Scientific Programming 17 (2009) 185–198
DOI 10.3233/SPR-2009-0271
IOS Press

3D seismic imaging through reverse-time migration on homogeneous and heterogeneous multi-core processors

Mauricio Araya-Polo [a,*], Félix Rubio [a], Raúl de la Cruz [a], Mauricio Hanzich [a], José María Cela [a] and Daniele Paolo Scarpazza [b]

[a] *Barcelona Supercomputing Center, Barcelona, Spain*
[b] *IBM T.J. Watson Research Center, Yorktown Heights, NY, USA*

Abstract. Reverse-Time Migration (RTM) is a state-of-the-art technique in seismic acoustic imaging, because of the quality and integrity of the images it provides. Oil and gas companies trust RTM with crucial decisions on multi-million-dollar drilling investments. But RTM requires vastly more computational power than its predecessor techniques, and this has somewhat hindered its practical success. On the other hand, despite multi-core architectures promise to deliver unprecedented computational power, little attention has been devoted to mapping efficiently RTM to multi-cores.

In this paper, we present a mapping of the RTM computational kernel to the IBM Cell/B.E. processor that reaches close-to-optimal performance. The kernel proves to be memory-bound and it achieves a 98% utilization of the peak memory bandwidth.

Our Cell/B.E. implementation outperforms a traditional processor (PowerPC 970MP) in terms of performance (with an 15.0× speedup) and energy-efficiency (with a 10.0× increase in the GFlops/W delivered). Also, it is the fastest RTM implementation available to the best of our knowledge.

These results increase the practical usability of RTM. Also, the RTM-Cell/B.E. combination proves to be a strong competitor in the seismic arena.

Keywords: Seismic imaging, multi-core, reverse-time migration, performance

1. Introduction

Energy is the lifeblood of the world economy. The importance of energy sources in the contemporary geopolitical scenario cannot be overstated. Oil and gas availability determines global growth, energy security and regional stability.

In their search for new reserves, oil companies are turning to complex geological structures so far left unexplored because they are inherently hard to prospect and analyze. The most prominent examples are reserves located under salt domes, like in the offshore of the US Gulf of Mexico, which are estimated to hold 3000 sub-salt oil pools, which account for 37 billion barrels of “undiscovered conventionally recoverable” oil, and 191 trillion cubic feet of gas reserves [1]. Also, the discovery of a Brazilian deep-water sub-salt exploration area has recently been announced, which may contain as many as 33 billion oil barrels [2].

Oil discovery is a theoretical, algorithmic and computational challenge. Prospection is primarily performed with acoustic depth-imaging techniques: an intense acoustic signal is directed into the ground, and receivers record the signal’s echoes, e.g. as in Fig. 1. Then *wavefield reconstruction* methods are used to solve approximate equations that govern the propagation of acoustic waves through the Earth. These methods aim to determine densities and shapes of the subsurface structures. Oil companies trust wavefield reconstruction methods enough to rely on them for crucial multi-million-dollar decisions like whether and where to start drilling operations.

Two imaging methods dominate the arena: one-way Wave Equation Migration (WEM) and Reverse-Time Migration (RTM) [3]. WEM is very popular thanks to its lower computational cost and its acceptable degree of accuracy in traditional scenarios. RTM is more accurate, but its computational cost (at least one order of magnitude higher than WEM) hinders its adoption.

[*] Corresponding author. E-mail: mauricio.araya.bsc.es

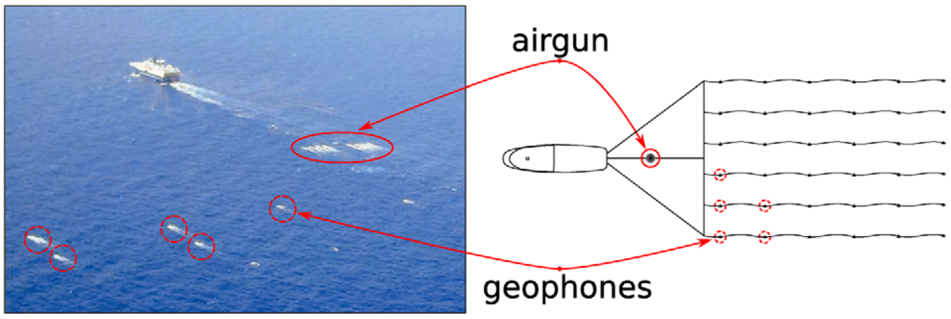


Fig. 1. A prospecting ship gathering data. The array of airguns is marked with a large circle in the photo. Airguns produce an acoustic wave whose echoes are recorded by the receivers (*geophones*). Receivers (small circles) are arranged in a grid located just behind the airgun array.

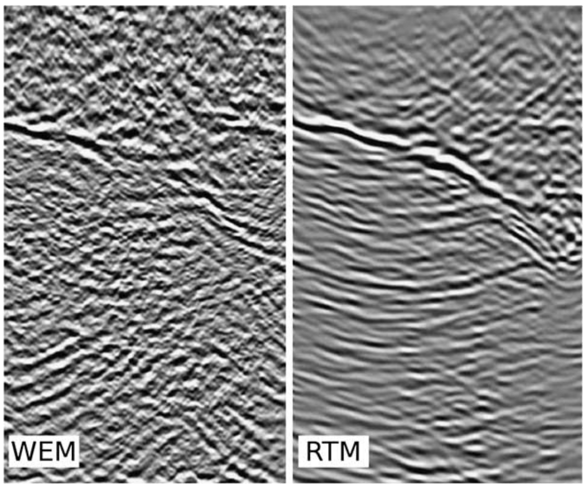


Fig. 2. A side-by-side comparison between the results of WEM and RTM. RTM provides higher quality images, with better signal-to-noise ratio and clearer structure delineation.

When it comes to new scenarios like reserves located beneath salt, large velocity contrasts or steeply dipping formations, RTM's imaging quality advantage over WEM becomes paramount. In fact, RTM can handle events that propagate along both directions of the depth axis, whereas WEM cannot. As a consequence, WEM performs poorly with structures, like salt flanks, that are illuminated by overturned reflections. Figure 2 presents a visual comparison of the results provided by WEM and RTM. In general terms, RTM allow savings when finding oil and gas.

On the computer architecture side, the last years' technological issues have put an end to frequency scaling. Architecture designers and application developers have turned to multi-core architectures in search for more performance. The IBM Cell/B.E. is a prominent example of the multi-cores employed in scientific computing, e.g. in the RoadRunner[1] project at the Los Alamos National Laboratory, with an expected peak performance of 1.4 PFlops/s. One Cell/B.E. processor provides remarkable floating point throughput (>200 GFlops/s) and main memory bandwidth (25 GB/s). Several works have shown how to reach almost-optimal utilization of these resources [4], depending on whether the workload is compute-bound or memory-bound [5], even in the case of irregular, nonnumerical workloads [6–9].

Moreover, the Cell/B.E. outperforms by a factor of 3–4 any other HPC platform in terms of energy efficiency [10], especially with high arithmetic-intensity codes. This makes the Cell/B.E. the only available choice to reach a given throughput for data centers located in certain power-limited urban areas. Its higher energy efficiency is mainly due to the use of software-managed scratchpad memories in place of per-core cache memories. Not only do caches occupy a relevant fraction of silicon area and absorb a major portion of the electrical power consumed in traditional multicore processors, but also the intense snooping traffic of cache-coherent systems requires a larger, power hungrier on-chip interconnect.

