

# BENCHMARKING 3D RTM ON HPC PLATFORMS

Francisco Ortigosa, Hongbo Zhou, Santiago Fernandez, Repsol-YPF,  
Mauricio Hanzich, Mauricio Araya-Polo, Félix Rubio, Raúl de la Cruz and  
José María Cela, Barcelona Supercomputing Center (BSC)

## SUMMARY

Reverse Time Migration (RTM) has become the dominant technology in seismic imaging for geologically complex subsurface areas. In particular has proven to be very useful for the subsalt oil plays of the US Gulf of Mexico. However, RTM cannot be applied on daily basis due to the extreme computational demand. The recent availability of multi-core processors, homogeneous and heterogeneous, may provide the required compute power. In this paper, we benchmark an effective RTM algorithm on several HPC platforms to assess viability of hardware.

## Keywords

3D RTM, High Performance Computing, Platform comparison

## INTRODUCTION

The increased exploration and development on complex deep-water targets in the Gulf of Mexico is giving rise to new challenges for subsalt seismic imaging. Technological advances in seismic imaging represent an opportunity to overcome this obstacle. Among these technologies, first-arrival Kirchhoff-like integral methods have difficulties in imaging complex geological structures where multi-pathing occurs. Downward-continuation algorithms, based on one-way non-elastic wave equation, can only propagate partial information and are incapable of accurately imaging overhang salt structures. However, RTM method overcome these problems by utilizing the complete two-way wave equation. As such, it does not suffer from the dip limitations inherent in one-way wave equation techniques, thus enabling imaging of overturned reflections and other complex structures. Unfortunately, RTM is highly compute intensive, more than the well-established one-way wave equation methods. Furthermore, RTM is compute intensive not only in terms of the increased data volumes of wide-azimuth marine acquisition, but also because of the high-fidelity algorithms.

In order for RTM to become a viable option for imaging complex geological structures, it is necessary to exploit parallelism to the best of its capabilities, making parallelism appear at all levels. The major CPU vendors have finished the clock speed race turning to adding parallelism support on-chip with multi-core processors. In this work, we will discuss on how RTM implementations can take advantage of these new multi-core processors, thus achieving massive computation capacity.

Because RTM will take advantage of multi-core processors, in particular if peta-scale capacity is required, as is the case for industrial size campaign. The following characteristics are desirable: low consumption, commodity and code porting. The multi-core processor must have very low power consumption to make peta-scale capacity economically viable and technically feasible. Moreover, widely available peta-scale capacity requires a commodity processor. Finally, but not less important, the porting or programming for this processor should be a feasible task. The processor would require widely available support to continue evolving, and given the previous requirement, support will be needed for proper programming.

Generally speaking, there are three-levels of concurrency: cluster, processor, and core levels. In this work, we will tackle the processor and core levels. At the processor level, multi-threading technique has been applied to fully utilize the multi-

core processor's shared-memory structures. At each core level, a vectorization process is introduced to take advantage of computer's SIMD (single instruction multiple data) register files, such as SSE2 (Intel). Obviously, understanding RTM mechanisms and numerical mathematics will add another dimension of performance speedup. A combined effort of all of these will give us a faster RTM implementation without sacrificing imaging quality.

The rest of the work is organized as follows: in the next section the main computational problems of RTM are introduced. Then, the homogeneous processors and implementation considerations regarding that kind of processors are described. After that, we introduce the heterogeneous processor and the RTM implementation on one of them. Then, the development effort is depicted, followed by the performance evaluation of the different platform. Finally the conclusions are presented.

### RTM IN A NUTSHELL

In order to solve the two-wave equation we need to deal with the laplacian calculation in a time loop, which is the most computing demanding segment of both ways of the RTM, this segment of code is called the kernel. Also, the RTM includes the source wave introduction (for the forward sweep (Alg. 1)), receivers traces introduction and wave field correlation (for the backward sweep), and boundary conditions.

```

1 forall time steps do
2     forall main grids points do
3         Wave field computation
4     end
5     forall shot area points do
6         Add the source wavelet
7     end
8     forall absorbing area points do
9         Apply absorption
10    end
11 end

```

Algorithm 1: RTM forward sweep.

