High-Performance Reverse Time Migration on GPU

Javier Cabezas*, Mauricio Araya-Polo†, Isaac Gelado‡, Nacho Navarro‡, Enric Morancho‡ and José M. Cela†‡
*Computer Sciences - Programming Models
Barcelona Supercomputing Center (BSC), Barcelona, Spain
Email: javier.cabezas@bsc.es
†Computer Applications in Science and Engineering dept.
Barcelona Supercomputing Center (BSC), Barcelona, Spain
Email: mauricio.araya@bsc.es
‡Department of Computer Architecture, Universitat Politècnica de Catalunya, Spain

***Abstract*—Partial Differential Equations (PDE) are the heart of most simulations in many scientific fields, from Fluid Mechanics to Astrophysics. One the most popular mathematical schemes to solve a PDE is Finite Difference (FD). In this work we map a PDE-FD algorithm called Reverse Time Migration to a GPU using CUDA. This seismic imaging (Geophysics) algorithm is widely used in the oil industry. GPUs are natural contenders in the aftermath of the clock race, in particular for High-performance Computing (HPC). Due to GPU characteristics, the parallelism paradigm shifts from the classical threads plus SIMD to Single Program Multiple Data (SPMD). The NVIDIA GTX 280 implementation outperforms homogeneous CPUs up to 9x (Intel Harpertown E5420) and up to 14x (IBM PPC 970). These preliminary results confirm that GPUs are a real option for HPC, from performance to programmability.**

***Keywords*-GPU, high-performance computing, finite differences, PDE, seismic imaging, stencil computation**

I. Introduction

The clock race is over, and for the last four years the only clear trend points to parallel architectures. Within the HPC community, a myriad of new architectures are presented as replacements for old hardware. The new architectures rely on heterogeneous or homogeneous multi-cores, or combinations of both. Lately, heterogeneous multi-cores (Cell/B.E., GPUs and FPGAs) are attracting attention, mainly due to their exceptional performance (e.g. the RoadRunner 1 petaFlops system), and the potential save in energy consumption. However, many issues have to be considered and evaluated before adopting these new architectures, from packaging to programmability.

In this paper, we analyze some of these issues by porting an algorithm used in geophysics to a couple of traditional multi-core architectures and to a GPU based architecture. The algorithm is called Reverse Time Migration (RTM). It is a wavefield reconstruction method that helps to delineate subsurface structures and is particularly useful for oil prospection. Oil companies trust this kind of methods enough to rely on them for crucial multi-million-dollar decisions like whether and where to start drilling operations.

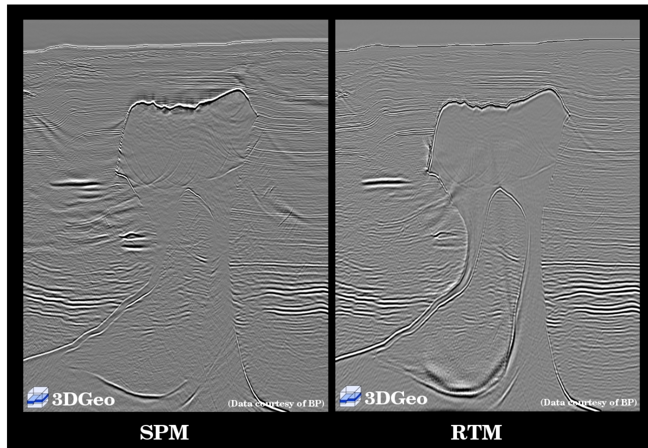


Figure 1. A side-by-side comparison between the results of two seismic imaging techniques, Shot Profile Migration (SPM) and RTM. As we see, RTM provides higher quality seismic images, with better signal-to-noise ratio and clearer structure delineation

When it comes to complex geophysic scenarios like reserves located beneath salt, large velocity contrasts or steeply dipping formations (as can be observed in Figure 1), RTM's imaging quality advantages other methods [1]. Unfortunately, its computational cost hinders its adoption. This fact motivates constant research in architectures/algorithms that can provide the required computational power.

RTM is composed by several tasks, as we explain in the following sections. In order to have a realistic use case we have implemented most of the tasks of the algorithm, not only the 3D finite difference computation which is considered by other works ([2], [3]) to calculate the peak performance of the algorithm. Therefore, having a real-world application and not only a synthetic kernel, we are able to evaluate aforementioned issues like programmability or full program performance.

GPUs have evolved from fixed hardware targeting graphics processing to fully programmable massive parallel machines. Data parallelism exists in many HPC applications and, thus, GPUs are an ideal platform to efficiently execute

them. GPUs API used to be graphics oriented and required a tedious mapping from algorithms to graphics primitives. With the release of CUDA a general purpose programming model for GPUs that enables a simple mapping of HPC applications to current GPUs is provided. We consider that data parallelism present in many parts of the RTM algorithm can be easily exploited using the CUDA programming model. Although, lack of advanced optimization tools hinders the fully exploitation of the performance.

We present an optimized implementation of the RTM that is specifically designed to exploit the architectural characteristics of a GPU: we have parallelized the workload into loosely coupled threads to exploit the multiple independent processing elements, orchestrated the data transfers to ensure the most efficient memory bandwidth utilization, employed loop unrolling, and other techniques that return high performance. We provide a comparison against a reference HPC platform that employs traditional cache-coherent cores. To do so, we have developed an equally optimized implementation of the RTM for IBM JS21 (PowerPC) blades and SGI Altix blades (Intel Xeon) (detailed in Section IV).

The two main contributions of this work are: having complete RTM algorithm that uses GPUs and delivers more performance per watt that the implementation in current homogeneous architectures, and an analysis on the programmability of each system that concludes that it is not an issue for GPGPU.

The remainder of this paper is organized as follows: Section II introduces the basics of the RTM algorithm and its main computational kernel. In Section III we introduce the arguments behind our decision to try the GPGPU way. Sections IV and Section V present respectively the results of developing/porting RTM to a traditional multi-core HPC platforms and to a GPGPU platform (NVIDIA GTX 280 based system). Section VI evaluates and compares the performance achieved by the two solutions. Finally, Section VII concludes the paper.

