Efficiently Solving Wave Equation Migration on Multicore Processors

Michael Perrone*
IBM Research, Yorktown Heights, NY
mpp@us.ibm.com

Jizhu Lu and Janet Shiu
IBM Research, Yorktown Heights, NY, United States

and

Mark Bransford
Mayo Clinic's Advanced Medical Imaging Group, Rochester, MN, United States

Summary

This research explores the benefits of using multicore processors to reduce the time to solution for wave equation migration workloads. We implemented the core computation of a wave equation migration workload on the dual-core Intel Woodcrest Processor™ and the IBM Cell/B.E. Processor™. This work involved both the paralellization of the code over the cores and the SIMDization of the code on each core. As expected, this compute-bound kernel benefits from the additional compute power that these multicore processors provide; however we demonstrate that even in this compute-bound situation, one needs to efficiently manage the bandwidth requirements of the problem in order to achieve high performance. These optimizations led to performance increases of up to ~16x versus single core.

Introduction

The imaging techniques used by the seismic industry have been known since the mid 1970's but only became economically viable for large scale models as parallel computers became available in the mid 1990's. However, even with today's commodity processor clusters, the need for computational power continues to grow. With typical seismic survey data sets from 1 to 10 TBs in size and with large data sets up to 20 or 30TBs, production seismic imaging jobs can require anywhere from days to months of run time, even when using hundreds or thousands of processors. Additionally, more accurate and computationally-intensive algorithms exist and are being adopted as cost-effect computational power becomes available. The ability to run more advanced, more accurate algorithms and to produce results more quickly represent a substantial advantage to the seismic industry and may allow oil companies to improve find rates and reduce exploration time.

At the same time that this computational demand is growing, the processor industry is moving towards multicore. This transistion is putting additional demands on our algorithmic implementation.

Practitioners must pay more attention than ever to the data flows in and out of their compute nodes to assure that they achieve significant percentages of peak performance.

The goals of this research are to examine this problem in the context of a specific imaging workload to understand what coding strategies are needed; and to measure the performance improvement. For the purposes of this research it is impractical and unnecessary for us to implement a full, end-to-end imaging system. Instead, we chose to focus on the core computational bottleneck of one of the most common imaging algorithms, Wave Equation Migration. This bottleneck consists of the repeated convolution of a set of 2-dimensional Green's functions with successive depths of image data for two waves (upward and downward) and various frequencies.

Method

In order to explore the value of multicore processors for seismic imaging, we selected two comparable multicore processors on which to implement the a seismic compute kernel: the Intel dual-core Woodcrest processor and the Sony Cell/B.E. processor. Below we describe the kernel implemented, how the code was optimized and the experiments that were performed.

The Compute Kernel

Wave Equation Migration typically involves the successive calculation of upward and downward waves for each frequency of a set of frequencies and at each depth of a set of depths. The calculation is performed by convolving a 2-dimensional, radially symmetric, spatially varying Green's function over a 2-dimensional data space for each frequency and depth. For each point (x,y) in the image data at given depth and frequency, the Green's function calculation is

$$\Sigma_i G_i(x,y)[\Sigma_j D_j(x,y)]$$

where G_i is the Green's function value at radius R_i and is summed over the unique radii values in a given Green's function; and D_j is the set of data values that are a distance R_i from (x,y). The Green's function is in principle non-zero everywhere but decays to zero with increasing radius; so we truncate the Green's function to zero beyond a fixed radius. This radius can vary with depth; so we consider Green's functions of radius 8, 12 and 16 leading to Green's functions of approximateyl size 17x17, 25x25 and 33x33 in 2D. Also, since for a fixed radius, the computation from depth to depth is essentially the same, we consider only the performance at a single depth.

Optimization Strategy

Because of the significant differences in the hardware platforms that we explored, we employed different optimization strategies on each. For the Woodcrest, where the programming model is comparatively more straighforward, we simply used the usual vector instruction set to SIMDize the calculations and divided the computation evenly between the cores on the chip. On the Cell processor, where the direct memory access is required for all data access, the optimizations were more involved and are described below.