Although the Cell/B.E.'s computational power seems a good match to RTM's demand, no work so far has investigated an efficient mapping of the RTM to the Cell/B.E. This mapping is the purpose of our work.

We present an optimized implementation of the RTM's computational kernel that is specifically designed to exploit the architectural characteristics of the Cell/B.E.: 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 Single

[1] See: http://www.lanl.gov/roadrunner/.

Instruction Multiple Data (SIMD) arithmetics in the computational kernel to exploit the large amount of SIMD registers. Due to the relatively low arithmetic intensity of the computational kernel, the workload turns out to be memory-bound, and the cost of computation can be completely hidden. Our implementation achieves 98% utilization of the memory bandwidth. Results have been obtained on a dual-socket IBM QS20/21 blade. To the best of our knowledge, our RTM implementation has the best published 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 kernel for IBM JS21 blades, which sport a dual-core PowerPC 970MP processor (detailed technical specifications are reported in Table 1). Our results show that a Cell/B.E.-based QS21 blade outperforms a JS21 blade by a factor of 15.0 in terms of mere performance. Additionally, by delivering 0.30 GFlops/W, a QS21 blade also surpasses a JS21 in terms of energy efficiency, by one order of magnitude.

Thanks to these results, RTM is finally a much more accessible approach. Additionally our results show that the Cell/B.E. has the potential to become a leading platform for seismic imaging.

The remainder of this paper is organized as follows: Section 2 introduces the basics of the RTM algorithm and its main computational kernel. Sections 3 and 4 present respectively the results of developing/porting RTM to a traditional multi-core HPC platform (an IBM JS21 BladeCenter blade) and to a Cell/B.E. platform (an IBM QS20/QS21 BladeCenter blade). Section 5 evaluates and compares the performance achieved by the two solutions. In Section 6 we evaluate the coding effort spent during the development/port to the platforms considered. Finally, Section 7 concludes the paper.

2. RTM in a nutshell

In this section we briefly present RTM, and we identify its main computational kernel, which is the optimization target of this work.

The purpose of RTM is to generate the image of a geological *medium*, e.g., multi-mile-deep volumes of subsea geology. The RTM inputs consists of: an initial version of the medium to be studied, a wavelet, and the set of recorded acoustic wave pressure traces.

RTM simulates mathematically the propagation of sound in the given medium. In the simulation, first the medium is excited by introducing a wavelet (a *shot*), expressed as a function of frequency and time. Then, wave propagation (called forward propagation) is mathematically simulated by using an acoustic wave equation. Then, RTM repeats the task in a backward fashion: starting from the data recorded by the receivers, it propagates the wave field back in time (backward propagation). Finally, when both fields representing the forward and backward propagation are available, a cross-correlation between them is performed to generate the output image.

The acoustic wave propagation equation is a Partial Differential Equation (PDE). We assume an isotropic, non-elastic medium, where density is not variable. This equation is solved with a Finite Difference method (FD) [11]. The PDE solver involves a 3D stencil com-

Table 1
Technical specifications of all the systems employed in our experiments

Blade	JS21 type 8844	QS20 type 0200	QS21 type 0792
Processors	PowerPC 970MP	Cell/B.E.	Cell/B.E.
Sockets × cores	2×2	$2 \times (1\text{ PPE} + 8\text{ SPEs})$	$2 \times (1\text{ PPE} + 8\text{ SPEs})$
Memory per blade (GB)	8	1	2
Clock frequency (GHz)	2.3	3.2	3.2
Peak throughput (GFlops/s)	18.4	250	250
SIMD registers (per core)	80	128 (SPE)	128 (SPE)
SIMD width	128 bit	128 bit	128 bit
Main memory standard	DDR2	XDR	XDR
Cache memory			
L1 (data + instr.)	32 k + 64 k	32 k + 32 k	32 k + 32 k
L2	1 M per core	512 k per PPE	512 k per PPE
Scratchpad memory	–	256 k per SPE	256 k per SPE

Forward propagation part

Input: medium, shots
Output: forward wavefield

```
1:  for all time steps do
2:      for all main grid points do
3:          compute the wave field
4:      end for
5:      for all source location do
6:          add the source wavelet
7:      end for
8:      for all absorbing area points do
9:          apply absorption conditions
10:     end for
11: end for
```

Backward propagation part

Input: medium, receivers' traces, forward wavefield
Output: image

```
1:  for all time steps do
2:      for all main grid points do
3:          compute the wave field
4:      end for
5:      for all receivers location do
6:          add the receivers data
7:      end for
8:      for all absorbing area points do
9:          apply absorption conditions
10:     end for
11: end for
12: for all main grid points do
13:     correlate wave fields,
        forward and backward
14: end for
```

Fig. 3. The RTM algorithm.

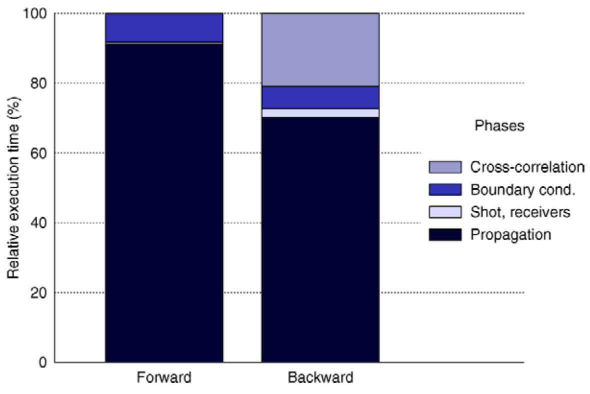


Fig. 4. Execution time breakdown of the RTM workload. The compute kernel ("Propagation", also line 3 in Fig. 3) dominates, with 91.2% of the forward part and 70.2% of the backward part. The cross-correlation between wave fields absorbs 20.9% of the backward part.

putation, followed by its corresponding integration in time.

Figure 3 gives a pseudo-code rendition of RTM, where the forward and the backward propagation parts are separated for sake of clarity. The PDE solver appears in line 3 of both the forward and the backward propagation part. The loop in lines 2–4 of both parts absorbs the vast majority of execution time (see Fig. 4, more details below). We elect this computational kernel as the object of study of this entire work.

The remainder of RTM as in Fig. 3 includes the following tasks. For the forward part: source wavelet introduction (line 6), boundary conditions (line 9). For the backward part: receivers' traces introduction (line 6), boundary conditions (line 9) and wavefield correlation (line 13). Lines 5–9 (both parts), and 12–14 (backward part) do not absorb significant execution time and are significantly simpler than the computational kernel.

We have benchmarked a sequential, unoptimized implementation of the code (Fig. 3) in order to determine the relative contribution to execution time of the different portions of the workload. The results are reported in Fig. 4. The benchmark shows that the workload is clearly dominated by the PDE solver [12] (line 3 in Fig. 3).

Figure 5 details the kernel. The loop structure shows its 3D nature. Line 5 computes the stencil for each point and lines 7–8 perform the time integration of the PDE solver. Thanks to our isotropic assumptions, the stencil can be computed as a Laplacian operator, i.e. in just one pass (rather than as a gradient followed by a divergence) as line 5 shows.