II. The problem

In this work, we choose as test case a well known geophysics technique, RTM, which is a classic example of what we call `PDE+FD` kind of problems, where a Partial Differential Equation (PDE) is solved using a Finite Difference (FD) scheme [4] (as in [5] for the electromagnetic case). We have the acoustic wave equation as:

$$\frac{1}{\rho c^2}\frac{\partial^2 u}{\partial t^2}(\mathbf{x},t) - \nabla\cdot\big(\frac{1}{\rho(\mathbf{x})}\nabla u(\mathbf{x},t)\big) = f_s(\mathbf{x},t) \qquad (1)$$

The unknown u represents the pressure, the function c the wave velocity, ρ the density and f_s the source term. We assume an isotropic, non-elastic medium, where density is not variable, thus yielding the following expression:

$$\frac{\partial^2 u(\mathbf{x},t)}{\partial t^2} \;-\; c^2\Delta u(\mathbf{x},t) \;=\; f(\mathbf{x},t) \qquad (2)$$

RTM is based on a two-way acoustic wave PDE. This implies solving two times the acoustic wave equation (Equation 2), called forward and backward propagations. The FD solver *kernel* (step 3 of Alg. 2) computes the stencil and the time integration for every iteration of the time dependent loop. The kernel is the most computational demanding segment of the RTM. Furthermore, RTM includes the following tasks: the source wave introduction (step 6 of forward propagation), the receivers traces introduction (step 6 of the backward propagation), and the absorbing boundary conditions (ABC) computation (step 9 for both ways). Finally, in every step of the backward propagation the correlation of the forward and backward wavefields is carried out (step 12 of the backward sweep).

Forward propagation	Backward propagation
input: velocity model shot location output: forward wavefields	input: velocity model, receivers' traces, forward wavefields output: image
1: **for all** time steps **do**	1: **for all** time steps **do**
2: **for all** main grid **do**	2: **for all** main grid **do**
3: compute wavefield	3: compute wavefield
4: **end for**	4: **end for**
5: **for all** source location **do**	5: **for all** receivers location **do**
6: add source wavelet	6: add receivers data
7: **end for**	7: **end for**
8: **for all** ABC area **do**	8: **for all** ABC area **do**
9: apply ABC	9: apply ABC
10: **end for**	10: **end for**
11: **for all** main grid **do**	11: **for all** main grid **do**
12: store forward wavefield	12: correlate wavefields, restored forward and backward
13: **end for**	13: **end for**
14: **end for**	14: **end for**

Figure 2. The RTM Algorithm

The computational intensive steps in Alg. 2 are: the kernel (step 3), and the ABC computation (step 9). Step 12 in both, forward and backward, is mainly intensive in Input/Output (I/O), but due to the temporal dependency among iterations, becomes a bottleneck to keep in mind. These tree steps set the performance bound for RTM. In order of importance, the kernel and then the I/O are our main concerns.

In terms of performance evaluation, we have only to care about the efficiency of a single iteration, due to the fact that all of them are alike. Nevertheless, in real-life executions of RTM the number of iterations is determined by the acquisition set up, but generally they are in the order of ten thousand. Therefore, the parameter that influence performance the most is the size of the computational domain (the grid). Our computational domain is composed by two parts, an external crust of n points (ABC area) around the main grid, which is a 3D parallelepiped ($nz \times nx \times ny$), where

$nz \approx nx \approx ny \gg n$. This means that bigger the grid lesser demanding is the ABC computation, but this also implies reduced I/O performance, notice that $nz \approx nx \approx ny \leq 1000$ points are common place.

III. Motivation

Graphics Processing Units (GPUs) are coprocessors designed to speed-up the execution of graphical applications, such as 3D rendering or video decoding. First GPUs were non-programmable hardware that executed a limited set of graphics operations (e.g. OpenGL). However, current generation of GPUs are designed in a more flexible way, allowing programmers to actually execute general purpose code. Programming models such as Compute Unified Device Architecture (CUDA) currently allow to use popular programming languages, such as C/C++ or FORTRAN, to write code to be executed by GPUs. Programmable GPUs, jointly to programmer-friendly programming models, enable us to use GPUs to perform general purpose computations on GPUs. The programming model offered by CUDA to implement data parallel computation (see Section V) perfectly matches the geometry of RTM, allowing a straightforward mapping of RTM to GPUs.

Most graphics computations exhibit massive amounts of data-parallelism: multiple instances of the code can be executed concurrently on different data. Due to the nature of graphics algorithms, GPUs are embarrassingly parallel architectures, formed by hundreds of cores (e.g. 240 cores in the NVIDIA GTX 280) and designed to maximize the instruction throughput. This in contrast with traditional general purpose processors, whose key design goal is to minimize instruction latency. For instance, the NVIDIA GTX 280 [6] easily outperforms the Intel Xeon E7450 processor [7] by 10 times for data-parallel computations. However, the E7450 achieves a factor of 10 higher performance over the GTX 280 for sequential control-intensive code. RTM and many other applications are composed by both data parallel and sequential control-intensive phases. Hence, general purpose CPUs and GPUs can be coupled together to form heterogeneous computing systems [8] that efficiently execute different application phases.

GPUs are world-wide available and have been included in most desktop, server and high-end computers sold during the last ten years. The huge amount of GPUs that are sold all around the world allow GPU vendors to offer relative small prices when compared to high-performance computing solutions. GPUs offer a performance per $ figure that it is about one order of magnitude higher than high-end CPUs. This allows building small-size and cheap GPU clusters that offer huge peak performance for massive data parallel codes, such as RTM.

Programmable GPUs and the CUDA programming model enabled us to do a full port of the RTM algorithm to a CUDA-capable GPU in a short period of time. Data parallel phases of RTM benefit from the impressive peak performance of GPUs for massively data parallel code while control-intensive parts can still be executed by high-performance CPUs.

