor.

We start by considering modeling the wave propagation in a constant density domain. This case has been the first example we tackled due to its simplicity and its high regularity and parallelism. We expose the learning curve we followed in order to improve the application performance. Then we describe the variable density modeling and the RTM implementation and performance results.

3.1. Constant Density Modeling

The equations and computations are described in 2.3. We have studied different steps to optimize this application on both the CUDA kernel and the host-GPU communication sides.

3.1.1. Host-GPU Communication

As described in 2.3, to update pressure field at a given iteration t_{n+1}, pressure field updates at iterations t_{n-1} and t_n are required. Data in the GPU global memory are persistent across different kernel launches. Time evolution of the wave field is then performed in the GPU memory by swapping t_n and t_{n+1} arrays. Only the ghost nodes need then to be exchanged between host and GPU at each time step. Exchanging only ghost nodes instead of the whole domain, although requiring more programming effort, largely reduces the amount of data exchanged and thus computing time. For example, transferring a whole 3D $528 \times 254 \times 1067$ domain to/from GPU takes approximately $0.1s$, that is equivalent to the time needed for a single time iteration.

3.1.2. CUDA Kernel Implementation

Our first approach has been to use global memory and to dedicate a CUDA thread for each grid point in the domain. Although it reached a 66% occupancy on N80 GPUs, this direct implementation was of poor performance essentially because it was memory bounded: each thread makes 25 global memory read accesses and 1 global memory write access to the wave field arrays.

To limit memory accesses, we used shared memory: data are first copied to shared memory (texture fetches in 2D, global memory in 3D), then read by threads of the same block. This classical technique reduces global memory accesses for the same data, but increases shared memory use. Since shared memory is limited to $16KB$ for a multiprocessor, using $16 \times 4 \times 4$ thread blocks requires at least 9kB of shared memory for each thread block. That means that only one thread block of 256 threads can run at a given time on one multiprocessor instead of running 768 possible threads on devices of compute capabilities[2] under 1.2 (33% occupancy) and 1024 threads for other devices (25% occupancy).

To limit shared memory use, and thus increase occupancy, we adopted a novel strategy: instead of dedicating a thread to each grid point, we used a sliding window algorithm, sliding over planes in the z direction. We used shared memory to store the $(x, y, z = z_{current})$ plane to be updated, so that all the threads can compute Laplacian value in the x and y directions. For the z direction, each thread loads into *float4* registers the 4 pressure values under and over the current plane. At each loop iteration, corresponding to a shift along z direction, wave field values are shifted as shown in Figure 5. Thus, only the $(x, y, z = z_{current+4})$ plane is read from global memory. This kernel (Basic) showed to be the most efficient. Adding padding to ensure global memory coalescing added 15% speed up on GPUs with compute capability under 1.2.

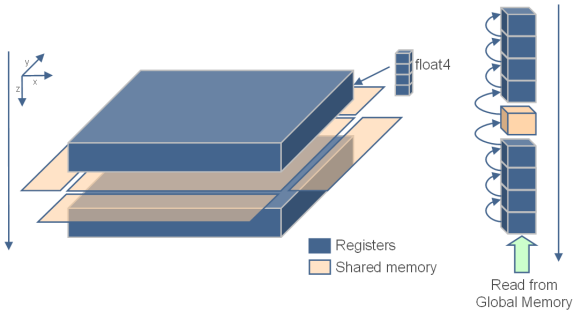

Figure 5. 3D Modeling Kernel

Table 1. Modeling CUDA Kernels, Resources and Occupancy

Kernel	Shared Memory	Registers	Occupancy
Basic	2400	20	75%
Damping	2432	28	50%
WF	2432	20	75%
PML	2720	17	75%

We used the same approach to implement the wave propagation in the damping zones (Damping). This induces more complex computations, leading to more memory accesses and registers needs. This makes the occupancy limited by registers used by each thread. We use thread blocks of size $256(16 \times 16)$. To achieve a 100% occupancy, each thread must use less than 16 $(16,384/256 \times 4)$ registers. Using up to 21 registers achieves a 75% occupancy. The full implementation of the damping kernel uses 32 registers leading to a 50% occupancy. To limit register use, we split the kernel into 2 separate ones. The first kernel (WF), updates the wave field array. The second one (PML) updates the damping function array. In Table 1 we sum up the kernels resources use and occupancy.