The main time consuming step in Alg. 1 is the implementation of the wave field computation, that entails the calculation of a 3D stencil (step 3) for a Finite Difference method (Ray et al., 2006). This segment of code consumes 90% of the execution time.

### Figure 1. 3D Stencil (stencil-grad-new.eps)

Figure 1: 3D stencil inscribed in a 3D dataset, its memory access pattern (a), and the stencil size used in our RTM (b).

Such 3D stencil has a memory access pattern (Operto et al., 2006) that can be observed in Figure 1.a, represented by the cross-shaped object. Notice, that as an example of RTM system we uses a 4-point stencil for solving the PDE (Figure 1.b).

As can be seen from Figure 1 the direction Z is the only consecutive for the data in memory, then accesses to memory for the other directions is very expensive, in terms of L2 cache misses. This situation forces us to be careful about the way that data is processed to maximize the memory access vs. operations ratio. Besides, due to the reduced size of each L2/L3 cache (Table 2), special techniques must be applied to efficiently process the data.

## RTM IN HOMOGENEOUS PROCESSORS

In this section we will introduce our approach to implement RTM for homogeneous processors, where an homogeneous processor is defined as the processor composed by multiple identical cores. In general terms, our approach can be applied to any homogeneous processor, just the final tuning for the approach is hardware dependent.

As mentioned before, the memory access pattern is a main concern when designing the RTM kernel code (Kamil et al., 2005). Further, this pattern is totally dependent on the memory hierarchy structure of the target processor. Among the details we need to consider for each processor we have: the number of cache levels and the size and access cost for each one of them. Moreover, the way the data is placed and accessed for each level must be also taken into account.

However, and even considering the memory structure as a main concern, there are several ideas that can be applied to any of the processors shown in Table 1. One of the main approaches when trying to diminish the memory access cost is the idea of blocking (Rivera and Tseng, 2000; Wolf and Lam, 1991). The goal of this technique is to fill the cache level that is closer to the CPU in order to maximize the locality of the data being accessed, hence diminishing the necessity of accessing slower memory levels for getting the data.

	PPC970MP IBM	Woodcrest INTEL	Opteron AMD
Num. cores	2	2	2
Memory (GB)	4 (shared)	4	8
Frequency (GHz)	2.3	2.33	2.0
Peak (GFlops)	18.4	24	11
SIMD Registers	80	16	16
L2-D p/core (kB)	1024	4096 (shared)	512
L3-D (kB)	NA	NA	2048

**Table 1: Homogeneous processors technical specs.**

For applying the blocking technique, we transform the 3D data space needed for representing the wave field in each time step (Alg. 1), into a 6D space that is better suited for the homogeneous processors cache hierarchy. Figure 2 show how this decomposition is carried out. Notice that each sub-block of the 3D space is computed as shown in Figure 1.a and depicted in the previous section.

**Figure 2: Blocking structure for accessing data (blocking3.eps).**

Due to the multi-core nature of these homogeneous processors, it is very important to exploit the thread-level parallelism. For instance, our main testbed is the JS21 blade (as in MareNostrum supercomputer (Rodriguez et al., 2008)), each blade has 2 PPC970 processors that shares 8GB of memory in a Symmetric Multiprocessor (SMP) configuration (Table 1). We take advantage of the thread level parallelism giving to each thread (i.e. core) a sub-block of the 3D-space (Figure 2). As each PPC970 has its own cache, the processing of each sub-block can be done in parallel with almost no interfering. This is possible because the computation of each sub-block is independent of each other, and every PPC970 in a JS21 node can fill its own L2 cache almost in full parallel way. The parallelization at this level was implemented by means of OpenMP (Dagum and Menon, 1998).

Another important processor's characteristic to be considered when creating code for a specific architecture, is the availability of single instruction multiple data (SIMD) hardware. This processor characteristic let the developer apply the same operation to different data in parallel, increasing the number of operations that can be completed at each computational cycle by 2 or 4. Table 1 shows the SIMD registers availability for the depicted processors.