IV. RTM on Homogeneous Architectures

To set a fair comparison between the GPGPU-based RTM implementation and some traditional cache-coherent multi-core platforms, these platforms have to be based on commodity hardware available in an HPC oriented configuration. Among the platform that satisfy these requirements, we choose: the IBM BladeCenter JS21 Type 8844 blade and the SGI Altix XE320. The JS21 blade sports two double-core PowerPC 970MP processors running at 2.3 GHz, these cores employ coherent L1 and L2 cache memories. In the Altix XE320 blade we find two "Harpertown" processors; this quad-core Xeon E5460 runs at 2.5 Ghz. These blades are off-the-shelf products, and they are actively employed in supercomputers (e.g., MareNostrum [9]). Detailed hardware specifications are reported in Table I.

We have ported RTM to these homogeneous platform, and then optimized the implementation to the same degree.

Blade	IBM JS21 Type 8844	SGI Altix XE320
Processors	PowerPC 970MP	Xeon E5460
Sockets x cores	2×2	2×4
Memory per blade (Gbytes)	8	8
Clock Frequency (GHz)	2.3	2.5
Peak throughput (GFlops/s)	36.8	80.0
SIMD registers (per core)	80	N/A
SIMD width	128 bit	128 bit
Main memory standard	DDR2	DDR3
Cache memory		
L1 (data + instr)	32Kb + 64Kb	32Kb + 32Kb
L2	1Mb per core	6Mb per pair
L3	N/A	8 Mb

Table I
TECHNICAL SPECIFICATIONS OF THE HOMOGENEOUS SYSTEMS EMPLOYED IN OUR EXPERIMENTS

As mentioned in Section II , the kernel is the most computational demanding task of the RTM. The computational weight of the kernel is due to the low number of operations it performs per each data point [10]. The PDE+FD solver uses a 8-point (per axis) stencil, depicted in Figure 6(left). Data are stored in Z-major form (see Figure 4), therefore accesses across the X and Y axes may be significantly more expensive in a cache-based architecture. The following paragraphs will focus on the kernel optimizations: memory accesses and computation.

Memory access patterns have been shown [11] to be critical for the performance of the stencil computation, heart of the RTM kernel. Without optimizations, the accesses in lines 5, 7 and 8 (Figure 3) cause heavy cache thrashing because u3, u2, u1 and v are much larger than the L2 cache, and the L2 cache has limited associativity. We tackle

```
input: dt2, C00, Z1...Z4, X1...X4, Y1...Y4, u2, u1
output: u3

 1: for  y = 4 ... Y - 4  do
 2:     for  x = 4 ... X - 4  do
 3:         for  z = 4 ... Z - 4  do
 4:             /* Stencil computation */
 5:             u3[z,x,y] = C00 · u2[z,x,y] +
                                 Y4 · (u2[z,x,(y-4)] + u2[z,x,(y+4)]) +
                                 ...
                                 X4 · (u2[z,(x-4),y] + u2[z,(x+4),y]) +
                                 ...
                                 Z4 · (u2[(z-4),x,y] + u2[(z+4),x,y]) +
                                 ...
                                 Z1 · (u2[(z-1),x,y] + u2[(z+1),x,y]);
 6:             /* Integration over time */
 7:             u3[z,x,y] = v[z,x,y] · v[z,x,y] · u3[z,x,y];
 8:             u3[z,x,y] = dt2 · u3[z,x,y] + 2 · u2[z,x,y] - u1[z,x,y];
 9:         end for
10:     end for
11: end for
```

Figure 3. Pseudo-code of the unoptimized PDE+FD solver invoked in line 3 of RTM (see Figure 2). Z, X, Y are the dimensions of the data set. Z1...Z4, X1...X4, Y1...Y4 and C00 are spatial discretization parameters, dt2 is a temporal discretization parameter. Integrating the equation requires maintaining the wavefield of at least 2 earlier time steps (u2 and u1), while u3 is the current wave field

that problem with *blocking* [12], [13]. To apply blocking, in particular the Rivera strategy, the original 3D space is divided in slices, where the X axis of the slices has a size that optimally fits the cache hierarchy. Figure 4 shows this decomposition.

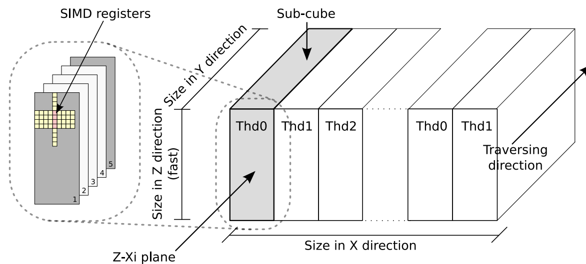
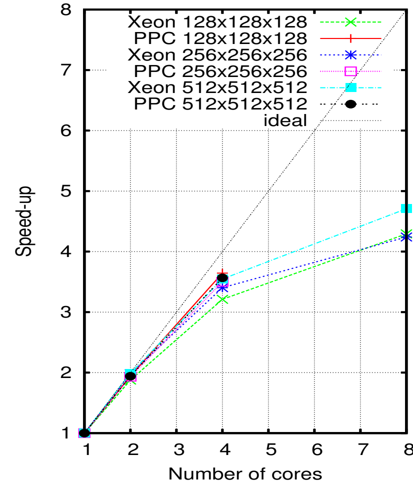


Figure 4. The blocking strategy we adopt in our homogeneous implementations.

As the optimization of computation is concerned, we exploit all the forms of parallelism provided by the architecture: the thread-level parallelism provided by the multiple cores, and the data-level parallelism provided by the SIMD instruction set. We use all the independent threads (4 or 8) per blade with a parallelization strategy that follows the blocking. Each core processes its assigned set of slices independently. Since each core has its own L2 cache, interference among threads is minimal. Our implementation employs OpenMP [14], and thanks to careful loop management, we provide to OpenMP with more opportunities for scheduling, thus enhancing scalability (Figure 5).