To solve the problem over a whole subdomain, we can use

2. The compute capability of a device indicates its core architecture and the features it supports.

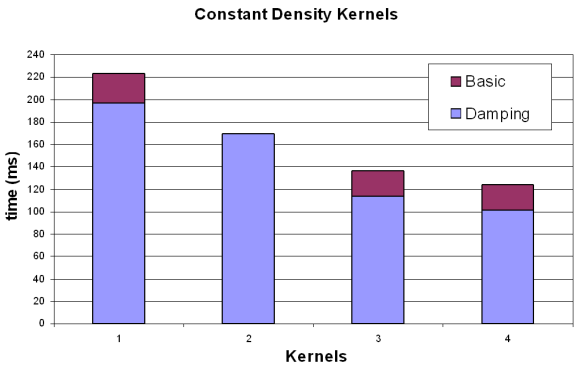

Figure 6. 3D Constant Density Modeling Strategies

two different strategies. Corresponding times are reported in Figure 6 for one time iteration on a domain of size $528 \times 254 \times 1067$ including 5 damping zones. We can either apply the damping kernel to all the domain (2) or apply it on the borders (or not, depending on the domain position) and apply next the basic kernel inside the domain (1,3 and 4). In (1) we limited the registers use to 21 using a compiler option. This increases occupancy but induces use of local memory which reduces the kernel's performance. In (3) Damping kernel was used in the damping zones while in (4) we used WF kernel and PML kernel. The latter strategy is the most efficient one.

3.2. Variable Density Modeling

The communication scheme remains the same, except that ghost node width is doubled. The kernel radically changes. In fact, as described in 2.3, the stencil is twice as large as in the constant density case. This makes the use of registers in a similar sliding window way impossible. The solution we propose computes the first derivative along the z axis in a first sweep, using a separate kernel. Then, a second kernel performs the remaining computations.

3.3. Reverse Time Migration

Our current implementation of the RTM propagates the source wave field on the GPU during the forward sweep. Then, during the backward sweep, the receiver wave field propagation is accelerated using GPU while the host CPU takes in charge the source wave field retro-propagation and the imaging condition. Thus, Reverse Time Migration induces more CPU-GPU communications especially during the backward sweep where the whole receiver wave field has to be sent back to CPU to apply the imaging condition. With CUDA, we can overlap communications with computations using CUDA asynchronous communications and

Table 2. Technical Specifications of the System Used in our Experiments.

Blade	CPU Node	NVIDIA S1070
Number	10	5
Processor	Xeon 5405	T10
Sockets $\times$ cores	2×4	4×240
Clock Frequency (GHz)	2.00	1.44
Memory per Blade (Gbytes)	16	16 (4×4)
Cache/Shared Memory	$4 \times 6MB$	$30 \times 16KB$
Peak Performance (GFlops)	64	1036

kernel launches. The domain to be updated on the GPU is decomposed into 4 *streams*. Each stream consists in a data transfer from host to GPU, a kernel launch, and a data transfer from GPU to CPU. Thus while executing a stream's kernel, the data corresponding to the next stream are transferred.

4. PERFORMANCE ANALYSIS

In this section we report and analyse some performance results obtained using GPUs and compare them with the CPU implementation. We focus on the scalability of each solution. The reference time is the time to process the entire domain without using any subdomain decomposition. Thus, the CPU reference time is given by running a sequential implementation on a single CPU core; and the GPU reference time is given by using one GPU. This implies that a GPU is considered as a processing unit, even though it comprises several multiprocessors. We also focus on the ratio of the execution time of the CPU-based implementation to that of the GPU-based one depending on the dataset size.

We consider a 3D test case (hereafter referred to as 3DModel) of grid dimensions $521 \times 254 \times 1067$ with $\Delta x = 12.5$, $\Delta y = 12.5$ and $\Delta z = 10$ in meters. Thus this model covers a surface of $20.7km$ and goes $10.7km$ in deep.

4.1. GPU Cluster Testbed

Our GPU cluster testbed is composed of 10 Xeon bi-socket quad-core nodes coupled with 5 NVIDIA TESLA S1070 servers. Each node is connected via one PCIe gen2 bus to the TESLA server. The TESLA server is composed of 4 T10 GPUs. Each pair is connected via a switch to a PCIe 2.0 connection. Thus each node has access to 2 GPUs via the same bus. Table 2 reports the detailed technical specifications of the system used in our experiments.

Table 3. Constant and Variable Density Modeling Averaged Times in Seconds on 3DModel.

