GPU Technology Applied to Reverse Time Migration and Seismic Modeling via OpenACC

Ahmad Qawasmeh, Barbara Chapman
Department of Computer Science
University of Houston, Houston, Texas
{arqawasm, chapman}@cs.uh.edu

Maxime Hugues, Henri Calandra
Advanced Computing Department
TOTAL E&P USA, INC, Houston, Texas
{maxime.hugues,
henri.calandra}@total.com

ABSTRACT

GPU computing offers tremendous potential to accelerate complex scientific applications and is becoming a leading force in speeding up seismic imaging and velocity analysis techniques. Developing portable code is a challenge that can be overcome using emerging high-level directive-based programming model such as OpenACC. In this paper, we develop OpenACC implementations for both seismic modeling and Reverse Time Migration (RTM) algorithms that solve the isotropic, acoustic, and elastic wave equations. We employ OpenACC to take advantage of the computational power of two Nvidia GPU cards: **1) M2090** and **2) K40**, residing in IBM and CRAY XC30 clusters respectively. Although we implement a hybrid OpenACC-MPI approach to parallelize seismic modeling and RTM on multiple GPUs, in this paper, we focus on developing mapping techniques to exploit potentials of one GPU. We observe an incremental improvement in performance while exploring different optimization techniques. Adequate code restructuring to tap GPU's potential seems critical. Depending on the intensity of computations, different propagators exhibit different speedup behaviors. A performance enhancement of $\sim$ 10x was obtained, when the acoustic model was ported to a single GPU, compared with a 1.3x speedup obtained using the isotropic model.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—Parallel algorithms

General Terms

Algorithms, Design, Measurement

Keywords

OpenACC, Seismic imaging, Reverse Time Migration, GPU, Accelerator, MPI

PMAM '15 February 07-11 2015, San Francisco, CA, USA

http://dx.doi.org/10.1145/2712386.2712401.

1. INTRODUCTION

The massive complexity of the exploration process in the oil and gas industry requires high performance computing technologies to produce an accurate and fast representation of the subsurface. Seismic modeling and Reverse Time Migration (RTM), which are based on a full wave equation discretization, are important applications that can solve complex wave propagation fields across different medias. The graphics processing unit (GPU), first invented by Nvidia in 1999, is the most prevalent parallel processor to date. GPUs can be identified as a combination of multicore general purpose processors and vector processing units, intending to generate a higher level of parallelism. Efforts to exploit the GPU for non-graphical applications have been underway since 2003, as GPUs were originally targeting graphical-based applications. Nowadays, GPUs are used in a large variety of applications that exhibit different characteristics and purposes.

Using GPU programming languages, such as CUDA [7] for Nvidia or OpenCL [11], requires a thorough understanding of the GPU architecture and design to accelerate applications. Prior to the introduction of high level GPU APIs, developers targeting HPC accelerators have had to rely on language extensions to their programs. Host+accelerators programmers have needed to program at a detailed level including a need to understand and specify data usage information and manually construct sequences of calls to manage all movements of data between host and accelerator. OpenACC [12], which is a standard directive-based GPU programming model, has alternatively gained a wider popularity among domain specialists, especially in the seismic community, because of its simplicity and portability across many platforms. Using this API, a single version of an application should be maintained with specific compiler options to be used to activate the inserted directives.

In this work, we have mainly focused on exploiting the massive computation capacity provided by the GPU technology to better analyze and process seismic modeling and RTM. We experienced different implementations of OpenACC against these applications while using different seismic imaging techniques. We also employed different optimization approaches that target the design and memory hierarchy provided in GPUs. We observed the influence of using different versions of CUDA, PGI, as well as CRAY compilers on the overall performance. We realized that different compiler versions have different heuristics to generate the GPU code. In some cases, conforming to the specification more closely might result in worse performance.

We also check the impact of using two Nvidia Tesla Cards: Fermi M2090 and Kepler K40 on the obtained performance in terms of speedup and throughput. Kepler cards arithmetically outpace Fermi cards in terms of memory bandwidth, number of cores, and throughput. Our experiments emphasized the advantage of using Kepler cards over Fermi cards.

The main motivation behind this work lies in observing the feasibility of using OpenACC to extract the maximum potential of GPUs in order to improve the performance of velocity analysis in seismic imaging applications. Two popular seismic applications used in the real world oil exploration industrial context were examined and optimized: **1) Seismic Modeling 2) Reverse Time Migration (RTM)**. The aforementioned observation was performed through the following contributions:

1. Studying and analyzing all aspects in these large applications to determine what to port on GPUs and how to minimize host-device data transfers and reduce the latency produced by accessing GPUâĂŹs global memory.
2. Combining OpenACC directives with various optimization techniques that include code transformations, loop scheduling, and compiler flags.
3. Assessing the OpenACC programming model with respect to its:
   - ease of use
   - capability to adapt to different GPU cards and compilers
   - performance gain compared with a parallel CPU code implemented using MPI
   - programming effort required

Performance is our main driven criteria. In order to accomplish the best possible speedup, the following steps were performed:

- Specify all the variables that should be transferred between host and accelerator.
- Determine the compute regions that should be offloaded to the accelerator, depending on computation intensity.
- Apply the necessary optimizations needed to improve the performance.
- Tune the generated code.
- Compare the speedup gained against a full socket MPI implementation on a multi-core CPU platform.
- Repeat the previous steps as needed to achieve the desired performance.

The workflow of these steps is depicted in Figure 1. Table 1 gives a detailed information about the different software/hardware used as our evaluation platform.

The remaining of this paper is organized as follows: Section 2 gives some hints about the previous work. We describe the context, the algorithm and the CPU implementation of the modeling and RTM applications in Section 3. An overview of the GPU architecture and OpenACC is given in Section 4. We explain our OpenACC and CPU implementations and optimization analysis in Section 5. Section 6 demonstrates our evaluation results and performance analysis. In Section 7, we conclude the work presented and give some hints about our future ideas.

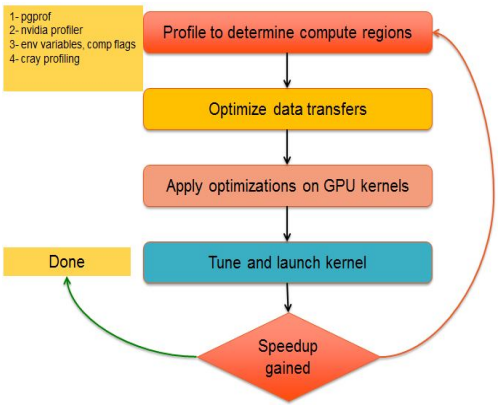

Figure 1: A workflow diagram showing how to accelerate applications

Table 1: Evaluation Platform

Software/Hardware	Facts
CPU/CRAY	Two Intel 10 core Xeon Ivy Bridge E5-2680 v2 @2.8 GHz, total 20 cores/64GB memory per node.
CPU/IBM	Two Intel Quad-core Xeon E5640 @2.8 GHz, 12MB L3 Caches, total 8 cores/24GB memory per node
GPU1	Nvidia Tesla Kepler K40 on CPU platform with 2880 cuda cores, 12 GB GDDR5 Memory, ECC: disabled.
GPU2	Nvidia Tesla Fermi M2090 on CPU platform with dedicated PCIe2x16 per GPU 512 cuda cores, 6 GB GDDR5 Memory, ECC: disabled
API and Languages	OpenACC 2.0, MPICH 3.1, FORTRAN 2003
Compilers	PGI Compiler 13.7 / 14.3 / 14.6, CRAY Compiler 8.2.6, Nvidia CUDA 5.0 / 5.5

2. RELATED WORK

Directive-based GPU programming models [12], [10], [8] proposed a simpler approach, compared with GPU programming languages, to handle GPU usage across many platforms. The directive-based approach was first proposed by OpenMP [2], which is a standard API for shared memory programming. OpenMP provides a directive-based programming approach for generating parallel versions of programs from sequential ones.