The data parallelization strategy for Cell is depicted in Figure 1. For each 2D set of data at fixed depth and frequency, we partitioned the data amongst the eight compute cores of the Cell (known as SPE's). Each SPE was responsible for calculating a different slice of the data so that in total, all the data was processed. In addition to these slices, redundant padding of width (depending on the size of the Green's function) of 8, 12 and 16 on each slice was used to account for the overlap inherent in the Green's function calculations. The red padded sections in Figure 1 correspond to the edge strip of blue neighboring data.

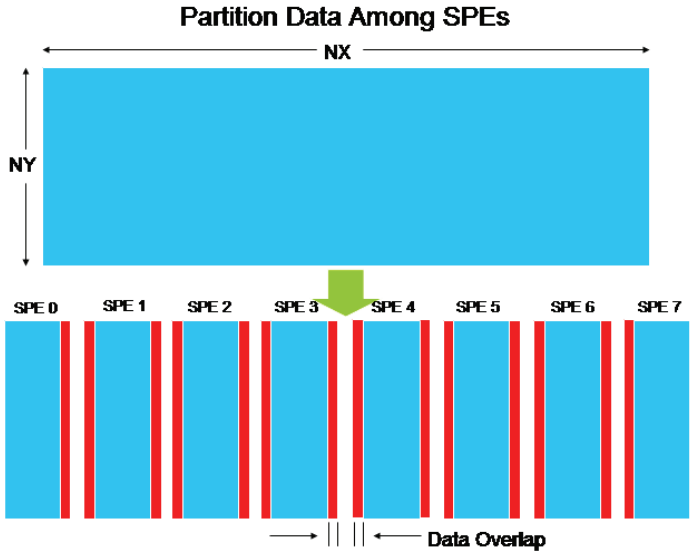

Figure 1: Partition of Data

The memory flow strategy on Cell is perhaps the most important aspect of the optimization. The detailed control that Cell provides enables the programmer to get significant performance improvements. In this case, we used the padding in Figure 1 to reduce communication requirements; but more importantly, we very efficiently managed the data flow required for each SPE. Figure 2 shows how the Green's function calculation progresses on an SPE. We initialize by loading all the image data (blue). Then for each (x,y) point we load the Green's function data (orange). While the calculation is being performed, the data for the next Green's function is loaded, effectively hiding the latency of the load. As the calculation progresses from left to right, the image data for the next row is loaded and again the computation hides the memory latency. When a full row of (x,y) points are calculated, the operation moves down a row and repeats from the left to right until all the image data is processed. For other processors that rely on a large L3 cache for hiding data latency this method won't work because of the large data size required to store all possible Green's functions and the fact that the access pattern of that memory is data drive and in principle random. To facilitate the latency hidding on Cell, double buffering was used for all of the data transfers. The end results of this attention to data flow is that the vast majority of the memory latency is hidden.

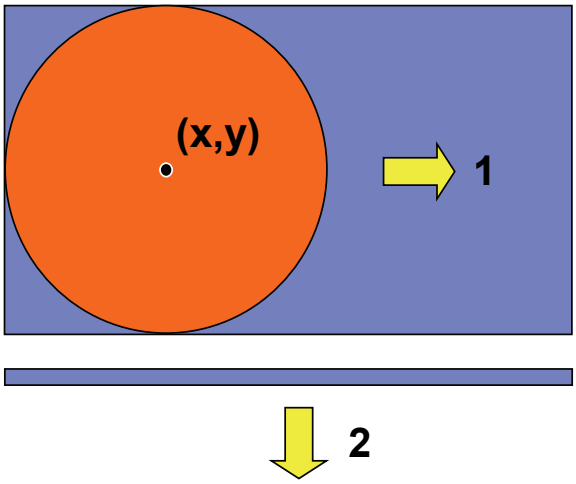

Figure 2: Green's function progression

The SIMDization strategy we used was to combine the image data from the upward and downward waves for two different frequencies. This gives us 4 data sets that are processed by the exact same procedure with different data. We apply the same proceudre to the corresponding Green's function data. This enables us to use the exact same scalar code to do the calculation with the modification that the scalar variables are now vector variables. There is an initial overhead for this data convertion which we can amortize over all of the depths in a calculation and so is negligible.