Number of subdomains (N)	1	2	4
1 Socket ($\rho = ct$)	3.90	2.60	1.45
N Sockets ($\rho = ct$)	3.90	1.59	0.94
1 Socket ($\rho(x, y, z)$)	9.23	7.55	4.96
N Sockets ($\rho(x, y, z)$)	9.23	4.54	1.59

Table 4. Constant Density Modeling Averaged Time in Seconds for one Iteration on 3DModel

Number of subdomains	GPU time	CPU time	Ratio
1	0.146	3.9	26.71
2	0.083	1.59	19.15
4	0.046	0.94	20.43
8	0.034	0.47	15.16

4.2. CPU Scalability

We compare the scalability of the CPU application, using cores of the same processor socket, and using different processors, for the 3DModel test case. Results are summarized in Table 3. They show that the memory access is the bottleneck when using Xeon Harpertown processors.

4.3. Constant Density Modeling

We study the speed up obtained with the implementation described below for the 3DModel. All the GPU computing times indicated hereafter, unless explicitly stated, take into account the kernel execution time as well as the CPU-GPU communication needed to initiate and finish the kernel.

Results reported in Table 4 show that the CPU/GPU time ratio decreases with the number of subdomains increasing because the computing time decreases and the communication time becomes more predominant (both CPU-GPU and MPI communications). Yet, only computing time is reduced when using GPUs.

In Figure 7 we report the averaged computing times in seconds for one time iteration on domains of increasing sizes and the corresponding CPU/GPU time ratio for a single domain. This figure shows that the ratio increases with the size of the domain increasing. This is due to the fact that for a domain of size n^3, CPU-GPU communication cost grows as $\Theta(n^2)$ while computing time grows as $\Theta(n^3)$. Yet, the CPU/GPU time ratio can be written as

$$R = \frac{t_{CPU}[\theta(n^3)]}{t_{comm}[\theta(n^2)]+t_{GPU}[\theta(n^3)]}.$$

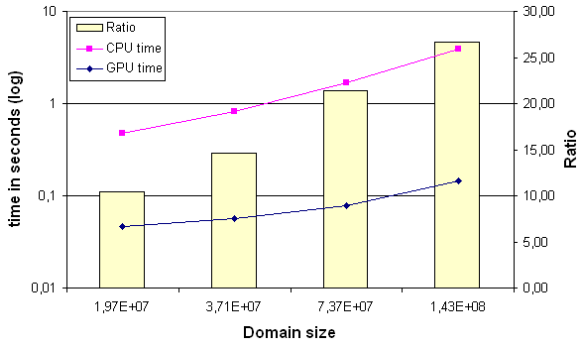

Figure 7. 3D Constant Density Modeling Times.

Table 5. Variable Density Modeling Averaged Time in Seconds for one Iteration on 3DModel

Number of subdomains	GPU time	CPU time	Ratio
1	0.342	9.90	28.90
2	0.183	4.75	25.96
4	0.11	2.45	22.34
8	0.073	1.22	16.66

4.4. Variable Density Modeling

The same statements can be made about variable density modeling as shown in Table 5 and Figure 8. In Table 6 reported data transfer times reinforce that the overhead due to MPI communications increases with the number of subdomains. When using 8 GPUs, 40% of one iteration time is spent in communications which significantly reduces the speed-up over the CPU implementation.

We evaluate the difference between a configuration with 2 GPUs accessible on the same node via the same PCI-X 2.0 connection and 2 GPUs hosted by two distinct nodes. As stated in 2.4 the data transfer rate is reduced when accessing 2 GPUs through the same PCI-X connection. Communication between the two MPI processes is impacted as well: in one case shared memory is used, in the other case high performance infiniband network. Thus communication times are expected to be quite different in these two configurations. Results are reported in Table 7 and show that even though communication times are significantly different, they remain quite low compared to

Table 6. Averaged Time in Seconds and Speedup (in Parenthesis) for one Iteration on 3DModel

Nb GPU.	2	4	8
Data transfer	0.0045	0.0133	0.029
GPU kernel	0.1702 (2.00)	0.0847(4.02)	0.0441 (7.72)
Total time	0.1831 (1.87)	0.1097(3.12)	0.0735 (4.66)

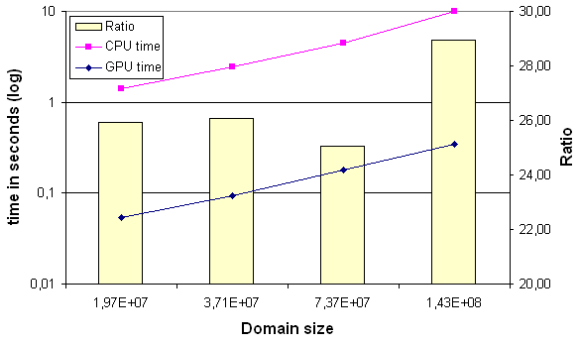

Figure 8. 3D Variable Density Modeling Times.