Reverse Time Migration (RTM) is a seismic depth imaging method for velocity analysis to better analyze the structure of the subsurface. RTM was introduced and presented by a

number of authors that include Baysal et al. [3], McMechan [6] among others. On the other hand, seismic modeling [5] is an essential geographical data processing technique for numerically simulating seismic wave propagation. Several attempts were previously conducted to port RTM and seismic modeling to GPUs. Abdelkhalek et al. [1] employed CUDA to solve the acoustic wave equation in RTM and seismic modeling on GPU. Ghosh et al. [9] had experiences in analyzing the performance and programmability of three high-level directive-based GPU programming models - PGI, HMPP and OpenACC on an Nvidia GPU against Isotropic (ISO)/Tilted Transversely Isotropic (TTI) finite difference kernels. A prediction-based performance tuning mechanism was proposed in [13] to tune OpenACC gang and vector clauses. This proposed tuning methodology was applied to the isotropic modeling kernel to dynamically adapt to the execution environment on a given system.

In contrast with the aforementioned experiences, the work reported here, to the best of our knowledge, is the first to target porting RTM and seismic modeling on GPU using OpenACC for solving the isotropic, acoustic, and elastic wave equation.

3. SEISMIC IMAGING

The efficiency of a seismic imaging technique can be determined via three primary concerns. These concerns include the accuracy of determination of the true subsurface medium, the quality of the produced image, and having a clear understanding of how waves propagate through the media. Among the numerous seismic imaging approaches, seismic modeling and RTM are two applications of particular interest.

3.1 Seismic Modeling

Numerical seismic modeling is a keystone tool for simulating wave propagation in the earth. It mainly forms the forwarding phase of RTM in which a seismic wave is propagated from the source to the subsurface. This useful technique predicts the seismograms that can be recorded by a set of sensors in order to give an assumption for the structure of the subsurface. To solve the wave equation by direct methods, which are used in our applications, the geological model is approximated by a numerical mesh, that is, the model is discretized in a finite numbers of points (25-point stencil in our propagators). As a consequence, we can model or simulate seismic wave propagation as a movie snapshots traveling through the earth.

3.2 Reverse Time Migration (RTM)

RTM is a seismic migration method for creating images in complex wave propagation fields. The source and receivers wave-fields in RTM are respectively propagated forward and backward in time. RTM then uses the well established imaging condition $I(z, x, y)$ of cross correlation between the forward propagated source wave-field $S(z, x, y, t)$ and the backward propagated receiver wave-field $R(z, x, y, t)$ summed over the sources s where z, x, y, and t denote depth, horizontal, lateral axis, and time respectively. Figure 2 depicts how the seismic image is created by combing the wavefields from source(s) and receiver(s) through imaging condition.

3.3 Propagators and Equations

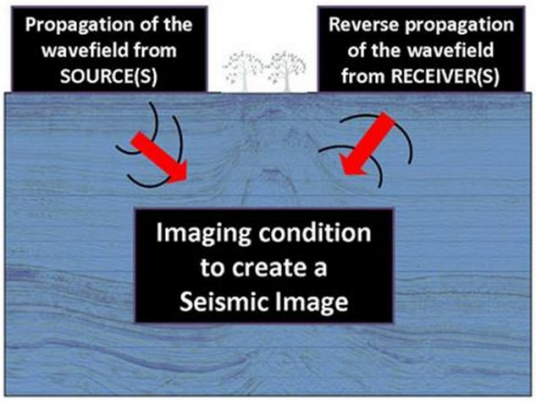

Figure 2: Seismic image creation through imaging condition

There are three basic formulations to represent the earth models. These formulations are purely isotropic or acoustic, isotropic elastic, and anisotropic. In our experiments, we focused on the first two formulations. However, we will consider the anisotropic case in the future. The following sections give more details about what we have implemented.

3.3.1 Isotropic

Isotropic acoustic models are based on the assumption that density and instantaneous velocity are the only physical parameters defining wave propagation. Hence, only fluids can be described by this model. This model has been the most prevalent as propagation in such environments can be simulated efficiently. A semi-analytic form of the isotropic propagator in constant density domain, where we only discretize in time, can be written as seen in Equation 1, where u is the wave-field, v_p is the pressure velocity, and f is the point source of propagation. A 2nd order 25-point stencil is used.

$$\begin{aligned} &u^{-1} = u^0 = 0, \quad u^{n+1} - Qu^n + u^{n-1} = \Delta t^2 v_p^2 f^n, \\ &Q = 2 + \Delta t^2 v_p^2 \nabla^2, \quad 0 \leq n \leq N-1. \end{aligned} \tag{1}$$

3.3.2 Acoustic

The variable density domain acoustic propagator (here in 2D) is implemented as a 25-point stencil staggered grid first order system. Equation 2 describes this model, where ρ is the density of fluid, p is the pressure in fluid, q_x is the velocity flow of fluid in the x-direction, and q_z is the velocity flow of fluid in the z-direction.

$$\begin{aligned} \partial_t p &= \rho v_p^2 \left(\partial_x q_x + \partial_z q_z\right) + \rho v_p^2 \partial_t^{-1} f\left(x_s, t\right) \\ \partial_t q_x &= \frac{1}{\rho} \partial_x p \\ \partial_t q_z &= \frac{1}{\rho} \partial_z p \end{aligned} \tag{2}$$

As in the constant domain case, this model assumes that the earth is fluid and hence it is not very accurate in some cases.

3.3.3 Elastic

Isotropic elastic models are described by density, compressional velocity, and shear velocity. The waves are characterized by particle motions (normal to the direction of propagation in the case of compressional waves and tangential in

the case of shear waves). Equation 3 represents the velocity stress formulation of the elastic wave equation. It consists of three particle velocity equations and six stress equations, where ρ is density, σ represents the stress wave-field, and v represents the particle velocity. This Equation is implemented as a first order system in a staggered grid.

$$\begin{aligned}
&\partial_t v_\omega = \frac{1}{\rho}\left(\partial_x \sigma_{x\omega} + \partial_y \sigma_{y\omega} + \partial_z \sigma_{z\omega}\right), \quad \omega = x, y, \text{ or } z,\\
&\partial_t \sigma_{xx} = \left(\lambda + 2\mu\right)\partial_x v_x + \lambda\left(\partial_y v_y + \partial_z v_z\right),\\
&\partial_t \sigma_{xy} = \mu\left(\partial_x v_y + \partial_y v_x\right), \quad \text{where } xy = xy, xz, \text{ or } yz.
\end{aligned} \tag{3}$$

This model assumes that the earth is a solid medium, which makes it more accurate compared with the acoustic model. However, it is more complicated and computationally intensive.