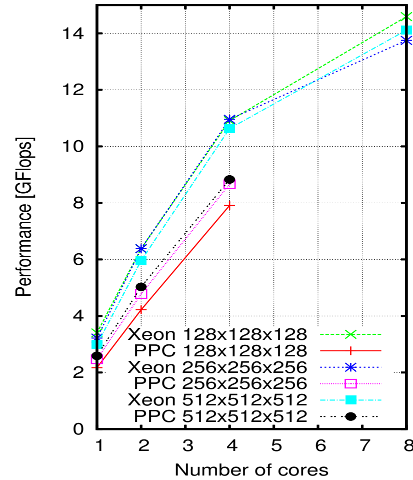


Figure 5. Our RTM algorithm enjoys a good scalability (up to 4 cores in both architectures), and better performance (up to 14.6 GFlops). The Xeon version does not scale linearly up to 8 cores, this is due to memory associativity problems, solution searching underway

To exploit data-level parallelism, our code uses the Altivec/VMX SIMD instruction set available in the PPC970MP. SIMD instructions allow to process 4 single-precision floating-point operands per cycle (Figure 6). The processor features a relatively large number SIMD registers (80), so that loop unrolling is used in conjunction with SIMDization to extract more performance from the application. On the Xeon processor case, we did not exploit explicitly the SSE instructions, thus we relied on the compiler flags that automatically take advantage of them.

The ABC task suffer from the same aforementioned

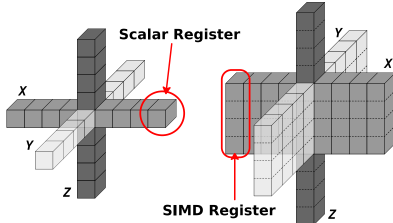


Figure 6. Visual comparison between a scalar (left) and a SIMD stencil (right) in our RTM computational kernel

problems, further the memory access is even more penalized, this is because the data to be accessed is barely continuous in memory. Also, the data reuse is very low, the number of instructions required to compute the ABC is of the order of 1/5 of the PDE solver. Fortunately, the mentioned optimizations helps to mitigate the ABC computation burden. Besides, the relative importance of it diminish when the main computational domain increases.

The RTM algorithm requires huge data sets to be transfer to/from disk. In fact, for every iteration of the forward propagation a wavefield is stored, the wavefields can easily be as big as 1Gb. Thus, we face two I/O interrelated problems: space (disk) and time (transfer). Firstly, the wavefield size is reduced previous to be stored by means of data compression. We are able to compress the wavefields from 4x up to 10x, with minimal data loss. Secondly, this compressed wavefield is stored in asynchronous fashion, thus the transfer time is overlapped with the computation time of the rest of the tasks of the algorithm.

The techniques employed in our RTM implementation on the homogeneous architectures provide us with an advanced starting point for our GPGPU port, which is the main subject of the following section.

V. RTM ON GPGPU

A. NVIDIA's Tesla architecture

NVIDIA's Tesla architecture, introduced in the GeForce G80 GPU family, enables developing high-performance parallel applications that take profit of NVIDIA hardware in the C/C++ language by using the Compute Unified Device Architecture (CUDA) programming model. Thus, previous knowledge on graphics programming is not required anymore.

As shown in Figure 7, a GPU is presented as a set of streaming multiprocessors, each with its own processors and shared memory (user-managed cache). The processors are fully capable of executing integer and single precision floating point arithmetic, with additional cores used for double-precision. However, all the stream processors in a multiprocessor share the fetch, control and memory units. Therefore each SM may be better conceptualized as a 8-wide vector processor. All multiprocessors have access to the global device memory, which is not cached by the hardware.

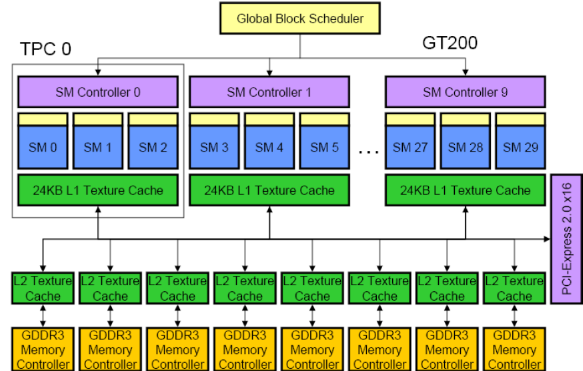


Figure 7. NVIDIA G200 Architecture

Memory latency is hidden by executing thousands of threads concurrently. Register and shared memory resources are partitioned among the currently executing threads. There are two major differences between CPU and GPU threads. First, context switching between threads is essentially free state does not have to be stored/restored because GPU resources are partitioned. Second, while CPUs execute efficiently when the number of threads per core is small (often one or two), GPUs achieve high performance when thousands of threads execute concurrently.

B. CUDA Programming model

CUDA arranges threads into thread-blocks. All threads in a thread-block can read and write any shared memory location assigned to that thread-block. Consequently, threads within a thread-block can communicate via shared memory, or use shared memory as a user-managed cache since shared memory latency is two orders of magnitude lower than that of global memory. A barrier primitive is provided so that all threads in a thread-block can synchronize their execution.

C. RTM implementation

We used a simple RTM implementation for homogeneous CPUs with no optimizations as the base of the port.

All the computation steps both in the forward and backward phase access and modify the same data. Consequently, a mixed CPU/GPU implementation such as the one in [3] require continuous memory transfers between the host and the GPU memories in order to keep the data coherent. However, as we explain in Section III, GPU's performance is hugely penalized by frequent and large memory transfers. Therefore we have implemented all the aforementioned steps in GPU kernels that access data which resides in the GDDRAM. The host code only orchestrates the execution environment, the kernels' invocations, and the I/O transfers to/from disk whenever it is necessary.

