A Novel GPGPU Approach to Kirchhoff Time Migration

William Brouwer[1] and Vincent Natoli, Stone Ridge Technology*
Matt Lamont, DownUnder GeoSolutions

SUMMARY

Kirchhoff Time Migration (KTM) is a traditional seismic imaging technique used by geophysicists, a numerical method related to the Greens function solution of the scalar wave equation. KTM has been used routinely in some form or another for at least forty years, and a variety of computational implementations exist within academic and industrial settings. The KTM algorithm is relatively simple, however the high dimensionality of the overall data processing problem makes computation fairly expensive and thus a good candidate for High Performance Computing (HPC). In recent years, the Graphical Processing Unit (GPU) has taken a significant place among HPC platforms used for computationally intensive tasks such as seismic imaging. This work outlines a novel strategy for implementing the KTM algorithm in CUDA, which demonstrates excellent scaling and timing properties as compared to a CPU implementation.

INTRODUCTION

The energy industry has long driven High Performance Computing (HPC) innovation owing to significant computational need. Among the most demanding tasks performed is seismic imaging, used for the purposes of both exploration and reservoir monitoring. Imaging requires both high arithmetic intensity and memory bandwidth and is thus a logical candidate for General Purpose GPU (GPGPU) computing. Data may be readily organized into blocks within a grid, and data referenced for operations via a thread and block level index. Input and/or output data may be staged in shared memory, allowing for cooperation between threads. Finally, the overall grid may be designed with the geometry of the problem in mind, promoting coalesced loads thus hiding latency of global memory operations behind computation. Several works exist in the literature with regards to implementation of imaging algorithms on the GPU architecture. To date, there exist GPU implementations for Reverse Time Migration (Micikevicius 2009) and the subject of this paper, Kirchhoff Time Migration (Panetta *et al.*, 2009, Shi *et al.*, 2009); this article details a novel approach to the latter. Performance is reported alongside a high level description of the code, boasting a greater than 20x speed increase over a 3.2 GHz Xeon (Nehalem) quad-core CPU implementation, for a dataset realistic in size.

THEORY

A wide variety of imaging techniques have been developed over the years, this work assumes a traditional acquisition geometry ie., seismic traces are recorded at receiver locations and correspond to a particular source-receiver pair $\xi' = (x_s, y_s, x_r, y_r)$, Figure 1. Trace data is a digitized record of reflection events below the earth or ocean surface, providing highly valuable information on very large scales.

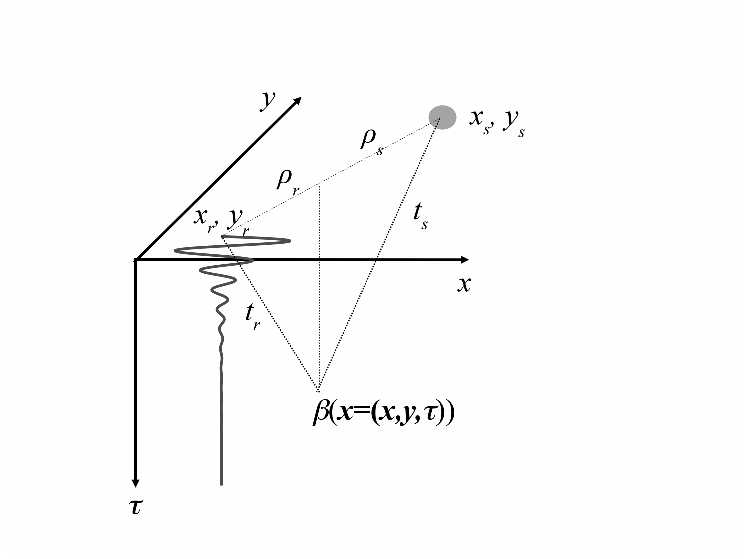

Figure 1: The x,y geometry of sources (solid circles), receiver trace signals, and image point $\beta(x)$ in a seismic survey.