The staggered grid approach used in our acoustic (variable domain) and elastic models has the advantage of accuracy with less computational effort because it allows a larger grid size.

4. AN OVERVIEW OF GPU ARCHITECTURE AND OPENACC

GPU, which indicates General Purpose GPU in this paper, has a different architecture than CPU. A CPU consists of cores optimized for sequential serial processing while a GPU has thousands of smaller and more efficient cores designed for handling multiple tasks simultaneously. Nvidia GPU cores are constructed into Streaming Multiprocessors (SMs). GPU has a complex memory hierarchy that includes global memory, shared memory, texture (read-only) memory and L1 cache, and registers distributed among different threads. The Kepler architecture focuses on energy efficiency with more advanced capabilities compared with Fermi, as seen in Table 2.

The desired and sometimes conflicting goals associated with using the OpenACC directive programming model intend to enhance productivity, portability, and performance. While this abstract programming model needs less effort/time to produce optimized codes targeting multiple architectures, understanding the architecture and memory hierarchy of GPU is crucial for a programmer to reduce access latency and maximize coalesced GPU memory accesses. The performance obtained still does not reach what can be achieved using CUDA or OpenCL. CUDA in particular, since we are using Nvidia cards, provides a low-level programming approach, which offers more flexibility to exploit the available potentials of GPUs.

OpenACC is an open pragma-based GPU directives standard, which was first announced by Nvidia in Supercomputing 2011 conference. OpenACC is a joint effort of CAPS enterprise, CRAY, PGI, and Nvidia. The OpenACC specification 1.0 was the first version to be supported by these vendors. The OpenACC specification 2.0 was released later with some new features, but still not fully implemented in both PGI and CRAY compilers. OpenACC concepts are inherited from the PGI accelerator model [14]. OpenACC execution model exposes three levels of parallelism via gang, worker and vector parallelism to better control the distribution of iterations among cores and to tune the code. For GPU mapping, a gang is a block which is executed by one Streaming Multiprocessor (SM). Worker represents a warp/group of threads and vector is represented as the threads in a warp. OpenACC offers two types of compute directives: *parallel* and *kernels*. With the *parallel* construct, a gang-redundant mode is exhibited, which means that if no loop directive is specified inside the parallel region, the encapsulated code will redundantly be executed by all gangs. On the other hand, the *kernels* construct produces a sequence of accelerator kernels, where each loop nest becomes a kernel.

5. OPENACC IMPLEMENTATIONS AND OPTIMIZATION ANALYSIS

Our CPU implementation of seismic modeling and RTM employs finite difference direct methods to solve the isotropic, acoustic, and elastic wave equation. We use operators with a 3D stencil width of 8. Discretizations in space and time occur along the forward and backward propagation to compute numerical solutions to the wave equation. The discretization process depends on the formulation used to describe the earth model as explained in the previous section. At each grid point, the stencil (25 data read accesses are performed) is applied to compute one grid point of the updated wave-field. Since seismic modeling represents the forwarding phase of RTM, we decided to have one algorithm explaining the overall process. Algorithm 1 describes the parallel CPU implementation we use as a reference to evaluate our OpenACC accelerated solution. This implementation is based on domain decomposition where each domain may be divided into sub-domains mapped onto several hosts to fit into memory and to decrease simulation time. Ghost nodes are exchanged via MPI non-blocking standard send (MPI_ISEND) and receive (MPI_IRECV). When all required sends and receives are posted, the communication request handles are then immediately checked for completion via corresponding number of MPI_WAITANY calls. Ghost node thickness is determined by the stencil used to solve the wave equation.

We step forward in the modeling phase to exchange boundaries, compute the updated source wave-field for all grid points, and perform source injection. The snap_period value depends on the maximum frequency used in the attached velocity model. A partial snapshot is saved after each snap_period and the cumulative propagated snapshot is then displayed as the output of seismic modeling. In RTM, this snapshot should be read in the backward phase to generate the final seismic image.

In practice, it is necessary to truncate the computational domain. The truncation is usually done by introducing a boundary layer that absorbs the outgoing waves. The Perfectly Matched Layer was introduced by Berenger [4] and has since been the most used absorbing boundary with different formulations. The standard PML is used in our second order (isotropic) formulation of the wave equation. One major problem with the standard PML is that the boundary layer does not absorb evanescent waves where the PML method suffers from large spurious reflections. Instead, we use the Convolutional PML (C-PML) method to simulate wave propagation in acoustic (variable density) and elastic medias by storing four different one-dimensional arrays with the cpml-coefficients for each dimension (x,y,z etc). A 2D case snapshot of wave propagation resulted from the modeling phase in acoustic media is depicted in Figure 3.

Imaging in RTM is represented by the backward phase as demonstrated in Algorithm 1. RTM simulates the source

wave-field in both forward and backward phases and the receivers wave-field. The imaging condition is applied in the backward phase to compute the cross-correlation between forward source and backward receiver wave-fields over the time iterations. A 2D seismic image in acoustic media for RTM is shown in Figure 5

Input: $wavefields$,$velocity\_model$
Output: $snapshot$, $seismic\_image$
for $time \leftarrow t_{start}$ **to** t_{end} **do**
  exchange_boundaries;
  forall the *domain grid points* **do**
    forward time step source wave-field (2D/3D);
  end
  source_injection;
  foreach *snap_period* **do**
    save_snapshot($time$);
  end
end
for $time \leftarrow t_{end}$ **to** t_{start} **do**
  foreach *snap_period* **do**
    read saved snapshot($time$);
    apply imaging condition;
  end
  exchange_boundaries;
  forall the *domain grid points* **do**
    backward time step receiver wave-field (2D/3D);
  end
  receiver_injection;
end

Algorithm 1: Reverse Time Migration and Modeling algorithm

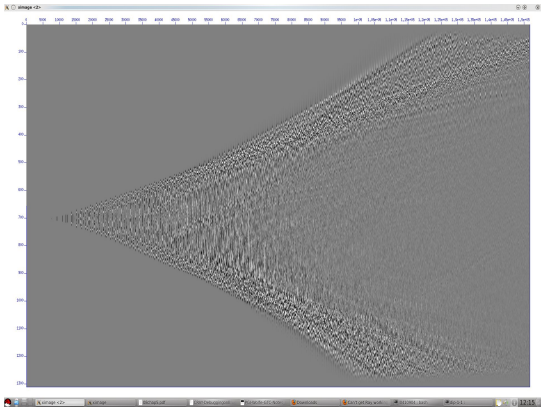

Figure 3: A 2D seismic modeling snapshot in acoustic media

5.1 OpenACC Implementation of RTM

Any OpenACC implementation should handle three main issues: host computations, accelerator computations and data allocation, and communications between host and accelerator. Our implementation targets exploiting the full potentials of the GPU by optimizing the intensive computation kernels that should be ported to GPUs, restructuring the code, and minimizing data transfers between host and device.