1) Program initialization: We use the CUDA driver API instead of the run-time API because it lets us manage CUDA contexts thus allowing the use of POSIX threads for doing

things like asynchronous accesses to disk in parallel to the computation. A unique context is created in order to allow all host threads to have access to all the data structures that reside in the GPU.

Some constant values are stored in the constant memory. We try to maximize the use of the constant memory as it is cached and leaves more available registers for the compiler kernel code generation.

The volumes used during the stencil computation are zeroed in the GPU memory and the velocity model is transferred from the host memory. The traces gathered by the receivers are also transfered to the device memory although this is done at the beginning of the backward phase.

2) Kernel implementation:

3D stencil computation: Global memory is not implicitly cached by the hardware. Furthermore, accesses to global memory are very expensive. Therefore, optimizing global memory bandwidth usage is a must in order to get good performance from the GPU. A k-order stencil computation calculates the value of each point by using the $k/2$ neighbor elements in each direction. That is $3k+1$ reads per calculated element. Micikevicius [2] refers to this number as read redundancy. The same concept is also applied to writes. The sum of them is called overall redundancy. We used this metric to analyze global memory accesses and improve memory bandwidth usage.

For the 3D stencil computation kernel, we use the 2D sliding window approach proposed in [2], also well known from [12]. Shared memory is used to store a $(n+k)*(m+k)$ tile which holds the elements of the z and x dimensions of the wavefield. Each thread loads an element into the shared memory. Moreover, since threads load elements that are consecutive in memory, they can be coalesced in order to maximize the global memory bandwidth. Accesses to neighbor elements for these dimensions are then fetched from shared memory. The kernel iterates on the y dimension; thus, each thread computes a column along this dimension. Neighbor elements for the y dimension are stored in k registers of each thread which behave like a queue: every time a tile is done the oldest element is popped out and a new element is enqueued. Using this approach we obtain a read redundancy of 3 and a write redundancy of 1 for $16x16$ tiles.

ABC: Wave propagates through the simulated medium and is reflected on the wavefield boundaries. This adds lots of noise to the final image. In order to avoid this undesired effect, a wave attenuation is performed. We have implemented a different kernel for each face of the 3D wavefield as they require different memory access patterns. The approach is the same as the one used for the stencil computation: there is a front of threads that traverses the points that belong to the ABC area and update them accordingly. In this case, the dimension which is traversed depends on the face that is being computed.

Shot insertion: This computation is very simple and can be run in the host. However, since the wavefields reside in the GPU memory, additional memory transfers must be performed in order to keep the data coherent. Thus, we have implemented a simple kernel that implements this functionality in the GPU and does not block the next computation steps.

Receivers' data insertion: The backwards phase of the RTM algorithm requires exciting the medium with the data previously gathered by the receivers. The algorithm is the one used for the shot insertion so the code has been reused. Furthermore, there can be up to thousands of receivers. Thus, we take profit of the GPU parallelism and calculate all of them in parallel.

3) I/O issues: During the backward phase, a correlation between the forward and backward wavefields must be performed every n time steps (where $n < 10$ for an accurate result). That implies that the forward wavefields for these time steps have to be accessible in the backward phase. Since that information takes hundreds of gigabytes of memory, it must be stored in an external storage system like a hard drive during the forward phase. In the case of GPUs an additional transfer between the GDDR and the host memory is necessary, as the data cannot be directly transferred to an external device (only possible with *peer DMA*, which is not supported in current GPUs). In order to minimize the performance impact of this data transfer, two measures have been adopted:

- Data compression: data is compressed in the GPU memory before being transferred to the host memory and the to disk. We achieve a 8x size reduction at the expense of a new computational kernel that takes 1/10th of the stencil computation time.
- Asynchronous I/O (A/IO): we use `librt` to program asynchronous transfers to external devices. This allows overlapping the transfer and the next computation steps.

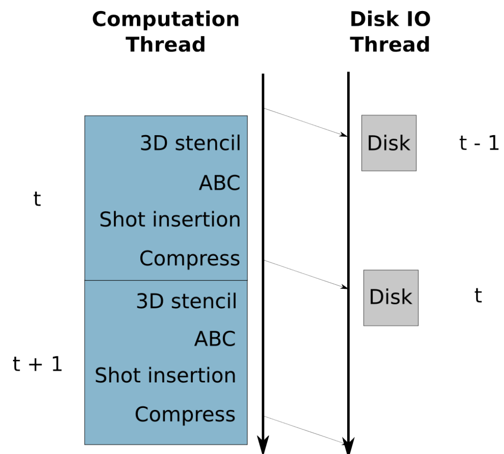


Figure 8. Using a dedicated thread for disk I/O can be overlapped with the computation of the next timestep

Withal, some issues are detected in this implementation. First, the transfer from the GDDRAM to the host memory is still performed synchronously (thus blocking the next computation step). Second, the Linux asynchronous API shows a random behavior across executions, with calls that immediately return (thus achieving a perfect overlapping with computation) and calls that block for an amount of time similar to 20 to 30 computational steps. The solution is to have a dedicated thread to perform the I/O transfers asynchronously (from the computation thread's perspective). This way, computation can continue without waiting for the memory transfer from the GDDRAM to the host memory and the synchronous API (which shows a much more consistent behavior) can be used to transfer data to the external storage system (Figure 8). In the backward phase the forward wavefields are read, transferred and decompressed in the same manner.