The computational weight of the kernel is due to the relatively high number of operations it performs per each data point. Its stencil uses the memory access pattern [13] depicted in Fig. 6(a). Our PDE solver uses a 8-point (per axis) stencil, depicted in Fig. 6(b). Data are stored in Z-major form (see Fig. 6(a)), therefore, accesses across the X and Y axes may be significantly more expensive in a cache-based architecture.

These concerns require us to pay special attention to how data are accessed: we will explicitly apply techniques to increase data reuse (reducing the overall amount of data transfers) and exploit the memory hierarchy as much as possible (reducing the overall transfer latency).

We devote the next two sections to discussing the details of mapping RTM to the two architectures that we have considered in this study: the PowerPC 970MP and the Cell/B.E. architecture, respectively.

```
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
```

Fig. 5. Pseudo-code of the unoptimized PDE solver invoked in line 3 of RTM (see Fig. 3). 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.

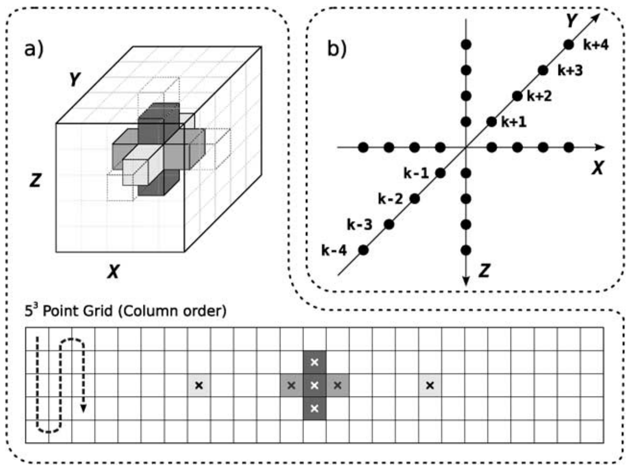


Fig. 6. The 3D stencil we employ: its memory access pattern (a) and the data points it uses (b).

3. RTM on the PowerPC 970MP

We want to be able to compare the Cell/B.E.-based RTM solution that is the focus of this paper against a reference solution based on a traditional cache-coherent multi-core platform. To set a fair comparison, both platforms must be based on commodity hardware available in an HPC-/supercomputing-oriented configuration. A platform that satisfies these requirements is the IBM BladeCenter JS21 Type 8844 blade, which sports two double-core PowerPC 970MP processors running at 2.3 GHz. These cores employ traditional coherent L1 and L2 cache memories. The AltiVec/VMX SIMD instruction set available in the 970MP processor is compatible with the one used in the Cell/B.E.'s Power Processing Element (PPE) core, and very similar in width (128 bit) and capabilities to the one of the Cell/B.E.'s Synergistic Processing Elements (SPEs). The blade is an off-the-shelf product and it is actively employed in supercomputers (e.g., MareNostrum [14]). Detailed hardware specifications are reported in Table 1. We have ported RTM to the JS21 platform and optimized its computational kernel to the same degree as its Cell/B.E. counterpart we discuss in Section 4.

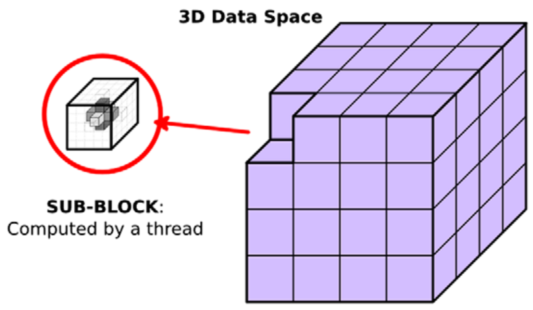


Fig. 7. The blocking strategy we adopt in our PowerPC 970MP implementation.

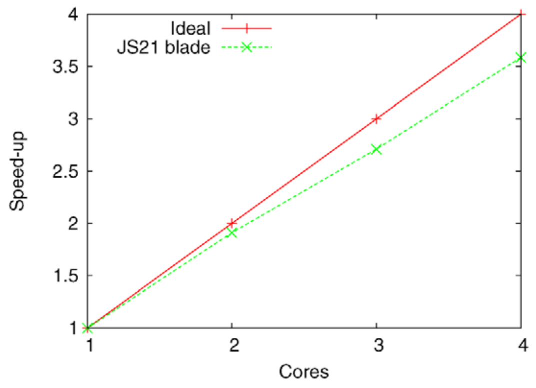


Fig. 8. Our RTM algorithm enjoys a good scalability on the JS21 platform.

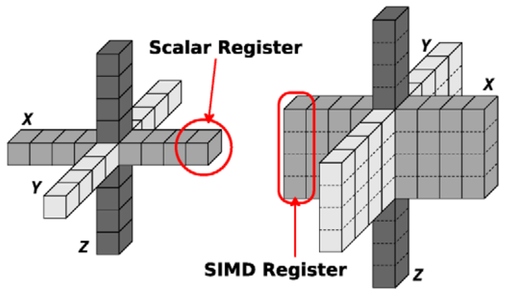


Fig. 9. Visual comparison between a scalar (left) and a SIMD stencil (right) in our RTM computational kernel.

For sake of clarity, we split the discussion on optimization into two topics: optimizing the memory accesses, and optimizing computation.

Memory access patterns have been shown [15] to be critical for the performance of the stencil computation, heart of the RTM kernel. Whereas this section focuses on JS21, we adopt general techniques, which are independent from low-level memory hierarchy details (e.g., number of levels, size, latency of each level, use of hardware caches or software-managed scratchpads). Without optimizations, the accesses in lines 5, 7 and 8 (Fig. 5) 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 counter that with *blocking* [16,17]. To apply blocking, we transform the 3D data space into a 6D space that better suits the cache hierarchy. Figure 7 shows this decomposition. Each sub-block of the 3D space is computed as in Fig. 6(a). Also, we employ OpenMP pragmas to enable the generation of explicit cache prefetching instructions, in an attempt to mitigate cache misses. We have considered including prefetching instructions manually in our source code, but our experiments showed a negligible performance improvement over compiler-generated code, at the expenses of an increased source code complexity and maintainability.

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 4 independent threads per blade (2 sockets $\times$ 2 cores) with a parallelization strategy that partitions the 6D data space into 3D sub-blocks (Fig. 7). Each core processes its assigned sub-set of sub-blocks independently. Since each core has its own L2 cache, interference among threads is minimal.

Our implementation employs OpenMP [18]. The blocked version of the algorithm sweeps through an index space of blocks that corresponds to the 3 outermost loops of the 6D space, whereas the 3 innermost ones are the ones in Fig. 5. We merged the 3 outer loops into a single one, to provide OpenMP with more opportunities for scheduling, thus enhancing scalability (Fig. 8).

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. The processor features relatively many SIMD registers (80), so that loop unrolling can be used in conjunction with SIMDization to extract more parallelism from the application. But SIMD instructions can be used only if the operand data are properly aligned. We specifically align our data to allow for a SIMDized compute stencil, as in Fig. 9(right). This organization delivers a significant speedup in the stencil computation with respect to a scalar implementation like the one in Fig. 9(left).

The techniques that we have just applied for our RTM implementation on the PowerPC 970MP platform provide us with an advanced starting point for our Cell/B.E. port, which is the main subject of the following section.