Our OpenACC implementation consists of five consecutive steps as described in Figure 4. These steps are as follows:

1. Data allocation: Due to GPU global memory constraints in both Kepler and Fermi cards, we found that the forward and backward wave-field variables of RTM cannot be allocated at the same time on GPU. Nvidia System Management Interface program (nvidiaâĹŠsmi) provided the required guidance in this matter. Using more than one *data* directive prevents data in the GPU global memory from being persistent across different kernel launches. Our solution was to exploit the new feature provided in OpenACC 2.0 specs by using the (ACC ENTER DATA COPYIN) (ACC EXIT DATA DELETE) after host allocation and before host de-allocation respectively. In the first step, the forward phase variables were moved to the GPU memory. A MACRO was used to automate the process of choosing the variables according to the used seismic model.

2. Forward phase: The forward propagation of the wave-field in time is performed in the GPU memory by logically swapping t_n and t_{n+1} arrays. Only the ghost nodes need to be exchanged between host and GPU at each time step when partitioning the domain among several GPUs. Exchanging only ghost nodes (partial transfers) instead of the whole domain, despite requiring more programming effort, significantly reduces the amount of data exchange and thus computing time. However, exchanging non-contiguous data remains a non-optimal solution. One workaround is rearranging data of these ghost nodes by performing a transposition on GPU. Our focus was on exploiting one GPU, and hence we did not try to optimize multiple GPUs. A branch condition was needed to ensure that the host snapshot data will not be updated at each time step. Instead, it is only updated after each snap_period.

3. Offloading forward and uploading backward: In order to save the global GPU memory, the modeling data is offloaded from the GPU, except for the forward wave-field, which is needed in the later steps. The imaging data needed in backward propagation and imaging condition is then uploaded to the GPU.

4. Backward phase: Computing the backward time step receiver wave-field on the GPU was suffering from bank conflicts in shared memory due to a lack of coalescing memory accesses. Coalescing memory accesses provided an initial solution to the problem. However, code restructuring was needed to gain a better performance. The better optimized kernel, which is used in the modeling phase and benefits from vectorization, was called instead by passing the backward wave-field data to it and the ghost node exchanges were rearranged to imitate the modeling phase. This technique showed a 3x performance speedup over the original RTM code in both acoustic and elastic models. The isotropic kernel used in both phases was the same, and hence it did not suffer from this issue. The imaging condition can be applied either on the GPU or the CPU. Our experiments demonstrated a better performance when the final seismic image was computed on the GPU instead of the CPU. However, this advantage was insignificant due to two main factors: 1) The overheads resulted

from transferring the image back to the CPU. 2) The lack of intensive computations that can benefit from the power of the GPU.

5. Storing image and offloading data from the GPU: The decision of applying imaging condition on the GPU requires sending the image back to host. Once the final seismic image is computed, the data is no longer needed on the GPU. The *PRESENT* clause was used in all kernels as the required data is persistent on the GPU across the different kernel launches.

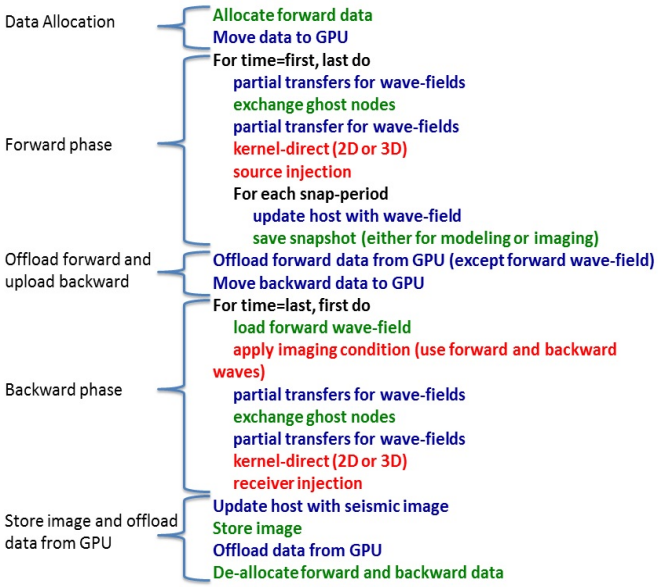

Figure 4: Reverse Time Migration and Modeling OpenACC implementation

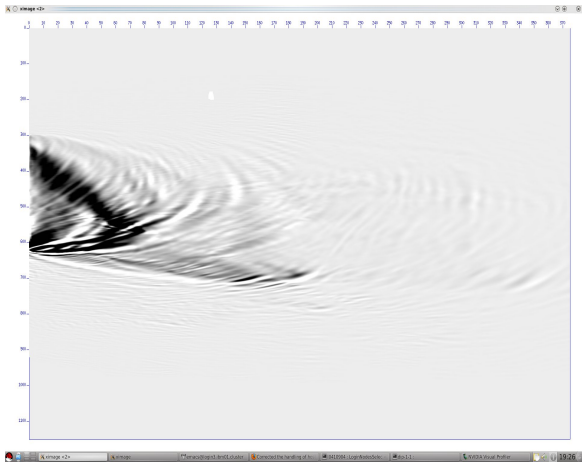

Figure 5: A 2D seismic image in acoustic media for RTM

5.2 Compiler Optimizations and Loop Scheduling

Different compilers apply different optimization techniques associated with OpenACC directives and clauses. Loop scheduling clauses, provided by OpenACC, determine how loops are mapped to the GPU architecture. However, compilers vary in terms of implementing these clauses. Based on our experiments, we found that the most efficient way to offload perfectly nested loops, which is the case in our acoustic and elastic models, relies significantly on the compiler itself. Using PGI, it was more efficient to use the *kernels* directive to allow the compiler to handle the existing worksharing among loops without using the gang/worker/vector paradigm. To ensure a good coalescing, the generator assumes that the cache-friendly loop is the innermost. Therefore, our 3D loop nest case led to the collapsing of the 2 innermost loops to generate a 2D grid of hardware accelerator threads. We also used the independent clause to specify that no dependencies exist among loop iterations, which facilitates the compiler's task to parallelize the code. Using CRAY, it was totally different, i.e., the more information you pass to the compiler, the better performance you get. Using the gang/worker/vector paradigm associated with the parallel directive gave the best performance. Having the 3D case as an example, if no loop scheduling is provided with the *parallel* directive, the CRAY compiler uses gangs on the outer i-loop, but then analyzes the j and k loops to determine which loop looks most profitable to be vectorized. The compiler may actually look at the i-loop as well depending on whether this loop-level has work to be done. Which loop gets vectorized is completely dependent on the code inside the loop. In our experiments with the CRAY compiler, vectorizing the innermost loop explicitly improved mapping the parallelism of the acoustic and elastic models to the accelerator. A performance comparison between kernels and parallel implementations using CRAY is depicted in Figure 8 and Figure 9 for the 2D and 3D acoustic models respectively.