4) Memory usage:

Forward: The 3D stencil computation and the boundary conditions steps only need the current wavefield volume. However, the velocity model and the two previous wavefields are also required for the time integration. Additionally a fifth volume is necessary to store the image illumination. The size of a wavefield volume depends on the dimensions of the field and need an additional *ghost area* in each dimension to calculate the 3D stencil (Equation 3). The velocity model and the illumination volume, on the other hand, do not need the ghost area (Equation 4).

$$v(z,x,y,s) = (z+2*s)*(x+2*s)+(y+2*s)*4 \quad (3)$$

$$v'(z,x,y) = z*x*y*4 \quad (4)$$

The compression buffer used for the correlation step must be also taken into account (Equation 5).

$$c(z,x,y,s) = \frac{v(z,x,t,s)}{8} \quad (5)$$

$$f(z,x,y,s) = 3*v(z,x,y,s)+2*v'(z,x,y)+c(z,x,y,s) \quad (6)$$

Backward: The three wavefields and the velocity model are needed again to solve the Partial Differential Equation. The illumination volume is replaced by another one used to perform the correlation that contains the wavefields previously stored in disk during the forward phase. Additionally, the information of the receivers (Equation 7) is also stored in the GDDRAM.

$$rcv(t) = (9+t)*4 \quad (7)$$

$$g(z,x,y,s,r,t) = f(z,x,y,s)+r*rcv(t) \quad (8)$$

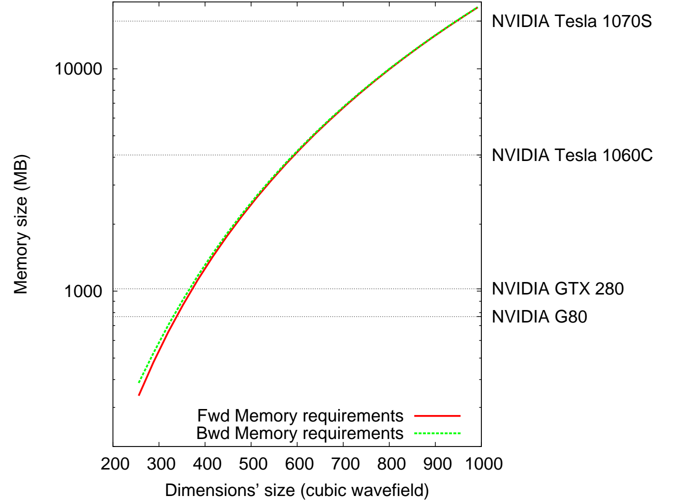


Figure 9. Memory footprint of the data structures used by RTM, for different wavefield (cubic) dimensions. Numbers computed for a number of `timesteps` = 2500 and 5000 receivers

Summary: RTM requires a huge amount of data in order to work with the wavefield dimensions used in real-world problems. As we can see in Figure 9 that the forward and backward phases a similar amount of memory as the size needed to store the data from the receivers depends on the number of simulated timesteps and not on the wavefield size. We also see that even the latest NVIDIA HPC devices cannot handle shots of sizes greater than 1000x1000x1000. In this paper, we use wavefields that fit entirely in the GPU GDDRAM to test our implementation. An implementation that uses multiple GPU devices and/or uses other techniques to compute full-sized problems is left as future work.

VI. Performance Evaluation

In this section we provide incremental performance results, from kernel to a complete RTM execution. For sake of clarity, we discuss separately three distinguishable benchmarks: kernel, I/O and RTM as a whole. The experiments setup is common for all selected architectures, this in terms of number of steps, computational domain sizes and number of experiments repetitions. The latter is particular important to avoid spurious operating system events.

A. Kernel Benchmarking

The following results are focused on the performance of the computation of the PDE solver (basically stencil and time integration), and ABC computation. This section reported execution times do not include: I/O time and initialization delays.

As can be observed in Tables II and III, the GPU based version of the RTM kernel outperform the homogeneous versions, up to 14x against the PowerPC, and up to 9x against the Xeon.

Algorithm	PowerPC Time [s]	Xeon Time [s]	GTX 280 Time [s]	GPU gain per iteration
kernel	0.228	0.146	0.016	14.3, 9.0
kernel + ABC	0.301	0.183	0.026	11.6, 7.1

Table II
PERFORMANCE RESULTS FOR THE RTM KERNEL IMPLEMENTATIONS, IN TERMS OF ITERATION ELAPSED TIME. EXPERIMENTS WERE CARRIED OUT FOR A 352^3 DATA SET

Algorithm	PowerPC Perform. [GFlops]	Xeon Perform. [GFlops]	GTX 280 Perform. [GFlops]	GPU gain
kernel	8.01	12.53	113.35	14.1, 9.0
kernel + ABC	6.02	10.17	69.39	11.6, 7.1

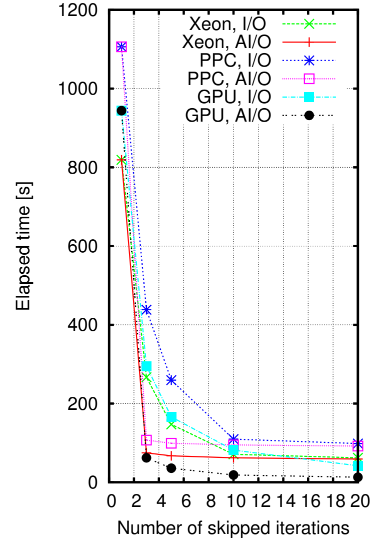
Table III
PERFORMANCE RESULTS FOR THE RTM KERNEL IMPLEMENTATIONS, IN TERMS OF ARITHMETIC THROUGHPUT PER SECONDS. SAME EXPERIMENTAL SETUP OF TABLE II

From Tables II and III, it is also possible to deduce that the ABC computation, in average, punishes the performance in 20% for the homogeneous versions, but up to 40% to the GPU version. The reason behind this is the cost of moving not fully continuous pieces of data between the memory of host to the GPU, and the fact that even when the data is in the GPU memory the computation of those ABC areas impede a full parallel treatment thread-wise. The results expose the two main limiting factors for most algorithms mapping to GPU, and in particular for our case, the bandwidth utilization and the efficient exploitation of GPU parallelism.