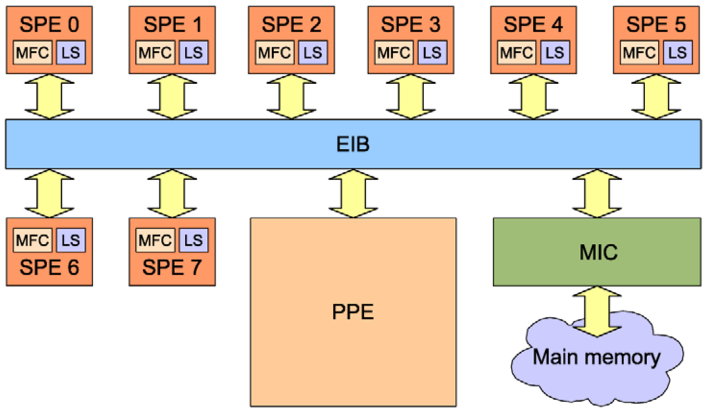


Fig. 10. Functional block diagram of the Cell/B.E. processor.

4. Porting the RTM onto the Cell/B.E.

In this section, we present briefly the Cell/B.E. architecture, and we illustrate the challenges and the details of our RTM optimized implementation for this target.

As a reference embodiment of the Cell/B.E. platform, we consider the IBM BladeCenter QS20 and QS21 blades (see Table 1 for the full specifications), which are dual-socket blades, mainly differing from each other by the amount of installed main memory (1 and 2 GB, respectively).[2] Each Cell/B.E. processor (see Fig. 10) on a QS2x blade contains a general-purpose 64 bit PowerPC-type PPE with cache memories, and 8 SPE with software-based scratch-pad memories called Local Stores (LS). The PPE and the SPEs have both a (slightly different) 128-bit wide SIMD instruction set, which allows for example to process simultaneously 4 single-precision floating-point operands.

Programming the SPEs in an efficient way is a challenging task because:

1. The use of SIMD instructions requires an appropriate data layout (padding and alignment);
2. The branch predictors are simple, and the mis-prediction penalty is high; control-flow-intensive should be rewritten as data-flow-intensive when possible;
3. Load/store instructions only operate on the LS, which is small (256 kB, shared for code and data); the programmer must divide computational kernels in fragments that operate on working sets small enough to fit the LS;
4. Accesses to the main memory only happen via Direct Memory Access (DMA) operations, and are performed independently by a Memory Function Controller (MFC); the programmer must use the MFC to overlap computation and transfers, to hide the shorter of the two latencies;
5. Also, DMA performance is influenced by usage parameters (transfer block size and alignment, average concurrent requests, bank congestion, controller congestion, NUMA issues); it is the programmer's responsibility to adopt congestion-avoiding memory access patterns.

In our implementation, we map our computational kernel (wavefield computation, line 3 in Fig. 3) onto to SPEs, while we run all the remaining tasks on the PPE. Our parallelization strategy partitions the input 3D space along the X axis as in Figs 11 and 12. Each SPE processes a distinct sub-cube. Within an SPE, Y is the traversing direction. Sub-cubes are much larger than the space available in the LS. Therefore, we implement a double-buffered Z–X plane streaming. This data-space traversal strategy proved [16] to be optimal in reducing the traffic in the memory hierarchy.

To exploit SIMD, we have adopted an aligned and padded data layout, and manually tuned the computational kernel by using the SIMD SPE intrinsics and language extensions available in the IBM Cell SDK 3.0. The many (128) SIMD registers allowed significant loop unrolling, i.e. each basic block computes 20 data points. This is a significant improvement over the Pow-

[2] At the time this study is written, QS22 blades are not available to us yet. The major improvements of QS22 blades are a larger memory endowment (32 GB) and the PowerXCell 8i processor, which features fully-pipelined double precision SPE floating point units. Since our RTM code only employs single-precision arithmetics, the impact of QS22 blades would not be substantial.

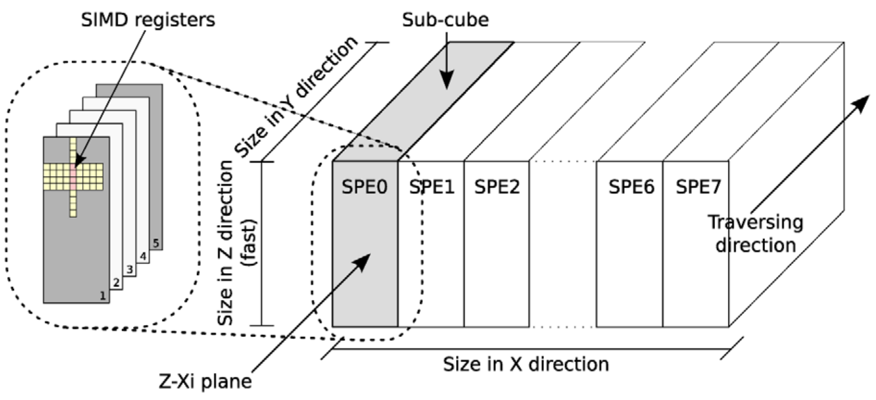


Fig. 11. Our parallelization strategy on the Cell/B.E. processor partitions the 3D input space along the X axis.

```
Input: dt2, C00, Z1, ..., Z4, X1, ..., X4, Y1, ..., Y4, u2, u1
Output: u3
 1: for all subcubes do
 2:   for $y = 1, \ldots, 4$ /* head */ do
 3:     for $x = 1, X\_subcube$ do
 4:       for $z = 1, Z/4$ do
 5:         Compute stencil and time integration /*4-way
            SIMD*/
 6:       end for
 7:     end for
 8:   end for
 9:   for $y = 5, \ldots, Y - 4$ /* body */ do
10:     for $x = 1, X\_subcube$ do
11:       for $z = 1, Z/4$ do
12:         Compute stencil and time integration /*4-way
            SIMD*/
13:       end for
14:     end for
15:   end for
16:   for $y = Y - 3, \ldots, Y$ /* tail */ do
17:     for $x = 1, X\_subcube$ do
18:       for $z = 1, Z/4$ do
19:         Compute stencil and time integration /*4-way
            SIMD*/
20:       end for
21:     end for
22:   end for
23: end for
```

Fig. 12. Pseudo-code of the SIMDized PDE solver as implemented on the Cell/B.E. The Z axis is 4-way SIMDized. The Y–X nested loops use in-place buffer recycling. This loop needs a separate head and tail to account for absorption boundary conditions. Every SPE computes one or more subcubes depending on the X dimension size and $X\_subcube$ size.

erPC 970MP-based implementation, where the fewer registers (80) allowed only for 12 data points to be processed for the same basic block.

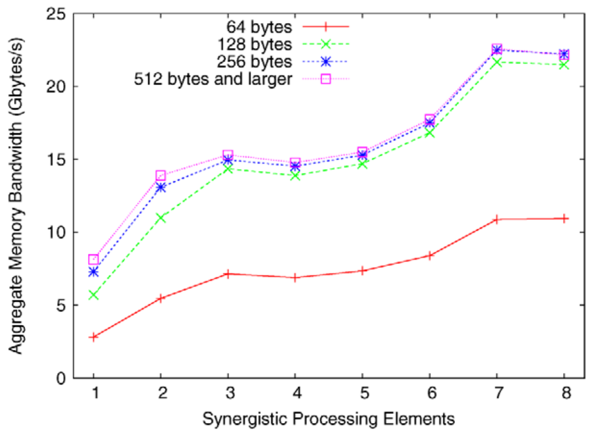


Fig. 13. Peak aggregate main memory read bandwidth per socket, as a function of the transferred block size. Transfers using a block size of 512 bytes of larger achieve the highest aggregate bandwidth, i.e. 22.1 GB/s (when 8 SPEs are used concurrently).