The main kernel in our isotropic code suffered from the if-statements, which are needed to compute PML at the boundaries of the grid domain only. These if-statements reduce GPU utilization and prevent efficient gridification. We implemented two approaches to make the loop levels perfectly nested in this kernel, and hence to improve gridification and loop scheduling. The first approach was to remove these if-conditions by changing the loop indices and restructuring the loop region accordingly. The loop independent scheduling in PGI triggers gridification in kernels regions, and 2D gridification requires perfectly nested loops. This approach significantly enhances the performance using PGI 14.3 compiler as shown in Figure 7. However, PGI 14.6, shown in Figure 6, did not give the same improvement. Our second approach was to compute PML everywhere in the grid domain. Although this change increases computations, it was more efficient than the original code with PGI 14.3, but not with PGI 14.6. CUDA 5.0 is the default version that was used with PGI 14.3, while CUDA 5.5 is the default with 14.6. The CUDA version used affects GPU code generation and justifies performance variation.

Occupancy (number of concurrently running threads) is another metric of performance. When register spilling is an issue, increasing the limit of registers per thread might reduce occupancy, which potentially reduces execution efficiency. However, this may still be an overall win due to fewer total bytes being accessed. The best number of registers per thread was found to be 64 in all implemented cases on both Fermi and Kepler GPU cards. This number gives the required balance between occupancy and number of accessed bytes. Figure 10 shows a performance comparison between

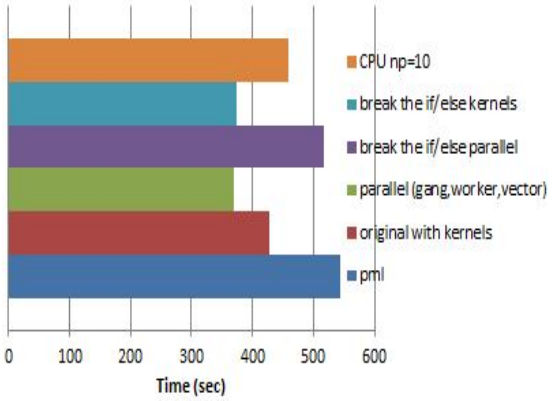

Figure 6: ISO Modeling 3D (PGI 14.6)

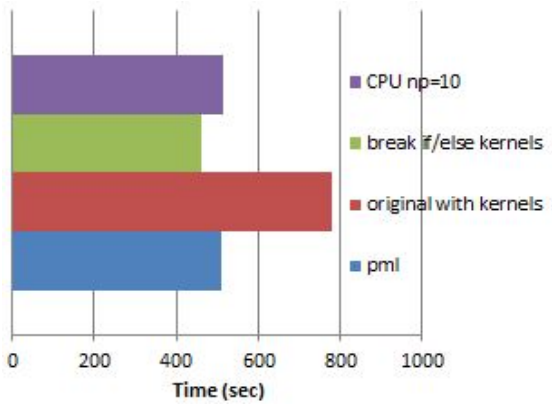

Figure 7: ISO Modeling 3D (PGI 14.3)

different numbers of registers for the 3D elastic modeling case.

Another interesting case was to execute multiple kernels concurrently using the *async* clause assuming that there are no dependencies among them, which is the case in our elastic model. There are three limiting factors to async: 1) There is a specific number of async streams, one of which is used by the implementation for automatic use. 2) The PCIe bus can only move so much at a time. 3) The device can only hold so much at a time. The async on *parallel* and *kernels* directives is useful to let the CPU queue up the next work unit and therefore CRAY compiler uses *auto_async_kernels* as its compilation default. Having said that, using different async streams with different kernels does not necessarily improve performance. PGI compilers gave a worst performance on both Fermi and Kepler when *async* was used to overlap GPU kernels with each other. On the contrary, using *async* with CRAY compiler reduces the execution time by 30%. Referring to Figure 11, we realized that overlapping GPU kernels is very hard to be accomplished in our elastic case, as the available streaming multiprocessors are occupied by one or few kernels. However, using multiple streams can lead to small jobs packing on to the device all at once and it can lead to reduced lag time between kernel launches. The 30% improvement was due to this reason.

The best compilation strategy used with the PGI compiler was ($ta = nvidia : pin, ptxinfo, maxregcount : 64 - Minfo = accel, loop, opt, mp$) on both Kepler and Fermi.

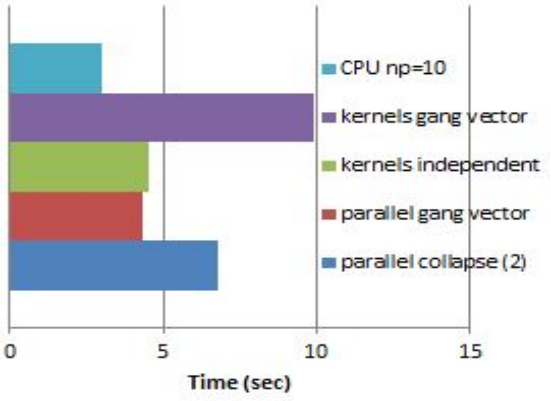

Figure 8: Acoustic Modeling 2D (CRAY Compiler)

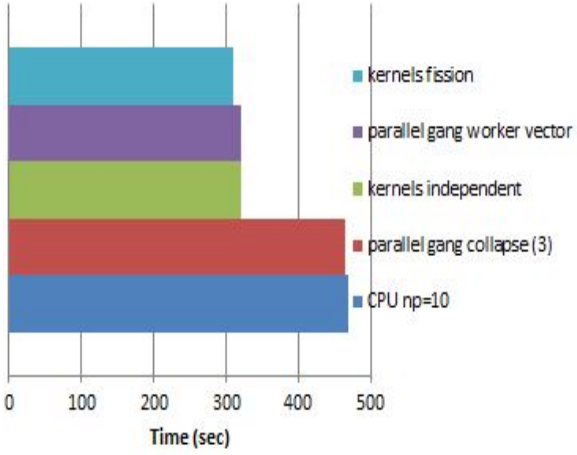

Figure 9: Acoustic Modeling 3D (CRAY Compiler)

We used the pin option to avoid the cost of transfers between pageable and pinned host arrays by directly allocating our host arrays in pinned memory and move them to device memory.

5.3 Loop Transformations

Code transformations are critical to improve loop mapping and GPU code generation. We have considered various transformation techniques that include, but not limited to inlining, permutation, fission, transposition, tiling, and collapsing. Due to space limitations, we will discuss the two techniques, which were most significant in our implementations.

Starting with loop fission, the most intensive 3D acoustic kernel, as depicted in Figure 12, consists of computations that handle wave-fields derivations over three dimensions for all grid points. We simply break this kernel into three kernels where each is responsible for one dimension. A 3x speedup was gained after applying loop fission when this code was executed on M2090 Fermi card. As a justification, the original loop needs too many registers and this may prevent effective execution. Most of the register pressure we were getting with the original code was with the array address variables (the compiler will use local temp variables to store the offsets into the various multi-dimensional arrays. Loop fission overcomes these issues, and hence improves per-

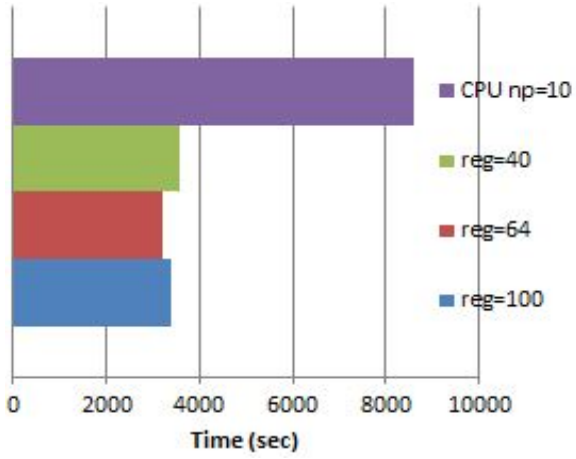

Figure 10: Elastic Modeling 3D (registers per thread)