**Figure 3: Scalar and SIMD stencil representation.**  
(stencil-dac-vec.eps)

Our RTM implementation takes advantage of the SIMD registers. Figure 3.left shows the stencil we calculate in a scalar (i.e. not SIMD) code and the SIMD stencil (Figure 3.right) that uses SIMD registers and let us calculate four points of the data space in parallel. Notice, that both stencils take the same time to compute, but the SIMD version let us compute 4 points while the scalar code compute just one.

After review the main techniques applied to the RTM implementation for homogeneous processors, the next section introduces the processors that combine technologies or use external accelerators to reach high level of performance.

### RTM IN HETEROGENEOUS PROCESSORS

The current trend in the multi-core processors goes beyond just adding cores of existing technologies, other remarkable strategies to gain performance take advantage of a combination of processors of different technology. For instance, general purposes processors plus SIMD enable processors or accelerators, this is the case of Cell/B.E. processor which combine a PowerPC processor with an specific set of SIMD processors. Furthermore, Intel and AMD processors combine with FPGA (Field Programmable Gates Array) or GPU (Graphic Processor Unit) accelerators (Baker et al., 2007).

Among the mentioned architecture, we have choose to port our RTM code to the Cell/B.E. processor. This decision is based on practical considerations: this processor is available in HPC computing centers, has a high energy efficiency, and it is memory aware, which means that the fast and local memory levels are under the application command. This last consideration open new horizons in terms of algorithms to solve the problem at hand.

Before going into details of the porting to strategy, we give a brief introduction to this novel architecture. Afterwards, a series of implementation and optimizations are depicted.

#### What is Cell/B.E.

The Cell/B.E. processor is a multi-core chip composed of a general 64-bit PowerPC Architecture processor core (PPE) and 8 SPEs (SIMD processors) that have a small scratch-pad memory called local store (LS). A high speed bus (EIB) is shared among all components, allowing all of them to directly access main memory through the Memory Interface Controller (MIC). Figure 4 depicts the Cell/B.E. architecture.

**Figure 4: Cell/B.E. architecture. (cell\_architecture.tif)**

When programming the Cell/B.E. we have to focus in implementing algorithms that work with very small pieces of data, due to limited LS in each SPE (see Table 2). Moreover, every register inside the SPEs is a 16-byte SIMD register, which is capable to apply the same operation to a set of disjoint data in parallel. This implies that, in order to get the most of this processors, all the code executed by the SPE must be SIMD code.

	QS20	QS21
--	------	------

Num. processors	2	2
Num. cores	2 x 8	2 x 8
Memory (GB)	1	2
Frequency (GHz)	3.2	3.2
Peak (GFlops)	250	250
Vec. Registers	128	128
SPE Local store (kB)	256	256
PPE L2-D p/core (kB)	512	512

**Table 2: IBM QS20 and QS21 blades technical specs. The main difference is the increased RAM of QS21 blade**

### Efficiently accessing data

Due to the size of the 3D space to be processed it is necessary to split the data for parallel processing. Unfortunately, the data splitting expose the problem associated to the limited local-store (LS) on each SPE. Therefore, the first constraint to keep in mind is the LS size. Figure 5 shows how the data space is divided and scattered among the SPEs. Notice that the 3D space is splitted in X direction, then each subcube given to one SPE to be processed. Furthermore, Y is the traversing direction while processing each subcube.

#### Figure 5: Data accessing and vectorization pattern. (blk-vec.tif)

Given the fact that the processing subcube does not fit the SPE LS, it is mandatory to compute the data implementing a Z-Xi plane streaming. It should be notice, that this way of traversing data space probed to be the best option in the literature (Rivera and Tseng, 2000), to reduce the data traffic in the memory hierarchy.

### Enhancing Performance

Another main concern to get the most benefit from Cell/B.E. is to use SIMD code. This is important because each SIMD register can apply the same instruction to 4 data field points in parallel. The alternative is to use scalar operations that serialize the points processing with the corresponding decrease in performance.

Considering the number of SIMD register in Cell/B.E. (Table2) it is possible to optimize the code in order to compute up to 20 points of data field simultaneously, which lead us to high performance gain. In general purpose processors (i.e. most homogeneous), however, the number of SIMD register is smaller (Table 1), thus limiting to a small number of points that can be processed in parallel. For example, in PowerPC 970 processors we can only compute 12 points at a time, because it has 40% less SIMD register than Cell/B.E. (Table 2 and 1).