5. Performance evaluation

This section presents the experimental performance results obtained by our optimized Cell/B.E. implementation, and compares them against the theoretical performance bounds and JS21 results.

For sake of clarity, we discuss separately the optimization of data transfers and computation. Finally, we show how the two overlap in time and motivate that memory bandwidth is the bottleneck of the architecture for this application.

To determine the peak aggregate bandwidth we employ the same micro-benchmark suite as Kistler et al. [20]. Benchmarks show that the maximum per-socket bandwidth when all the SPEs transfer concurrently is 22.1 GB/s, and it is obtained when the block size is 512 bytes or larger (Fig. 13) and each SPE issues as many concurrent transfers as needed. This

Table 2

Impact of our Cell/B.E. optimizations on the performance of RTM

Implementation version	1	2	3	4	**5**
SIMD vectorization	No	Yes	Yes	Yes	**Yes**
Loop unroll factor	–	–	–	–	**3**
Clock cycles per iteration (M)	55	14	12	10	**6**
Average CPI	2.16	4.36	4.35	1.92	**1.03**
Single issue rate (%)	18.9	8.0	8.0	21.2	**51.7**
Dual issue rate (%)	11.0	6.6	6.6	12.2	**18.7**
Stall due to dependency (%)	13.4	10.8	10.7	9.8	**22.4**
Speedup	1.00	13.9	15.1	17.8	**18.9**
Aggregate bandwidth used (GB/s)	1.16	16.5	17.5	20.4	**21.6**

Notes: Our best implementation achieves a 18.9× speedup with respect to the unoptimized code. Results refer to a single Cell/B.E. processor.

represents the theoretical bound that our implementation should approximate. We have benchmarked the data-transfer part of our application, which transfers data in blocks as large as 15.6 kB, with the minimum number of concurrent transfers needed to feed the in-place multi-buffering recycling. We have measured a bandwidth value that is 98% of the peak value (i.e. 21.66 GB/s).

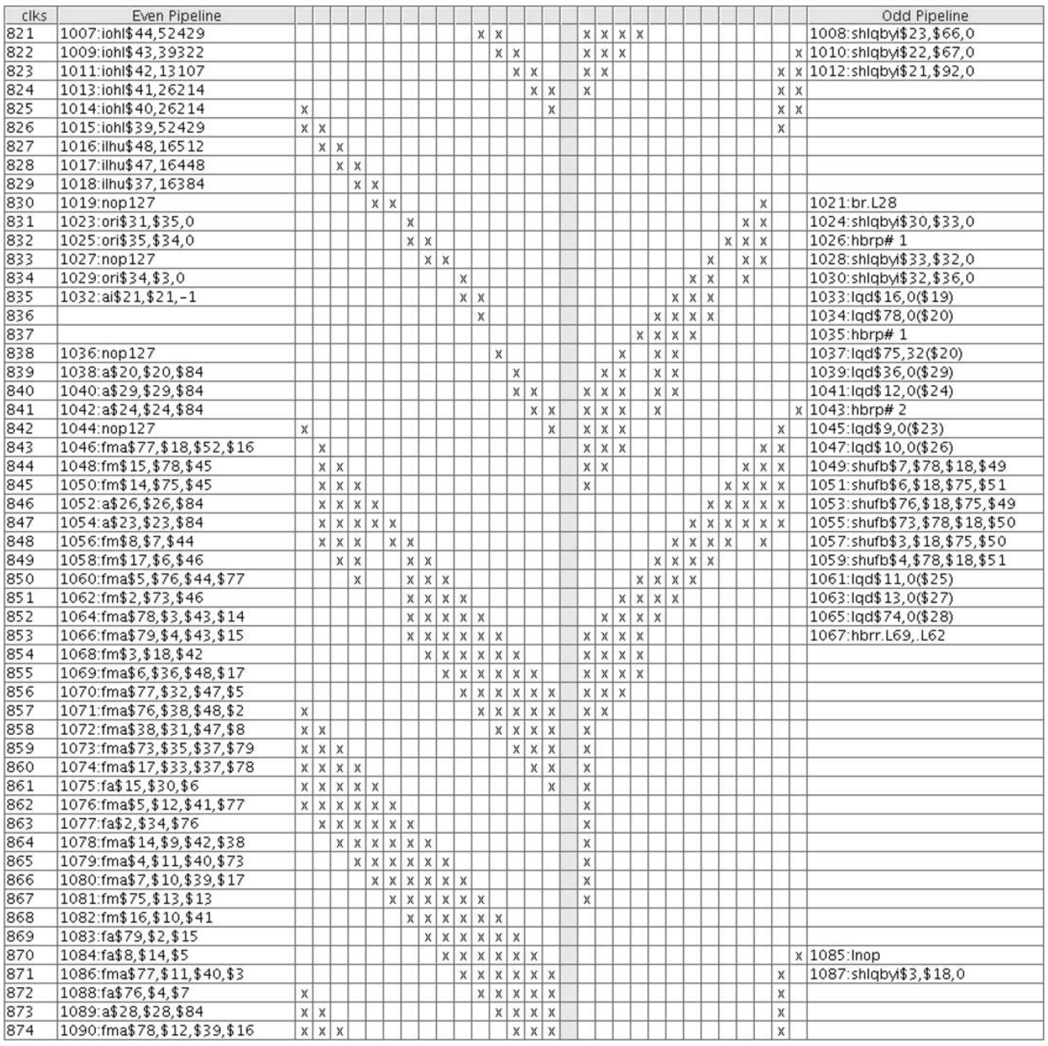
We now discuss the optimization of the computational kernel in isolation. We have optimized it in a sequence of refinement steps, which are summarized in Table 2. Version 1 is unoptimized. Version 2 uses SIMDization. Version 3 uses in-place buffer recycling to minimize the data transfers, at the expenses of a few additional address arithmetic instructions. Version 4 reduces the register pressure in the innermost loops, reducing the amount of spilling. Version 5 unrolls the loop along the X direction, reducing the dependency stalls and amortizing better the loop branches. Our best version (version 5) is a significant improvement over the unoptimized version, in terms of SIMD efficiency, dual issue rates and stalls: the instruction schedule, as statically analyzed by asmvis [19] (in Fig. 14) shows very dense code, with a high dual issue rate, limited stalls and `nops`, and a corresponding high CPI.

Finally, we discuss how computation and data-transfer interact to determine the overall performance of the algorithm. We analyzed the latencies involved in the computation kernel (for our best implementation) and in the associated data transfers. Results show that memory bandwidth is the bottleneck (last row of Table 2), and the cost of computation can be hidden completely under the cost of data transfers as depicted in Fig. 15. This condition holds for every possible input set and any trace segment. For instance, in Fig. 16, the body segment depicts has a computation time of 58 μs and a transfer time of 63.6 μs.