More commonly one works in terms of offset-midpoint coordinates, defined as:

$$\xi = \left(\frac{x_r - x_s}{2}, \frac{y_r - y_s}{2}, \frac{x_s + x_r}{2}, \frac{y_s + y_r}{2}\right) = (h_x, h_y, m_x, m_y) \tag{1}$$

A single dataset for a particular survey might comprise $10^4 - 10^7$ traces, each consisting of $10^3 - 10^4$ digitized time samples, the voluminous data to be handled providing a challenge in of itself. Seismic imaging techniques differ in the manner in which this information is processed, for the purposes of this work attention is restricted to Kirchhoff Migration. As mentioned previously, this method of producing an output image $\beta(\mathbf{x})$ is related to the Greens function solution of the scalar wave equation (Bleistein *et al.*, 2001, Schneider 1978), which has an integral form:

$$\beta(\mathbf{x}) = \int_D W(\mathbf{x}, \xi) D(\xi, t(\mathbf{x}, \xi)) d\xi \tag{2}$$

where $\mathbf{x}$ refers to the image space, D is the acquired data, W are weights, and t is the total traveltime, which in the absence of lateral velocity variation has an analytic form:

[1] Currently at DownUnder GeoSolutions

$$t = t_s + t_r = \sqrt{(\rho_s/v)^2 + \tau^2} + \sqrt{(\rho_r/v)^2 + \tau^2} \qquad (3)$$

Figure 1 displays this geometrical relationship, one which depends on horizontal distance between source and image point projection on the surface ρ_s, receiver and image point projection on the surface ρ_r, as well as vertical traveltime or image pseudo-depth τ and velocity v. The latter is in general a complicated function, determined numerically from the data itself; where lateral velocity variations cannot be ignored, the traveltime must be calculated numerically as is the case with Kirchhoff Depth Migration (KDM). It is the traveltime which is used to select trace amplitude points from data D. Selected values are then used in turn to reconstruct the image $\beta(\mathbf{x})$, constructively and destructively interfering and weighted according to the value of W, which can also be a complicated function, particularly with regards to KDM. Referring to equation 2, a full migration constitutes a five dimensional summation; a variety of different approximations are often employed to reduce computational cost. This work assumes the full prestack dataset is to be migrated ie., there is no reduction in the input data fold size. Also, in practice, acquired data is subjected to a variety of processing steps before and during migration, including convolution with a derivative filter (Bleistein *et al.*, 2001, Biondi 2006). This work simply focuses on implementing the migration step alone in KTM, which typically constitutes more than 90% of computation time.

GPU IMPLEMENTATION

Overview

High throughput data processing algorithms are generally one of two kinds, either ‘scatter’ or ‘gather’. In a Symmetric Instruction Multiple Data (SIMD) or Symmetric Instruction Multiple Thread (SIMT) machine such as the Nvidia GPU, these methods map to the thread perspective, either the input or output grid respectively. In a multi-threaded environment the scatter method proves problematic if writing to a single memory location, in that race conditions are likely to occur. The gather method on the other hand writes to distinct memory locations, however, strided input array access is necessary. Roughly speaking, the high level computational model chosen should be dictated by relative sizes of input and output data volumes. In the model proposed here for KTM, a single thread is responsible for a single input trace. Threads must therefore also loop across trace records, as well as image depth τ. For each depth, traveltime $t = f(\xi, \mathbf{x})$ as a function of block index (image x, y), thread index (trace x, y) is calculated by each thread. If within the data bounds, trace data is loaded, weighted and stored in a shared memory array. A grid block is dedicated to a single output image x, y bin, and a tree reduction over the input data takes place in shared memory. The thread with lowest index in the tree reduction subsequently writes the image point $\beta(x)$ to global memory, Algorithm 1. This approach to performing migration is thus data-domain oriented, in distinction to the majority of algorithms which are best described as image-domain oriented or gather methods.

Algorithm 1

```
Input:
1.  Weighted trace values in shared array imageBlock
Output: Single image point, written to global memory
(∗ j is the loop index for image time ∗)
2.  offset = thread_y + THREADS_Y*j
3.  if thread_x < 64
4.          imageBlock[thread_x][thread_y] +=
5.          imageBlock[thread_x+64][thread_y];
6.          __syncthreads();
7.  if thread_x < 32
8.          imageBlock[thread_x][thread_y] +=
9.          imageBlock[thread_x+32][thread_y];
10.         imageBlock[thread_x][thread_y] +=
11.         imageBlock[thread_x+16][thread_y];
12.         imageBlock[thread_x][thread_y] +=
13.         imageBlock[thread_x+8][thread_y];
14.         ...
15.         imageBlock[thread_x][thread_y] +=
16.         imageBlock[thread_x+1][thread_y];
17.         __syncthreads();
18.
(∗ write image point to global memory ∗)
19. if thread_x==0
20.         image[block_x * imageDepths + offset] +=
21.         imageBlock[0][thread_y];
22.
```

Memory Configuration

In order to promote high data throughput while limiting the total number of global memory accesses, the trace and coordinate data are removed from SEG-Y format files and stored as separate, flat binary files. Each half warp or 16 threads accesses sequential addresses within a single 128-byte segment of coordinate data, and thus these loads are fully coalesced. Velocity and image coordinate data are stored in constant memory. The use of constant memory does however place a limit on the complexity of the velocity model, since constant memory is limited to 64k bytes, which amounts to 16k floats. Trace data is organized with time as the fast dimension, and trace record the slow dimension; both dimensions are padded with zeros according to the grid block size in x and y, `THREAD_X` and `THREAD_Y` respectively.