B. I/O Benchmarking

As initially mentioned in Section II, the RTM implementations demand huge amount of I/O effort, this is mainly because wavefields are stored for every iteration of the forward sweep, then restored in every iteration of the backward sweep, this mechanism enables the correlation task that at the end produces the final image. In Figure 10 we can observed the performance of the implementations with the I/O mechanism active. As it is expected, the performance of the RTM versions is greatly affected (top figure), just to keep in mind, the elapsed time for the same experiment without I/O is for the homogeneous architecture versions around 60 seconds, and for the GPU is 12 seconds.

Fortunately, as mentioned in Section V-C3, and thanks to numerical properties of the mathematical scheme, a trade-off is exposed, where wavefield store frecuency is traded for quality of the resulting seismic image. This allows to store fewer number of wavefields with minimal loss of image



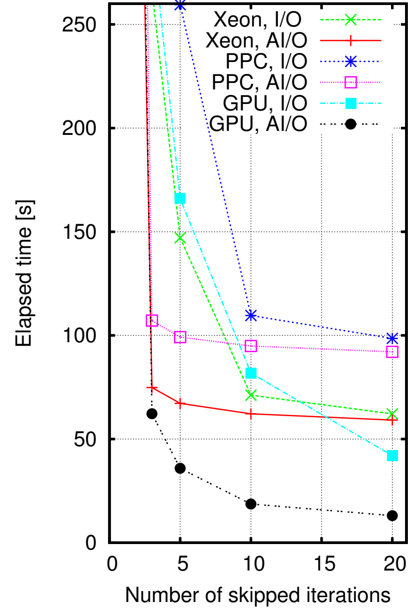


Figure 10. The experimental setup correspond to a data set size of 352^3, 300 forward iterations, and kernel + ABC computations. `Top` graph shows the complete set of experimental results. `Bottom` graph shows a detailed section of the former, where we can observe the behavior of the implementations in the 5–10 skipped iterations range, the most widely used.

quality, but this also means that the stores are separated by a fixed number of iterations. Regarding the geophysical model, the number of skipped iterations range between 5 and 10. As our experiments (Figure 10) show, the deployment of this trade-off shows a substantial performance improvement, in particular when is accompanied by the utilization of

AI/O. For instance, elapsed time gains when the skipped iteration is set to 5 are: 2.2x for Xeon, 2.6x for PowerPC and 4.6x for GPU. In the GPU case is particular important due to strong relation between bandwidth (host – device) and overall performance.

Results (Figure 11) show a good load balance among the threads, in particular for these traces of the PowerPC version of RTM. The top trace graph shows multiple iterations of the algorithm (forward and backward sweeps). The detailed trace graph (bottom Figure 11) shows the initialization of the AI/O, and the rest of I/O is completely overlapped with computation. As mentioned in Sections IV,V-C3, along the AI/O deployment, we introduce a compression algorithm, which for itself reduces up 10% the execution time.

C. Complete RTM Benchmarking

We have also run the complete RTM kernel in order to fully compare the three platforms to take into account the remaining computation and the I/O transfers needed to perform the correlation in the backwards phase. The GPU version needs an additional transfer from the GDDRAM to the host memory to be able to write the data to the disk. Thus, its performance loss is greater than that for homogeneous processors. However, as we can see in Table IV, it is still 7.5× faster than the PPC version and 6.7× faster than the Harpertown version. It is Important to remark that the global performance follows Amdahl's law, but not all RTM taks scale alike, this partially explains the performance loss from just the kernel experiments to the complete RTM executions.

Experiments	PowerPC Time [s]	Xeon Time [s]	GTX 280 Time [s]	GPU gain
F500B700S10	326.95	253.58	40.05	8.16, 6.33
F1400B2500S10	998.51	886.55	131.97	7.57, 6.72

Table IV
PERFORMANCE RESULTS FOR THE RTM IMPLEMENTATIONS RUNNING COMPLETE EXECUTIONS OF THE ALGORITHM. EXPERIMENT F500B700S10, STANDS FOR 500 FORWARD ITERATIONS, 700 BACKWARD ITERATIONS AND THE CORRELATION DISTANCE IS 10 ITERATIONS. EXPERIMENT F1400B2500S10, STANDS FOR 1400 FORWARD ITERATIONS, 2500 BACKWARD ITERATIONS AND THE SAME CORRELATION DISTANCE. THE DATA SET SIZE USED IN THE EXPERIMENTS IS 352^3

Finally, we provide a comparative table that relates computational performance with energy consumption. This is an important issue when a platform is assessed for HPC.

In Table V can be observed that the GPU implementation features an energy efficiency corresponding to 0.36 GFlops/W, which is 12× higher than the JS21 and 3× than the Altix.

Platform	Avg. Power [W]	Arithmetic Throughput [GFlops]	Energy Efficiency [GFlops/W]
JS21	257	8.01	0.03
Altix	120	12.53	0.10
GPU system	236 + 75	113.35	0.36

Table V
COMPARISON BETWEEN A JS21, ALTIX AND GPU SYSTEMS IN TERMS OF POWER EFFICIENCY. THE GPU SYSTEM (GPU + HOST) FEATURING A NVIDIA GTX 280 SHOWS SIGNIFICANTLY BETTER VALUE, UP TO 0.36 GFLOPS/W

VII. CONCLUSIONS

We have presented an optimized software design, based on the CUDA GPGPU programming model, for the RTM seismic imaging algorithm. Our implementation shows, running on a NVIDIA GTX280 card, up to a 14.1× speedup when compared against a reference traditional multi-core platforms based on a PowerPC 970MP and Xeon processors. Furthermore, our implementation features an energy efficiency corresponding to 0.36 GFlops/W, which is up to 12× higher than the reference.

Our implementation shows that running RTM on GPUs is a viable solution for everyday use in industrial-size deployments. Moreover, it proves that the GPU inherently parallel architecture has the potential to be the leading architecture in the scientific computing in terms of performance and energy-efficiency.

Future developments will focus on further optimizations, support for bigger problem sizes, and developing a system-scale parallelization to support multiple NVIDIA devices which exploits efficiently the intra/inter-blade interconnect.