Our experiments use a test configuration where the input data set has 192 × 384 × 560 points (directions: $Z \times X \times Y$). In this configuration, computation takes (on the average) 91.1% of the transfer latency. The cost of scheduling and the impact of the loop heads and tails on the overall performance are negligible, as Fig. 16 shows. Scalability within a single Cell/B.E. chip is very good, as Fig. 17 shows. Our current efforts are targeting scalability of a single problem instance across multiple chips and blades. The good intra-chip scalability motivate little further optimization effort in the compute kernel. As the last line in Table 2 shows, our best implementation absorbs all the throughput that our data transfers are capable of providing in isolation.

We report comparative performance results in Table 3. The reported execution times do not include I/O time and non-recurring allocation or initialization delays, but include the entire algorithm presented in Table 3. Also, they are averages over repeated runs, to eliminate spurious effects (e.g. bus traffic, or unpredictable operating system events). Figure 18 shows an output images for both the model (Fig. 18(left)) and a RTM migration (Fig. 18(right)) when the algorithm is applied in a test configuration with a constant velocity field and one receiver.

The results show a clear advantage of QS2x blades over JS21 ones, both in terms of performance and energy efficiency. The Cell/B.E.'s higher energy efficiency is mainly explained by the absence of per-SPE cache memories. Additionally, the programmer can implement exactly his explicit working set policy on the SPEs, thanks to the MFC's programmability, whereas prefetching instructions offer a limited



clks	Even Pipeline																														Odd Pipeline
821	1007:iohl$44,52429											x	x					x	x	x	x										1008:shlqbyi$23,$66,0
822	1009:iohl$43,39322												x	x				x	x	x										x	1010:shlqbyi$22,$67,0
823	1011:iohl$42,13107													x	x			x	x										x	x	1012:shlqbyi$21,$92,0
824	1013:iohl$41,26214														x	x		x											x	x	
825	1014:iohl$40,26214	x														x													x	x	
826	1015:iohl$39,52429	x	x																										x		
827	1016:ilhu$48,16512		x	x																											
828	1017:ilhu$47,16448			x	x																										
829	1018:ilhu$37,16384				x	x																									
830	1019:nop127					x	x																					x			1021:br.L28
831	1023:ori$31,$35,0							x																			x	x			1024:shlqbyi$30,$33,0
832	1025:ori$35,$34,0							x	x																	x	x	x			1026:hbrp# 1
833	1027:nop127								x	x															x		x	x			1028:shlqbyi$33,$32,0
834	1029:ori$34,$3,0										x													x	x		x				1030:shlqbyi$32,$36,0
835	1032:ai$21,$21,-1										x	x											x	x	x						1033:lqd$16,0($19)
836												x										x	x	x	x						1034:lqd$78,0($20)
837																					x	x	x	x							1035:hbrp# 1
838	1036:nop127												x							x		x	x								1037:lqd$75,32($20)
839	1038:a$20,$20,$84													x					x	x		x	x								1039:lqd$36,0($29)
840	1040:a$29,$29,$84													x	x			x	x	x		x	x								1041:lqd$12,0($24)
841	1042:a$24,$24,$84														x	x		x	x	x		x								x	1043:hbrp# 2
842	1044:nop127	x														x		x	x	x									x		1045:lqd$9,0($23)
843	1046:fma$77,$18,$52,$16		x															x	x	x								x	x		1047:lqd$10,0($26)
844	1048:fm$15,$78,$45		x	x														x	x								x	x	x		1049:shufb$7,$78,$18,$49
845	1050:fm$14,$75,$45		x	x	x													x								x	x	x	x		1051:shufb$6,$18,$75,$51
846	1052:a$26,$26,$84		x	x	x	x																			x	x	x	x	x		1053:shufb$76,$18,$75,$49
847	1054:a$23,$23,$84		x	x	x	x	x																	x	x	x	x	x	x		1055:shufb$73,$78,$18,$50
848	1056:fm$8,$7,$44		x	x	x		x	x															x	x	x	x		x			1057:shufb$3,$18,$75,$50
849	1058:fm$17,$6,$46			x	x			x	x													x	x	x	x						1059:shufb$4,$78,$18,$51
850	1060:fma$5,$76,$44,$77				x			x	x	x											x	x	x	x							1061:lqd$11,0($25)
851	1062:fm$2,$73,$46							x	x	x	x									x	x	x	x								1063:lqd$13,0($27)
852	1064:fma$78,$3,$43,$14							x	x	x	x	x							x	x	x	x									1065:lqd$74,0($28)
853	1066:fma$79,$4,$43,$15							x	x	x	x	x	x					x	x	x	x										1067:hbrr.L69,.L62
854	1068:fm$3,$18,$42								x	x	x	x	x	x				x	x	x	x										
855	1069:fma$6,$36,$48,$17									x	x	x	x	x	x			x	x	x	x										
856	1070:fma$77,$32,$47,$5										x	x	x	x	x	x		x	x	x											
857	1071:fma$76,$38,$48,$2	x										x	x	x	x	x		x	x												
858	1072:fma$38,$31,$47,$8	x	x										x	x	x	x		x													
859	1073:fma$73,$35,$37,$79	x	x	x										x	x	x		x													
860	1074:fma$17,$33,$37,$78	x	x	x	x										x	x		x													
861	1075:fa$15,$30,$6	x	x	x	x	x										x		x													
862	1076:fma$5,$12,$41,$77	x	x	x	x	x	x											x													
863	1077:fa$2,$34,$76		x	x	x	x	x	x										x													
864	1078:fma$14,$9,$42,$38			x	x	x	x	x	x									x													
865	1079:fma$4,$11,$40,$73				x	x	x	x	x	x								x													
866	1080:fma$7,$10,$39,$17					x	x	x	x	x	x							x													
867	1081:fm$75,$13,$13						x	x	x	x	x	x						x													
868	1082:fm$16,$10,$41							x	x	x	x	x	x																		
869	1083:fa$79,$2,$15								x	x	x	x	x	x																	
870	1084:fa$8,$14,$5									x	x	x	x	x	x															x	1085:lnop
871	1086:fma$77,$11,$40,$3										x	x	x	x	x	x													x		1087:shlqbyi$3,$18,0
872	1088:fa$76,$4,$7	x										x	x	x	x	x													x		
873	1089:a$28,$28,$84	x	x										x	x	x	x													x		
874	1090:fma$78,$12,$39,$16	x	x	x										x	x	x													x		

Fig. 14. The static-time analysis of our computational kernel performed with `asmvis` [19] shows a compact instruction schedule. Both pipelines are occupied most clock cycles, the rate of dual issues (i.e., cycles in which both pipeline issue one instruction) is high, there are no stalls and `nop`/`lnop` instructions are limited. These indicators vouch for a good code quality.

amount of control over the PowerPC 970MP's cache hierarchy.

6. 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 the RTM concepts.

Our development process is described below as a sequence of broad refinement steps (not to be confused with the "implementation versions" introduced in the previous section, which only refer to the Cell/B.E. port).