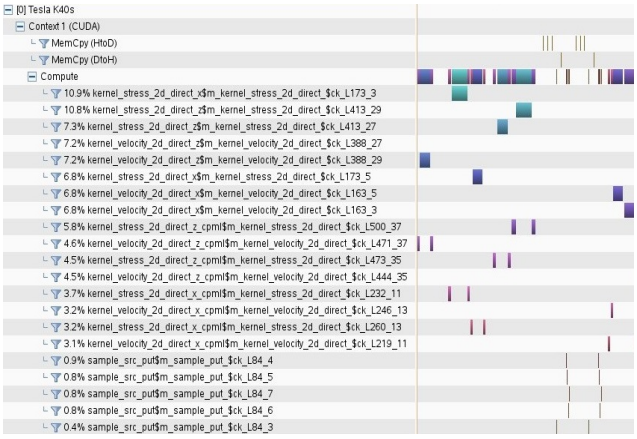

Figure 11: Nvidia Profile for a 2D Elastic Model (Async) on CRAY

formance when Fermi card is used. That was not the case though on Kepler card, as the register per thread count is doubled with 255 registers per thread. This increase allows K40 card to handle the pressure generated from the original loop.

Coalescing memory accesses on the GPU is essential for any OpenACC application to efficiently move data from global memory into shared memory and registers. Each time a location is accessed, many consecutive locations, including the requested location, are accessed. Recalling that all threads in a warp execute the same instruction, the most favorable global memory access is achieved when the same instruction for all threads in a warp accesses consecutive memory locations. In this favorable case, the hardware coalesces all memory accesses into a consolidated access to consecutive global memory locations. The generator assumes that the best loop for cache utilization is the innermost to ensure a good coalescing. However, this is not the case in our acoustic and elastic models. Taking the 2D acoustic backward phase kernel, as depicted in Figure 13, the inner *ix* loop is not parallelized due to loop carried dependencies. We solved that in three steps. The first step was to transpose the original array by creating a new temporary array on GPU. The second step was to replace the original array

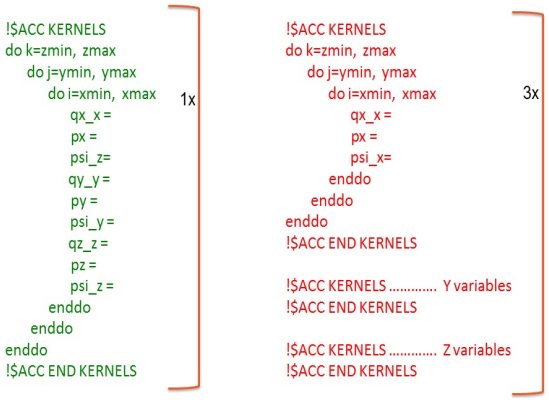

Figure 12: Loop fission optimization (Acoustic 3D)

with the temporary one inside the kernel. Having done that, the inner loop has no dependencies any more and should not suffer from a coalescing problem. The last step was transposing the array back and removing the temporary array from GPU. This technique allows us to gain a 3x speedup compared with the original code on both GPU cards using PGI and CRAY compilers.

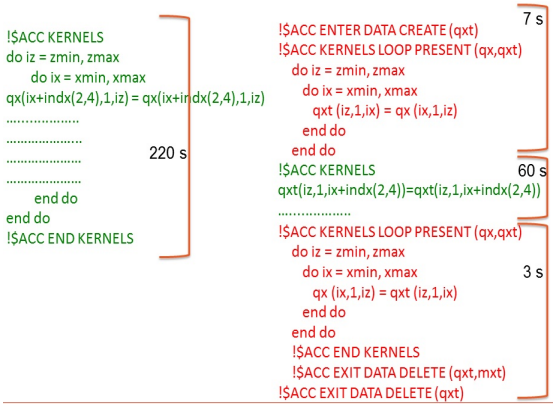

Figure 13: Coalescing of memory accesses (Acoustic 2D)

5.4 Reducing Data Transfers

(ENTER DATA COPYIN) and (EXIT DATA DELETE) directives were used after host allocation and before host de-allocation respectively to ensure the persistence of the required data on GPU across different kernel launches. The forward phase data and backward phase data were handled separately to save GPU memory. (UPDATE HOST)and (UPDATE DEVICE) directives were used to manage data transfers between host and GPU. All intensive computations were ported to GPU using either *kernels* or *parallel* directives. *pgprof* tool from PGI was used to determine these intensive regions. The *present* clause was associated with all GPU kernels to assist the compiler to determine the variables that are already present in GPU memory and to reduce data transfers across different kernels. In some cases, the *create* clause was used to create temporary variables needed in a single kernel, such as the transposition case mentioned earlier in this section.

Recalling our OpenACC implementation of RTM in Fig-

ure 4, source injection, receiver injection, and applying imaging condition were optionally computed on GPU to reduce data transfers and to improve performance. Source and receiver injection code was called as a function inside a one dimensional loop. In the case of adding a receiver term, the loop iterates over the number of receivers provided in the model. We decided to port both injections to avoid updating the host with the wave-field at each time step. However, as seen in the Nvidia profile for the 2D isotropic model in Figure 14, GPU utilization of source injection is 0.04%, due to lack of computations. For receiver injection, we performed inlining to let the *kernels* directive encapsulate the loop iterating over receivers. GPU utilization was much better 26%. Regarding the imaging condition, two kernels (one is to compute the wave-field at odd time steps and the other is at even time steps) were ported as depicted in Figure 15. Again, GPU utilization in both cases was very low 1.9%. However, the performance was slightly better than computing the seismic image on CPU because there was no need to update the host with the source wave-field anymore. GPU utilization of the main kernel as depicted in Figure 14 and Figure 15 was almost the same, because this kernel is not affected by applying the imaging condition. The acoustic and elastic models exhibited a similar behavior to the isotropic model regarding the imaging condition.