ACKNOWLEDGMENT

The authors thank the Barcelona Supercomputing Center for their permission to publish the material reported in this article.

REFERENCES

[1] G. A. McMechan, "A review of seismic acoustic imaging by reverse-time migration," *International Journal of Imaging Systems and Technology*, vol. 1, no. 1, pp. 0899–9457, 1989.

[2] P. Micikevicius, "3d finite difference computation on gpus using cuda," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. New York, NY, USA: ACM, 2009, pp. 79–84.

[3] R. Abdelkhalek, H. Calandra, O. Coulaud, G. Latu, and J.Roman, "Fast seismic modeling and reverse time migration on a gpu cluster," in *Proceedings of The 2009 High Performance Computing & Simulation - HPCS'09*, June 2009.

[4] A.-C. Lesage, M. Araya-Polo, and G. Houzeaux, "Wave acoustic propagation for geophysics imaging, finite difference vs finite element methods comparison and boundary condition treatment," in *8th. World Congress on Computational Mechanics (WCCM8) 5th European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS 2008)*, Venice, Italy, Jun 2008.

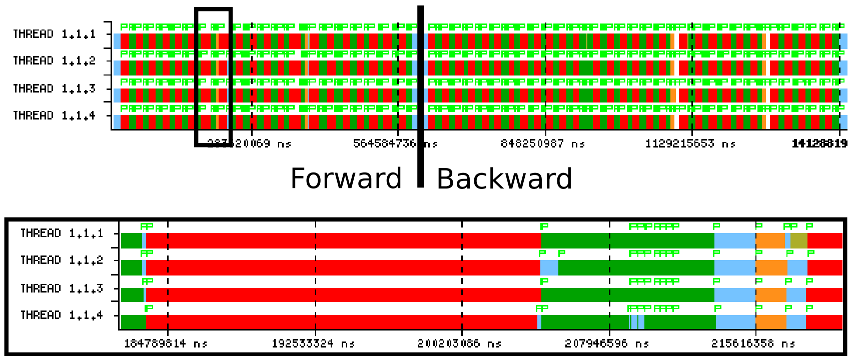


Figure 11. Execution trace of a complete RTM execution (PowerPC case). For sake of clarity, this execution only performs 20 and 30 steps in forward and backward sweeps respectively. `Top`: it is possible to distinguish: kernel computation (red), ABC computation (green) and waves correlation (white). `Bottom`: Detailed iteration, the kind that have to store a wavefiled. Same color scheme as Top, but also in this trace orange represents compression and light brown depicts AI/O. The execution traces were obtained with `Paraver` ([15])

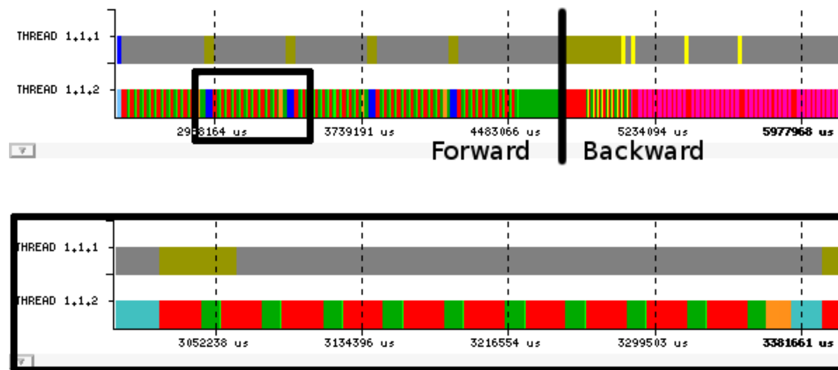


Figure 12. `Top`: Complete GPU execution trace. The first thread executes the disk I/O operations. The second thread invokes the GPU kernels. `Bottom`: Zoom of the forward phase. Same color scheme as previous figure, besides the light blue which represents the GDDRAM to host transfers

[5] A. Ray, G. Kondayya, and S. V. G. Menon, "Developing a finite difference time domain parallel code for nuclear electromagnetic field simulation," *IEEE Transaction on Antennas and Propagation*, vol. 54, pp. 1192–1199, April 2006.

[6] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with cuda," *Micro, IEEE*, vol. 28, no. 4, pp. 13–27, 2008.

[7] "Intel xeon processor 7400 series." [Online]. Available: http://www.intel.com/Assets/PDF/datasheet/320335.pdf

[8] S. Patel and W. mei W. Hwu, "Guest editors' introduction: Accelerator architectures," *IEEE Micro*, vol. 28, no. 4, pp. 4–12, July 2008.

[9] G. Rodriguez, R. M. Badia, and J. Labarta, "An evaluation of Marenostrum performance," *Int. J. High Perform. Comput. Appl.*, vol. 22, no. 1, pp. 81–96, 2008.

[10] R. de la Cruz, M. Araya-Polo, and J. M. Cela, "Introducing the *semi-stencil* algorithm," in *Proceedings of Eighth International Conference On Parallel Processing And Applied Mathematics (to be published)*. Springer-Verlag, September 2009.

[11] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick, "Impact of modern memory subsystems on cache optimizations for stencil computations," in *MSP '05: Proceedings of the 2005 workshop on Memory system performance*. New York, NY, USA: ACM Press, 2005, pp. 36–43.

[12] G. Rivera and C. W. Tseng, "Tiling optimizations for 3D scientific computations," 2000. [Online]. Available: citeseer.ist.psu.edu/rivera00tiling.html

[13] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," *SIGPLAN Not.*, vol. 26, no. 6, pp. 30–44, 1991.

[14] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, pp. 46–55, January 1998. [Online]. Available: http://www.cs.umd.edu/ hollings/cs714/papers/c1046bw.pdf

[15] J. L. Gabriele Jost, Haoquian Jin and J. Gimenez, "Interfacing computer aided parallelization and performance analysis," in *International Conference on Computational Science (ICCS'03)*, Melbourne, Australia, 2007.