In step 1 of Table 4, we profiled the code and applied 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 [18] directives. The result exploits thread-level parallelism,

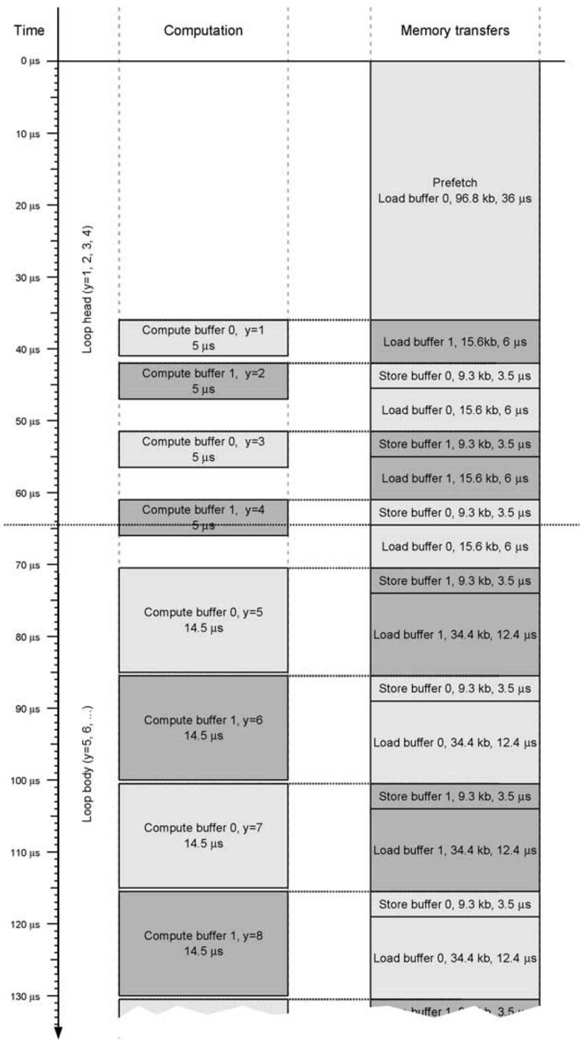

Time
Computation
Memory transfers
0 µs
10 µs
20 µs
30 µs
40 µs
50 µs
60 µs
70 µs
80 µs
90 µs
100 µs
110 µs
120 µs
130 µs
Loop head (y=1, 2, 3, 4)
Loop body (y=5, 6, ...)
Prefetch
Load buffer 0, 96.8 kb, 36 µs
Compute buffer 0, y=1 5 µs
Load buffer 1, 15.6kb, 6 µs
Compute buffer 1, y=2 5 µs
Store buffer 0, 9.3 kb, 3.5 µs
Load buffer 0, 15.6 kb, 6 µs
Compute buffer 0, y=3 5 µs
Store buffer 1, 9.3 kb, 3.5 µs
Load buffer 1, 15.6 kb, 6 µs
Compute buffer 1, y=4 5 µs
Store buffer 0, 9.3 kb, 3.5 µs
Load buffer 0, 15.6 kb, 6 µs
Store buffer 1, 9.3 kb, 3.5 µs
Compute buffer 0, y=5 14.5 µs
Load buffer 1, 34.4 kb, 12.4 µs
Store buffer 0, 9.3 kb, 3.5 µs
Compute buffer 1, y=6 14.5 µs
Load buffer 0, 34.4 kb, 12.4 µs
Store buffer 1, 9.3 kb, 3.5 µs
Compute buffer 0, y=7 14.5 µs
Load buffer 1, 34.4 kb, 12.4 µs
Store buffer 0, 9.3 kb, 3.5 µs
Compute buffer 1, y=8 14.5 µs
Load buffer 0, 34.4 kb, 12.4 µs

Fig. 15. By using multi-buffering and by scheduling accurately computation and data transfers, we can completely hide the latency of computation.

but not data-level parallelism yet. This step involved the most time-consuming task: performing blocking 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.

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 [16] that were not applicable on JS21. Finally, step 4 includes all the optimizations presented for the implementation version 5 of Table 2.

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 18.9× 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 [22]. 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.

7. Conclusions

We have presented an optimized software design for the computational kernel of the RTM seismic imaging approach, based on the Cell/B.E. architecture. Our implementation is close to optimality according to performance indicators (e.g., 98% of the peak bandwidth throughput), and it shows a 15.0× speedup when compared against a reference traditional multi-core platform based on a PowerPC 970MP processor. Furthermore, our implementation features an energy efficiency corresponding to 0.30 GFlops/W, which is 10.0× higher than the reference.

Our implementation is the fastest RTM implementation of which performance data are available. This solution contributes to enabling RTM – which has been regarded so far as desirable but too computationally demanding: gigantic datasets and months of CPU

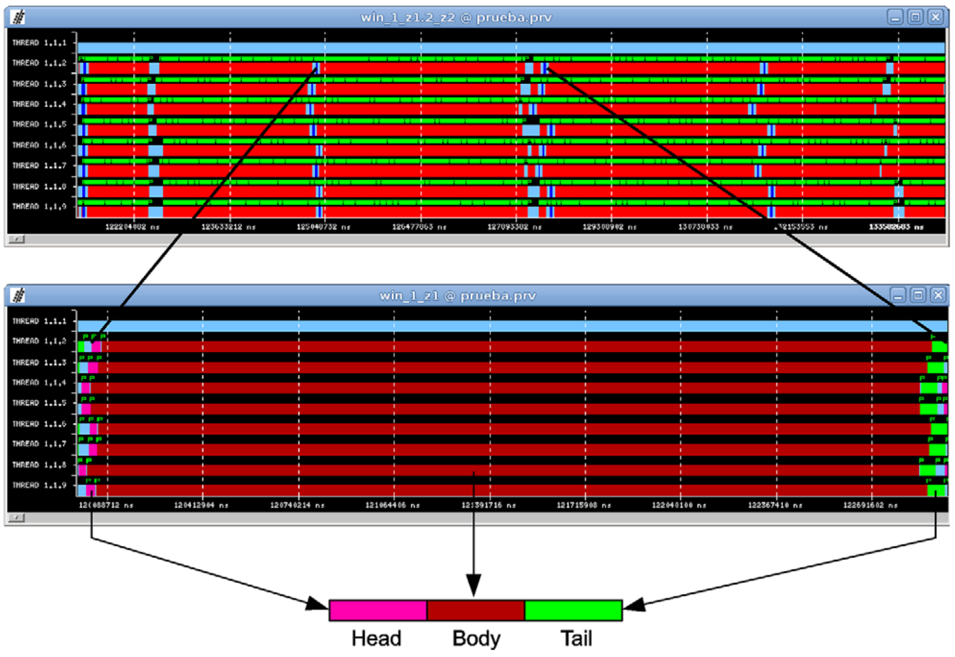


Fig. 16. Results show a good load balance among the SPEs. The top trace graph shows multiple time steps of the algorithm. The bottom trace zooms on a single time step. The head and tail (in light colors) represent a small portion of the overall execution time. The execution traces were obtained with `paraver` [21].

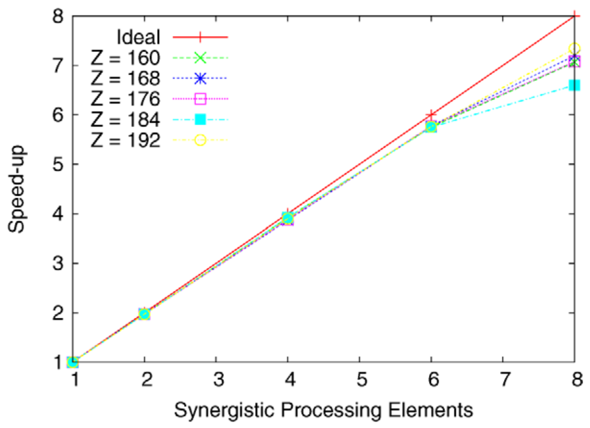


Fig. 17. Scalability of our Cell/B.E. implementation as a function of the Z dimension size, compared to ideal scalability.

time for today commodity processors – as a practically viable solution in everyday use for industrial-size deployments. Moreover, it proves that the Cell/B.E. has the potential to be the leading architecture in the seismic domain in terms of performance and energy-efficiency.