### DEVELOPMENT EFFORT

In this section we report productivity considerations regarding the development effort required to map RTM onto the JS21 and the QS2x architectures.

The work started by creating a portable, readable, unoptimized implementation whose emphasis was on the physics and mathematics of the underlying phenomena. It is worth mentioning that our first, naïve version, paid a high development effort price because of the steep learning curve of RTM concepts. Our development process is described below as a sequence of large refinement steps. In Step 1 of

Table 4, we produced the JS21 code starting from our naïve implementation and applying architecture-independent transformations to improve the execution time (selective loop merging), and reduce the memory footprint. We parallelized the code for the IBM JS21 platform by using OpenMP (Dagum, L. and R. Menon, 1998) directives. The result exploits thread-level parallelism, but not data-level parallelism yet. This step involved the most time-consuming task: performing blocking (**REF!**) by hand. The change required a rewrite of the data layout, caused a large increase in project size and required significant debugging effort. In Step 2, we manually SIMDized the computational kernel.

	Target	Project size (lines of code)	Effort (man-months)	Speedup
Step 1	JS21	1000	1.25	3.5x
Step 2	JS21	800	1.00	1.3x Step 1
Step 3	QS2x	1500	2.00	13.9x
Step 4	QS2x	500	1.25	18.9x

**Table 4: Development effort summary for the RTM project. For each step we report the amount of lines of code written, the effort required and the speedup caused by the step. Steps 3 and 4 taken from Table 2**

In Step 3, we ported the SIMDized JS21 implementation to the Cell/B.E. platform. We could reuse most SIMDization effort done for the JS21, but we had to replace the JS21-optimized blocked loops with explicit code to manage the LS. Most of the effort involved partitioning the workload in working sets small enough to fit the LS, and orchestrating computation and data-transfers. Incidentally, during this port we could apply a larger number of locality-improving techniques (Rivera, G. and C. W. Tseng, 2000) that were not applicable on JS21. Finally, Step 4 includes all the optimizations regarding double buffering, loop unrolling (i.e. better SIMD register utilization) and stalls reduction. Table 4 summarizes the above considerations. The larger size of the Cell/B.E. code with respect to the JS21 is due to the need for explicit thread- and LS-management code on the Cell/B.E. (1500 lines of code (LOC) and 2 man-months). The last 500 LOC lead us to a 18.9x speedup at a reasonable man-month cost. Admittedly, the development path we have adopted mixes steps that were beneficial to both targets. Nevertheless, some conclusions on programmer productivity can be drawn (J.S. Vetter et al., 2007). In summary, our experience suggests that:

- a significant corpus of memory-footprint-reducing and performance-improving optimizations are architecture independent; effort must be spent on them no matter what is the target platform;
- the effort to manually SIMDize a computational kernel is fundamentally the same on all the target platforms considered;
- despite the memory hierarchy management code (via cache blocking or LS management), which varies a lot between architectures, we did not observe a significant difference in man-month cost between the Cell/B.E. and the PowerPC.

## PERFORMANCE EVALUATION

To carry out the performance evaluation process, a testing-tuning-testing sequence was followed. As a first step we built several basic versions of the RTM code for both the homogeneous and the heterogeneous processors. The initial performance of such codes was far from being competitive, but it gives us a solid starting point in terms of algorithm robustness and numerical results correctness. After this, we introduced the optimizations mentioned in the previous sections, achieving

performance gains, that lead us to achieve a competitive performance on the Cell/B.E.

The velocity model we have used in the experiments has 192 x 384 x 560 points, (directions: Z x X x Y). We have repeated the migration benchmarks multiple times in order to eliminate the spurious effects on the execution time measurements of unpredictable phenomena like changes in bus traffic and operating system events. Figure 6 show the resulting images for both: the modeling (Fig. 6.left) and a RTM migration (Fig. 6.right) for a constant velocity field and one receiver.