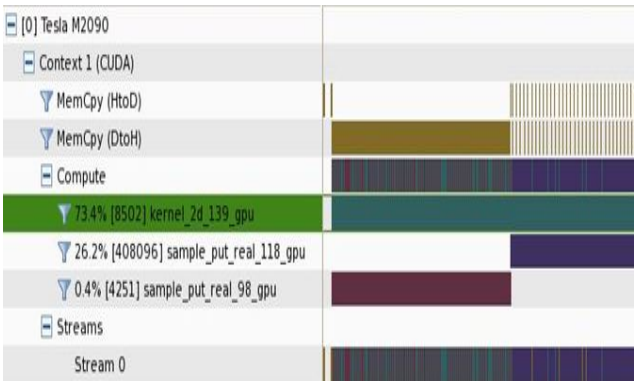

Figure 14: Nvidia Profile while image is on CPU (Isotropic 2D)

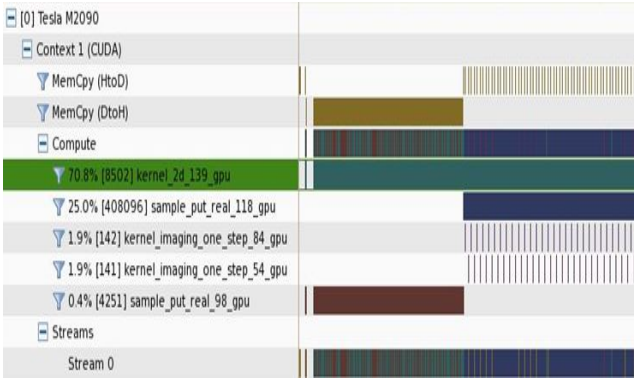

Figure 15: Nvidia Profile while image is on GPU (Isotropic 2D)

6. EVALUATION RESULTS

In our experiments, we have used two Nvidia GPU cards representing different accelerator generations to compare their performance against our seismic image applications. The first GPU card is Fermi M2090, which is built on top of an IBM cluster. The second card is Kepler K40, which is built on top of an XC30 CRAY supercomputer. Table 2 shows the specifications of each GPU card and the attached CPU cluster. All computations were carried out in single precision for both the CPU reference implementation and the GPU implementation. The reference CPU total time is the time to process the entire domain while using sub-domain decomposition. It is given by running a full socket MPI implementation using the PGI compiler, and the GPU reference time is given by using one GPU without using sub-domain decomposition. On CRAY, the full socket MPI implementation consists of 10 cores and was executed using PGI 14.6 compiler with CUDA 5.5, which is the default for this compiler version. On IBM, the full socket MPI implementation consists of 8 cores and was executed using PGI 14.3 compiler with CUDA 5.0, which is the default for this compiler version. CUDA 5.0/5.5 employ LLVM compiler front-end and can link multiple object files into one program. With Kepler cards, GPU can generate work for itself, in contrast with the previous CUDA versions where GPU work can only be generated by CPU.

The reference CPU kernel time for RTM compromises both the forward and backward propagation kernels. In seismic modeling, the reference CPU kernel time consists of the forward propagation kernel. Our performance metric was the ratio of the execution time of the CPU-based implementation to that of the GPU-based one for a one shot profile. The GPU-based implementation represents the best optimized version of each seismic case. As mentioned in the previous section, different optimizations were applied to each seismic case before obtaining the timing and speedup measurements. The best optimization technique also differs from one compiler to the other. Nvidia profiler was the main tool used to analyze our performance measurements.

Table 2: GPU cards specs and attached CPU platforms

Card	GFLOPS (SP)	Bandwidth	Memory	#cores	CPU
M2090	1331.2	180 GB/sec	6GB	512	Two Intel Quad-core Xeon E5640 @2.8 GHz, 12MB L3 Caches, total 8 cores/24GB memory per node
K40	4291	288 GB/sec	12GB	2880	Two Intel 10 core Xeon Ivy Bridge E5-2680 v2 @2.8 GHz, total 20 cores/64GB memory per node.

6.1 Modeling

Table 3 illustrates the timing and speedup measurements obtained for seismic modeling for solving the isotropic, acoustic, and elastic wave equations in both two-dimensional and three-dimensional domains. In general, the execution time obtained while using PGI was lower than that obtained with CRAY. The reference CPU execution time was obtained using the PGI compiler. Our GPU CRAY implementation can still be optimized though. We found that the CPU/GPU time ratio decreases while increasing the number of subdomains because more CPU parallelism is achieved while communication time becomes more predominant. The elas-

Table 3: Seismic modeling timing and speedup measurements

	CRAY cluster								IBM cluster			
	Total GPU time(s)		Total speedup		Kernels time(s)		Kernels speedup		Total GPU time(s)	Total speedup	Kernels time(s)	Kernels speedup
Model	CRAY	PGI	CRAY vs 10	PGI vs 10	CRAY	PGI	CRAY vs 10	PGI vs 10	PGI	PGI vs 8	PGI	PGI vs 8
ISOSTROPIC 2D	2.3	1.4	0.6	1	1.6	1	0.7	1.1	2	2	1.5	2.3
ACOUSITC 2D	4.1	3.2	0.7	0.9	3.4	2.7	0.9	1.1	5	1.3	4.4	1.2
ELASTIC 2D	7	4.5	0.9	1.2	6.6	4.3	0.7	1.1	7	1.9	4.8	2.4
ISOSTROPIC 3D	460	365	1	1.3	365	285	0.9	1.2	448	1.2	385	1.0
ACOUSITC 3D	310	235	1.5	2	220	155	1.2	1.7	260	2.3	200	2.3
ELASTIC 3D	4000	3200	2.1	2.7	3100	2700	2.4	2.7	x	x	x	x

tic variables could not fit in GPU memory when Fermi card was used. The best speedup (2.7x) was achieved with the elastic model since it is the most computationally intensive case. On the contrary, the isotropic model gave the worst speedup because it is a memory-bound application, which exhibits inefficient GPU utilization. Due to avoiding CPU-GPU communication overheads, Kernel speedup was better than total speedup in all implementations.

The total GPU time gained on CRAY cluster with Kepler card was slightly better than what was gained on IBM with Fermi card. The speedup range of (1.1x-1.5x) in all cases is still far from the optimal capacity that should be achieved by using Kepler over Fermi. The CUDA version used on CRAY was 5.5, while CUDA 5.0 was used on IBM. This may have a big impact on GPU code generation, and hence may significantly affect performance.

The full socket MPI implementations of both isotropic and acoustic models in the three-dimensional domain produce similar execution times. Yet, porting these two models to GPU produce different execution times. The total GPU time of the acoustic model was almost 2x faster than the isotropic model. This emphasizes the big variation in the architectures of CPU and GPU and illustrates the differences in handling memory/compute bound applications by the GPU architecture and OpenACC programming model.

6.2 Migration

The Cray XC30 supercomputer integrates a novel intercommunications technology that provides improvements on all of the network performance metrics: bandwidth, latency, message rate, etc. This makes our CPU implementation run much faster on CRAY, compared with the old interconnection technology provided by the IBM cluster. This justifies the higher speedup rates on IBM, compared with CRAY. Table 4 shows the timing and speedup measurements obtained for RTM for solving the isotropic, acoustic, and elastic wave equations in both two-dimensional and three-dimensional domains.. A speedup of 10.8x was obtained on IBM for the acoustic model in 3D domain, in contrast with the speedup obtained on CRAY for the same model, which was 1.3x. The RTM implementation involves more CPU/GPU communications than the modeling. This significantly reduces the obtained CPU/GPU time ratio.

Seismic models with variable density domain (acoustic and elastic) demonstrate low communication times compared with the computation time, and hence the overall time is slightly impacted. On the contrary, the isotropic case requires many host-GPU updates within the (enter data/exit data) region to keep the variables consistent on both host and GPU. The three-dimensional cases showed better speedup measurements compared with the two-dimensional cases due to better GPU utilization. The best GPU utilization achieved in the 2D cases was around 70% for the most intensive compute kernel, in contrast with 90% in the 3D cases.