CUDA grid

Current Nvidia hardware supports a maximum of 64k blocks, which restricts the total number of x, y image points per kernel invocation to 64k. As mentioned, within each block a single thread is responsible for a different input trace in space, as well as image depth. Throughout execution, should a calculated traveltime index fall within the acceptable range in the depth `for` loop, input trace data is loaded into thread registers, weighted and accumulated in a shared memory location. The algorithm completes the depth loop iteration by reducing the data within the shared memory block, recursively as mentioned. This final step corresponds to reducing the data horizontally (input trace x, y) and is highly efficient, although barrier synchronization is necessary (Harris 2007) until within the last warp. Figure 2 illustrates the pertinent features of the grid used within this application. Additionally, as alluded to in Algorithm 1, a second block dimension y may be introduced, allowing for the possibility of scaling the total image loops performed.

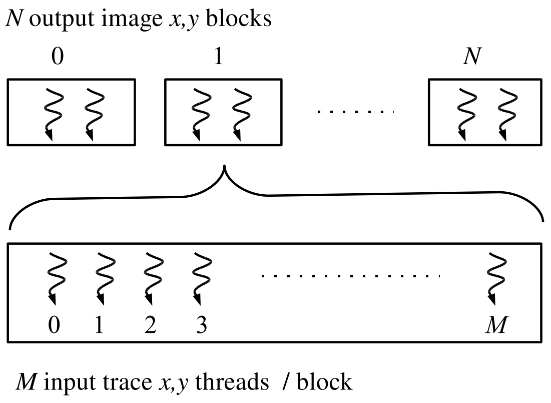

Figure 2: The CUDA grid; each block calculates one image x,y location by parallel reduction in shared memory, each thread corresponds to a signal input trace x,y location, looping over trace record blocks and image depths. A second thread block dimension may be introduced in order to scale the total image loops.

RESULTS

A CPU implementation of the KTM algorithm was written in the C language, paying careful attention to avoid superfluous calculation. As with the CUDA implementation, the innermost loop is dedicated to image depth, the outermost to trace record. The CPU code was compiled using the Intel compiler with the `-O3` flag, on a 3.2GHz (Nehalem) Xeon processor blade server running CentOS. OpenMP `parallel for` pragmas were also introduced, allowing for execution on all available cores. A CUDA version of the algorithm as outlined in this work was compiled using nvcc, with `-arch sm_12` to support polling functions, and `-ptxas-options=-v`, which confirmed the use of 24 registers, 564+16 bytes shared memory, 16424 bytes constant memory. The final code was executed on a single C1060 Tesla card, within the same blade server.

Initial experiments were conducted in order to validate output from the codes, the result of a test using the SEG/EAGE narrow-azimuth salt data set conducted on the GPU is displayed in Figure 3. Both CPU and GPU images compared well with another, confirming correct execution. After these steps and further optimizations, including the pre-computing of squared variables eg., velocity, the GPU code was profiled using the CUDA profiler. Total execution time and global memory bandwidth were optimized, by varying the grid dimensions, results are reported in Table 1. The best performance was achieved using a blocksize of (128,1), which represent a balance between scaling the outerloop iterations, and the range of global memory trace data locations accessed. However a gridsize of (64,2) compares favorably, a result which may improve as the size of the output grid relative to input increases, since the `THREAD_Y` dimension scales the image depth loop.