Future developments of this work include: tuning the algorithm to peak performance, extending our approach to the entire, production-level RTM workload,

Table 3

Comparison between a JS21 and QS2x blades in terms of power efficiency

Platform	Average power (W)	Execution time (s)	Arithmetic throughput (GFlops)	Energy efficiency (GFlops/W)
JS21	267	41.0	7.3	0.03
QS20	315	3.0	102.8	0.33
QS21	370	2.7	110.8	0.30

Notes: The QS2x blades featuring Cell/B.E. processors show significantly better values, up to 0.30 GFlops/W. QS2x results refer to two instances of the kernel, one per each Cell/B.E. Power consumption values were obtained in internal IBM measurements.

and developing an optimized system-scale parallelization which exploits efficiently the inter-blade interconnect.

Acknowledgments

The work described in this paper is part of the Kaleidoscope Project,[3] which is a cooperation be-

[3]See: www.kaleidoscopeproject.info.

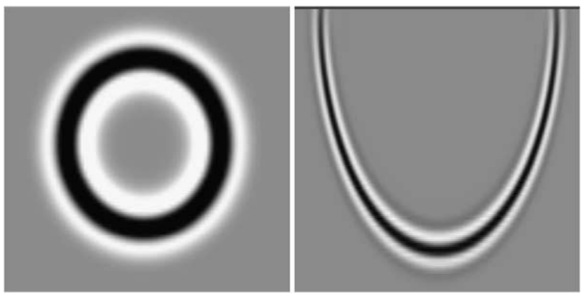


Fig. 18. On the left: The forward wave propagation for a given time step. On the right: the final result of RTM, for one shot and one receiver in a constant-velocity medium. The important features in both images are the perfect definition of the front-wave and the absence of artifacts like ripples (secondary waves).

Table 4

Development effort summary for the RTM project

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

Notes: 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.

tween Repsol–YPF, Barcelona Supercomputing Center, 3DGeo and CSIC.

The authors thank Repsol and the Barcelona Supercomputing Center for their permission to publish the material reported in this article. The authors thank Jizhu Lu and Michael Perrone from the IBM T.J. Watson Research Center for their valuable contributions to the optimization of the Cell/B.E. RTM kernel.

References

[1] G.L. Lore, D.A. Marin, E.C. Batchelder, W.C. Courtwright, R.P. Desselles and R.J. Klazynski, 2000 assessment of conventionally recoverable hydrocarbon resources of the Gulf of Mexico and Atlantic outer continental shelf, October 2001.

[2] Associated Press, Brazil official cites giant oil-field discovery, April 2008.

[3] G.A. McMechan, A review of seismic acoustic imaging by reverse-time migration, *International Journal of Imaging Systems and Technology* **1**(1) (1989), 18–21.

[4] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands and K. Yelick, The potential of the cell processor for scientific computing, in: *Proc. ACM Intl. Conf. on Computing Frontiers*, Ischia, Italy, May 2006.

[5] S. Williams, J. Carter, L. Oliker, J. Shalf and K. Yelick, Lattice Boltzmann simulation optimization on leading multicore platforms, in: *Proc. IEEE Intl. Parallel & Distributed Processing Symposium (IPDPS'08)*, Miami, FL, USA, April 2008.

[6] D.P. Scarpazza, O. Villa and F. Petrini, Peak-performance DFA-based string matching on the cell processor, in: *Proc. Third IEEE/ACM Intl. Workshop on System Management Techniques, Processes and Services (SMTPS'07), within IEEE/ACM Intl. Parallel and Distributed Processing Symposium (IPDPS'07)*, Long Beach, CA, USA, March 2007.

[7] D.P. Scarpazza, O. Villa and F. Petrini, High speed string searching against large dictionaries on the Cell/B.E. processor, in: *Proc. IEEE Intl. Parallel & Distributed Processing Symposium (IPDPS'08)*, Miami, FL, USA, April 2008.

[8] D.P. Scarpazza, O. Villa and F. Petrini, Efficient breadth-first search algorithms for advanced multi-core processors, *IEEE Transactions on Parallel and Distributed Systems* **19**(10) (2008), 1381–1395.

[9] A. Sarje and S. Aluru, Parallel biological sequence alignments on the cell broadband engine, in: *Proc. IEEE Intl. Parallel & Distributed Processing Symposium (IPDPS'08)*, Miami, FL, USA, April 2008.

[10] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands and K. Yelick, Scientific computing kernels on the cell processor, *International Journal of Parallel Programming (IJPP)* **35**(3) (2007), 263–298.

[11] 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: *Proc. 5th European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS 2008), co-Located with the 8th World Congress on Computational Mechanics (WCCM8)*, Venice, Italy, June 2008.

[12] 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* **54** (2006), 1192–1199.

[13] S. Operto, J. Virieux, P. Amestoy, L. Giraud and J.-Y. L'Excellent, 3D frequency-domain finite-difference modeling of acoustic wave propagation using a massively parallel direct solver: a feasibility study, in: *SEG Technical Program Expanded Abstracts*, Vol. 25, Society of Exploration Geophysicists, Melville, NY, USA, 2006, pp. 2265–2269.

[14] G. Rodriguez, R.M. Badia and J. Labarta, An evaluation of Marenostrum performance, *International Journal of High Performance Computing Applications* **22**(1) (2008), 81–96.

[15] S. Kamil, P. Husbands, L. Oliker, J. Shalf and K. Yelick, Impact of modern memory subsystems on cache optimizations for stencil computations, in: *Proc. 2005 Workshop on Memory System Performance (MSP'05)*, New York, NY, USA, 2005, pp. 36–43.

[16] G. Rivera and C.-W. Tseng, Tiling optimizations for 3D scientific computations, in: *Proc. ACM/IEEE Supercomputing Conference (SC 2000)*, Dallas, TX, USA, November 2000, p. 32.

[17] M.E. Wolf and M.S. Lam, A data locality optimizing algorithm, in: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, Toronto, ON, Canada, June 1991.

[18] L. Dagum and R. Menon, OpenMP: An industry-standard API for shared-memory programming, *IEEE Computational Science & Engineering* **5**(1) (1998), 46–55.

[19] G. Russell, Asmvis: IBM Assembly Visualizer for the Cell Broadband Engine, version 1.1, March 2008; available at: http://www.alphaworks.ibm.com/tech/asmvis/.

[20] M. Kistler, M. Perrone and F. Petrini, Cell multiprocessor communication network: Built for speed, *IEEE Micro* **26**(3) (2006), 10–23.

[21] J. Labarta, G. Jost, H. Jin and J. Gimenez, Interfacing computer aided parallelization and performance analysis, in: *Proc. International Conference on Computational Science (ICCS'03)*, Melbourne, Australia, June 2003.

[22] J.S. Vetter, S.R. Alam and J.S. Meredith, Balancing productivity and performance on the Cell Broadband Engine, in: *Proc. IEEE Annual International Conference on Cluster Computing (Cluster 2007)*, Austin, TX, USA, September 2007.