One of the major issues encountered in the RTM CPU implementation was the receiver injection. Adding receiver term has to be done on GPU to avoid updating the host with the wave-field at each time step. However, iterating over the available receivers will launch the receiver term on GPU for (#receivers×#timesteps) times, as explained in Algorithm 1. To avoid that, we inlined the function, in which the receiver term was added, to have one kernel encapsulating all receivers. Inlining was successfully processed by the CRAY compiler, but could not be processed by the PGI compiler. This justifies the improvement of CRAY measurements over PGI in RTM. Inlining was crucial to avoid kernels launches overheads, but could not solve the loop carried dependencies between the different receivers. This significantly hurts the performance, especially in the 2D seismic cases. Having said that, a future solution that might be interesting is to integrate the receiver injection into the backward time step kernel or to separate the source injection from the receiver injection and compute the receiver injection on CPU instead of GPU.

6.3 Constraints, Limitations, and Suggestions

In the following, the major constraints, associated with using the OpenACC directive approach in seismic imaging, are summarized. Some suggestions are also presented.

- How to explicitly use shared memory for specific variables is still a bottleneck. The tile and cache features are not working properly in both CRAY and PGI.
- The routine directive does not support multiple levels of subroutine parallelism.
- Loop mapping, which describes how loop nests are converted into GPU threads, should be more efficient.
- Performance portability is an issue (parallel and Kernels directives demonstrate different performance from one compiler to another).
- Applying code transformations is critical to optimize GPU code generation. However, this prevents maintaining a single version of code that can still run efficiently on CPU. Providing new OpenACC directives to apply code transformations is a necessity.
- Overlapping kernels execution on GPU can dramatically enhance performance. OpenACC current asynchronous operations efficiently overlap computations with data transfers. However, this is not the case when multiple kernels can benefit from concurrent execution on GPU.

Table 4: RTM timing and speedup measurements

	CRAY cluster								IBM cluster			
	Total GPU time(s)		Total speedup		Kernels time(s)		Kernels speedup		Total GPU time(s)	Total speedup	Kernels time(s)	Kernels speedup
Model	CRAY	PGI	CRAY vs 10	PGI vs 10	CRAY	PGI	CRAY vs 10	PGI vs 10	PGI	PGI vs 8	PGI	PGI vs 8
ISOSTROPIC 2D	8.5	14	0.4	0.2	2	2.3	1.2	1	11.5	0.5	4	1.3
ACOUSITC 2D	12.2	16	1.2	0.9	4.5	5.6	2.4	2	19	5.3	9	7.9
ELASTIC 2D	20	23	0.8	0.7	7	8	1.7	1.5	30	1.1	12	2.3
ISOSTROPIC 3D	1600	1500	0.6	0.6	600	550	1.1	1.2	1200	0.9	800	1.1
ACOUSITC 3D	870	765	1.1	1.3	320	310	1.3	1.3	530	10.2	400	10.8
ELASTIC 3D	x	15000	x	1.3	x	6000	x	2.9	x	x	x	x

- Minimizing memory traffic has a great impact on performance. Different techniques, similar to CPU, can be added to the OpenACC specification to reduce access latency such as thread affinization and placement, and cache blocking.

7. CONCLUSION AND FUTURE WORK

In this work, we presented and discussed our experiences in evaluating and analyzing the impact of using GPU technology on seismic imaging via the OpenACC directive-based standard model. We developed OpenACC versions for 12 different seismic cases using seismic modeling and Reverse Time Migration (RTM) algorithms. We employed OpenACC to exploit the powerful capacity of two Nvidia GPU cards: 1) M2090 and 2) K40, residing in IBM and CRAY clusters respectively. Our OpenACC implementations for the different models showed different speedup behaviors depending on the implied intensity of computations, the compiler used, and the applied GPU/CPU technology. The best speedup results were 3x on CRAY and 10.2x on IBM. The innovative network technology provided in CRAY XC30 enhances communications across distributed memory systems and among multiple GPUs. However, the restriction of using multiple asynchronous streams is still a bottleneck that prevents efficient utilization on a single GPU. Code restructuring seemed crucial to enhance GPU utilization. Understanding GPU register pressure management and the architecture led to better exploitation of OpenACC features. The lack of enough computations justified the speedup measurements and GPU utilization obtained in two-dimensional domains. Our experiments emphasized that OpenACC solutions can still benefit from a margin of improvement in order to be effectively used from a seismic imaging industrial point of view.

Path forward, we believe that exploiting multiple GPUs will provide powerful insights. Consequently, overlapping MPI communications with GPU computations could improve performance, especially when larger grid dimensions are used. Last but not the least, our experiments exposed some inefficiencies in the implementations of RTM and seismic modeling applications that affect GPU utilization through OpenACC. Some solutions were applied in this work to overcome these inefficiencies and some others will be used to enhance the usage of OpenACC in seismic imaging.

8. ACKNOWLEDGMENTS

The authors would like to thank the PGI group support team and the CRAY compiler support team for their collaboration in this work. We would also like to thank Sunita Chandrasekaran from the HPCTools group at University of Houston for her feedback on the paper. Many thanks go to TOTAL for providing the computing resources.

9. REFERENCES

[1] R. Abdelkhalek, H. Calandra, O. Coulaud, J. Roman, and G. Latu. Fast Seismic Modeling and Reverse Time Migration on A GPU Cluster. In *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*, pages 36–43. IEEE, 2009.

[2] O. ARB. The OpenMP API Specification for Parallel Programming. `http://openmp.org/wp/`, 2014.

[3] E. Baysal, D. D. Kosloff, and J. W. Sherwood. Reverse Time Migration. *Geophysics*, 48(11):1514–1524, 1983.

[4] J.-P. Berenger. A Perfectly Matched Layer for the Absorption of Electromagnetic Waves. *Journal of computational physics*, 114(2):185–200, 1994.

[5] Carcione, Jose M and Herman, Gérard C and Ten Kroode, APE. Seismic modeling. *Geophysics*, 67(4):1304–1325, 2002.

[6] W.-F. Chang and G. A. McMechan. Reverse-time Migration of Offset Vertical Seismic Profiling Data Using the Excitation-time Imaging Condition. *Geophysics*, 51(1):67–84, 1986.

[7] N. CUDA. Compute Unified Device Architecture Programming Guide. 2007.

[8] C. entreprise. HMPP Directives. `https://www.olcf.ornl.gov/wp-content/uploads/2012/02/HMPPWorkbench-3.0_HMPP_Directives_ReferenceManual.pdf`, Nov. 2010.

[9] S. Ghosh, T. Liao, H. Calandra, and B. M. Chapman. Experiences with OpenMP, PGI, HMPP and OpenACC Directives on ISO/TTI Kernels. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 691–700. IEEE, 2012.

[10] T. P. Group. PGI Accelerator Programming Model for Fortran & C. `http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.3.pdf`, 2010.

[11] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg. *OpenCL Programming Guide*. Pearson Education, 2011.

[12] OpenACC-Standard.org. The OpenACC Application Programming Interface. `http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf`, 2013.

[13] S. Siddiqui and S. Feki. Predictive Performance Tuning of OpenACC Accelerated Applications. In *Supercomputing*, page 511. Springer, 2014.

[14] M. Wolfe. Implementing the PGI Accelerator Model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 43–50. ACM, 2010.