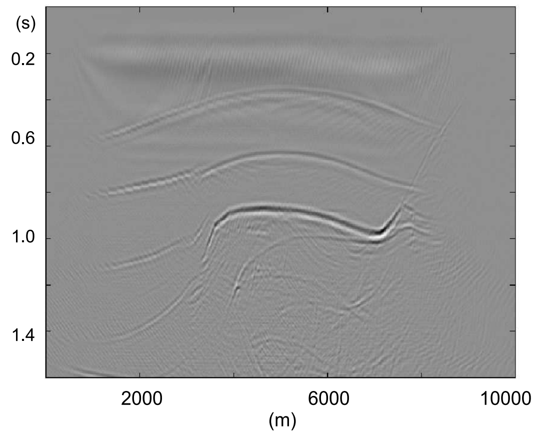

Figure 3: A migrated raw image of the SEG/EAGE C3-NA salt set at X=4630m, as determined via the proposed KTM GPU kernel.

Table 1: GPU grid optimization in CUDA profiler using a dataset of 640×6727 input trace points and $1 \times 255 \times 1248$ output points (bandwidth GBs/ time s)

threadIdx.y	threadIdx.x			
	64	128	256	512
1	44.8/1.2	57.5/0.9	49.3/1.0	45.0/1.1
2	56.8/0.9	49.5/1.0	43.8/1.1	-
4	48.3/1.1	42.5/1.2	-	-
8	37.2/1.4	-	-	-

In order to test the scaling of the GPU versus CPU implementation, larger test sets were created from the test dataset. In all cases, the output grid was $255 \times 255 \times 1024$ points. The CPU and GPU codes were then executed and timed for the aforementioned datasets, results are reported in Table 2. Referring to the table, the GPU times were a factor 19-21 times better than the optimized CPU code executed using four threads. Overall, the GPU implementation outperforms the CPU both in terms of scaling as well as absolute execution time.

Table 2: GPU/CPU timing results for input trace data of size $x \times 1024$ points and $255 \times 255 \times 1024$ output image grid

x	CPU Time (s)	GPU Time (s)
4096	2096	100
8192	4343	200
40960	24124	1243
83968	49663	2616

CONCLUSIONS AND FURTHER WORK

The GPU implementation of Kirchhoff Time Migration as detailed in this work significantly outperforms a comparable multi-threaded CPU implementation. The kernel execution also scales very well, taking advantage of coalesced reads from global memory, as well as low-latency on chip memory locations. The kernel is limited by both the maximum grid dimensions (64k blocks $\times$ 512 threads) and also the size of the constant memory cache. However, given the data decomposition possible with this type of problem, it is a relatively simple matter of dividing larger jobs into multiple kernel invocations, which is likely the case in multi- CPU/GPU distributed environments. This work has also neglected filtering and other preprocessing steps, as well as corrections due to geometrical and other factors. However the algorithm as described should provide a reliable template for incorporation into larger and more complicated codes. Current work is dedicated to accelerating the KDM workflow using GPGPU, particularly the fast marching solution to the eikonal equation.

EDITED REFERENCES

Note: This reference list is a copy-edited version of the reference list submitted by the author. Reference lists for the 2011 SEG Technical Program Expanded Abstracts have been copy edited so that references provided with the online metadata for each paper will achieve a high degree of linking to cited sources that appear on the Web.

REFERENCES

Biondi, B., 2006, 3D seismic imaging: SEG Investigations in Geophysics Series No. 14.

Bleistein, N., and S. H. Gray, 2001, From the Hagedoorn imaging technique to Kirchhoff migration and inversion: Geophysical Prospecting, **49**, no. 6, 629–643, doi:10.1046/j.1365-2478.2001.00290.x.

Harris, M., 2007, Optimizing parallel reduction in cuda, http://developer.download.nvidia.com/.

Micikevicius, P., 2009, 3D finite difference computation on GPUs using CUDA: Proceedings of 2nd General Workshop on General Purpose Processing on Graphics Processing Units, 79–84.

Panetta, J., T. Teixeira, P. R. P. de Souza Filho, C. A.da Cunha Filho, D. Sotelo, F. M. Roxo da Motta, S. S.Pinheiro, I. Pedrosa Junior, A. L. Romanelli Rosa, L. R.Monnerat, L. T. Carneiro, and C. H. B. de Albrecht, 2009, Accelerating Kirchhoff migration by CPU and GPU cooperation: Proceedings of the 21st International Symposium on Computer Architecture and High Performance Computing, 26–32.

W. A. Schneider, 1978, Integral formulation for migration in two and three dimensions: Geophysics, **43**, 49–76.

Shi, X., C. Li, X. Wang, and K. Li, 2009, A practical approach of curved ray prestack Kirchhoff time migration on GPGPU, *in* Y. Dou, R. Gruber, and J. Joller, eds., Advanced parallel processing technologies: Springer, 165–176.