The over code flow on Cell in shown in Figure 3 in which the PPE box indicates the thread control process and the SPE box only shows one of 8 SPE threads.

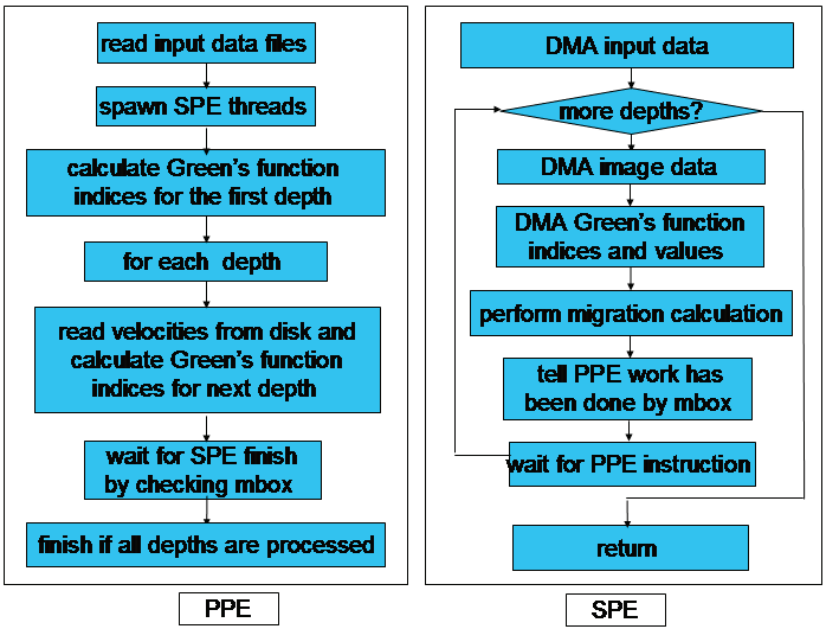

Figure 3: Cell code flow

Experimental Results

The experiments are performed on an Intel Woodcrest (3.0 GHz, 8GB RAM, 4MB Cache, 4GB Swap, 2 Cores) and a Sony Cell/B.E. (3.2 GHz, 1GB RAM, 512MB Cache, 2GB Swap). For a benchmark performance we compare to a single Woodcrest core. For the purposes this experiment, we used 5 iterations of the algorithm and a data space of several hundred elements in both the X and Y directions to help get stable performance numbers. We compared three different Green's function sizes: 8, 12 and 16 radii. The results are in Table 1 below comparing the performance between Woodcrest and Cell. As a baseline, we use single-core Woodcrest performing a single run which yields 1.95 seconds, i.e., a 15.7x Cell advantange.

Green's Function Size	Dual-core Woodcrest (two_jobs)	Cell/B.E. (one_job)	Speed-up:
8 radii	2043 mS	124 mS	8.3x
12 radii	1463 mS	94.9 mS	7.7x
16 radii	608 mS	81.6 mS	3.7x

Table 1: Performance Results

Conclusions

The work demonstrates that if one is willing to go the extra mile to optimize their code for multicore, the performance benefit can be substantial, nearly 16x in the case of Cell versus single Woodcrest core. Further we have seen that the Cell processor can significantly outperform the Woodcrest processor. In conclusion, it seems clear that as the number of cores on multicore processors increases, the issues raised in the paper will become increasingly important for practitioners to address if they hope to achieve high performance from multicore.

Acknowledgements

The authors would like to thank Bob Arenburg for useful discussions.