Processor/Blade	Normalized time per shot
QS21 blade	1
QS20 blade	1.15
PPC970MP	8.80
Woodcrest	10.18
Opteron	17.36

**Table 3: RTM performance for different processors. Only one processor of the QS20 and QS21 was used for the test**

The performance result for the evaluated processors can be observed in Table 3, where the time required for migrating one shot is normalized to the best of them (i.e. QS21). It is important to remark that the time measures that are the origin of Table 3 cover all the computational segments of RTM, thus the measured time does neither include Input-Output nor data allocation and initialization. Table 3 shows a clear lead for Cell/B.E. This is mainly explained by a LS management that allows for high data reutilization. Furthermore, thanks to the double-buffering technique, we successfully manage to fully overlap the computation and data transfer, thus completely hide the computation latency. The experiments shows that our RTM reach as much as 27% of the peak performance of the machine.

**Figure 6: The left figure depicts the modeling result of the wave propagation, the right picture show the RTM final result (circle3.eps / impulse.tif)**

On the power efficiency front, from Table 4 we can deduced that being QS20/21 even 5 times more power consumption than Woodcrest (best homogeneous case), the migration takes more than 10 times faster (Table 3), thus still it is possible to save half of the power consumption needed by homogeneous processors.

Processor	Power consumption (W)	Arithmetic Throughput (GFlops/s)	Energy Efficiency (GFlops/W)
IBM QS20/QS21	315	58.3	0.16
IBM JS21 (PowerPC 970M)	267	7.3	0.03
Intel Woodcrest	160	5.73	0.04
AMD Opteron	304	3.36	0.01

**Table 4: Power consumption**

## CONCLUSIONS

The amount of computational resources needed for running a single industrial size migration are in the order of several Giga-bytes of memory and up to months of CPU time. Nevertheless, the benchmarking showed in the study allow us to conclude that

multi-core HPC platforms allow processing speeds applicable for realistic use of RTM as seismic imaging tool. However, not all current available platforms perform in a similar way. In particular the Cell/B.E. processor outperforms homogeneous processors.

**RTM is the fastest choice (almost an order of magnitude) and the most efficient, energetically speaking. Besides, the code porting is not too much expensive, but considering that some of the price was once paid for homogeneous platforms and reused for Cell processors.**

## **ACKNOWLEDGEMENTS**

This paper constitutes a contribution to the Kaleidoscope Project ([www.KaleidoscopeProject.info](http://www.KaleidoscopeProject.info)). The authors acknowledge Repsol and BSC for permission to submit this paper.

Also, the authors would like to thank Jizhu Lu and Daniele Scarpazza from IBM T.J. Watson research center, for their contribution on the optimization of the Cell/B.E. RTM kernel.

## **REFERENCES**

Baker, Z. K., M. B. Gokhale, and J. L. Tripp, 2007, Matched filter computation on fpga, cell and gpu: Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on, 207-218.

Dagum, L. and R. Menon, 1998, OpenMP: An industry- standard API for shared-memory programming: IEEE Computational Science & Engineering, 5, 46-55.

Kamil, S., P. Husbands, L. Oliker, J. Shalf, and K. Yelick, 2005, Impact of modern memory subsystems on cache optimizations for stencil computations: MSP '05: Proceedings of the 2005 workshop on Memory system performance, 36- 43, ACM Press.

Operto, S., J. Virieux, P. Amestoy, L. Giraud, and J. Y. L'Excellent, 2006, 3d frequency-domain finite-difference modeling of acoustic wave propagation using a massively parallel direct solver: a feasibility study: SEG Technical Program Expanded Abstracts, 2265-2269.

Ray, A., G. Kodayya, and S. V. G. Menon, 2006, Developing a finite difference time domain parallel code for nuclear electromagnetic field simulation: IEEE TRANSACTIONS ON ANTENNAS AND PROPAGATION, 54, 1192-1199.

Rivera, G. and C. W. Tseng, 2000, Tiling optimizations for 3d scientific computations.

Rodriguez, G., R. M. Badia, and J. Labarta, 2008, An evaluation of marenostrom performance: Int. J. High Perform. Comput. Appl., 22, 81-96.

Wolf, M. E. and M. S. Lam, 1991, A data locality optimizing algorithm: SIGPLAN Not., 26, 30-44.

J.S. Vetter, S.R. Alam and J.S. Meredith. Balancing productivity and performance on the cell broadband engine. In IEEE Annual International Conference on Cluster Computing (Cluster 2007), September 2007.