Table 7. Averaged Time in Seconds for one Iteration on 3DModel

Configuration	Shared Memory	InfiniBand
GPU kernel	0.1708	0.1702
Other comp.	0.0163	0.0129

the computation time. Then, the overall simulation time is slightly impacted. The host-GPU, the host memory and the network bandwidths are not a bottleneck in this particular case.

4.5. Reverse Time Migration

We report in Figure 9 times and speedup for the Reverse Time Migration. The RTM implementation involves more host-GPU communications than the modeling. This reduces the obtained CPU/GPU time ratio very significantly.

5. CONCLUSION

We ported a seismic modeling application and Reverse Time Migration on a GPU cluster. The difference between

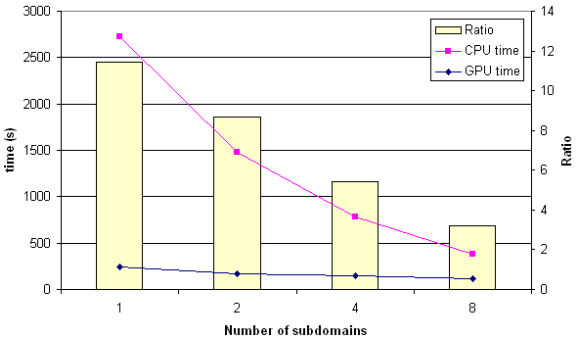

Figure 9. Reverse Time Migration Times in Seconds for one Iteration on a $288 \times 118 \times 338$ Test Case.

synthetic seismograms produced using the CPU and the GPU implementations for a realistic data set used in production process is proportional to the wave field amplitude but remains very low (0.1% in terms of percentage error). This validates our implementation. We compared our GPU implementation to the original CPU version on several test cases and obtained a performance increase of 10x for RTM and up to 30x for modeling.

Results may still be improved by recovering MPI communications by GPU computations. This shows that GPGPU solutions are worth being used in a large-scale industrial context, especially when double precision is not required.

REFERENCES

[1] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[2] J. M. Carcione, G. C. Herman, and A. P. E. ten Kroode, "Seismic modeling," *Geophysics*, vol. 67, no. 4, pp. 1304–1325, 2002.

[3] E. Baysal, D. D. Kosloff, and J. W. C. Sherwood, "Reverse Time Migration," *Geophysics*, vol. 48, no. 11, pp. 1514–1524, 1983.

[4] B. L. Biondi, *3D SEISMIC IMAGING*, ser. Investigations in Geophysics. SEG, 2006.

[5] J. F. Claerbout, *IMAGING THE EARTH'S INTERIOR*. Blackwell Scientific Publications, 1985.

[6] J. T. Etgen and M. J. O'Brien, "Computational methods for large-scale 3d acoustic finite-difference modeling: A tutorial," *Geophysics*, vol. 72, no. 5, pp. SM223–SM230, 2007.

[7] O. Holberg, "Computational aspects of the choice of operator and sampling interval for numerical differentiation in large-scale simulation of wave phenomena," *Geophysical prospecting*, vol. 35, no. 6, pp. 629–655, 1987.

[8] F. Ye and C. Chu, "Dispersion-relation-preserving finite difference operators: derivation and application," *SEG Technical Program Expanded Abstracts*, vol. 24, no. 1, pp. 1783–1786, 2005.

[9] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Implicit and explicit optimizations for stencil computations," in *MSPC '06: Proceedings of the 2006 workshop on Memory System Performance and Correctness*. New York, NY, USA: ACM, 2006, pp. 51–60.

[10] G. Cohen, *HIGHER-ORDER NUMERICAL METHODS FOR TRANSIENT WAVE EQUATIONS*, ser. Scientific Computation. Springer-Verlag, 2001.

[11] *NVIDIA CUDA PROGRAMMING GUIDE*, 2nd ed., NVIDIA, June 2008.