Performance of RTM with subsurface offset gathers computiation fully offloaded to GPU

Abdullah AlTheyab, Robert Clapp

April 13, 2009

Abstract

Nvidia's graphics processing units (GPU) powered with CUDA (Compute Unified Device Architecture), the supporting API extension to the C programming language, allowed a significant speedup to reverse time migration (RTM) computation. FDTD GPU kernels, in particular, is an order of magnitude faster than similar CPU kernels [1]. Bus connection speed can be a performance bottleneck for mixed CPU-GPU processing. This can be avoided by fully offloading RTM to GPUs.

Imaging kernels that generate subsurface offset gathers can be run on the GPU to avoid transferring wavefields from the GPU. With a 4GB of memory, a single Tesla 10 series GPU can perform the 2D RTM with subsurface offset gathers generation. Computing subsurface

offset gathers are generally slower than wave propegation. In this report, we examine the performance of a general 2D RTM algorithm with subsurface offset gather computation fully offloaded to a single GPU, eliminating any data transfer from the GPU during migration. We also show an optimized imaging kernel that utilizes the shared memory to reduce the cost to $\approx 60\%$ of the naive imaging kernel.

1 Introduction

Reverse time migration (RTM) is one of subsurface imaging techniques used to construct an image that best represent the subsurface. Seismic data collected from the field is *migrated* by an imaging algorithm given an estimate of the wave propagation velocity in the subsurface. Estimates of the velocity field can be inaccurate at the first imaging attempt. Subsurface offset gathers can provide a measure of the errors in velocity estimation. Several iterations of imaging and error estimation is done in practice to reduce the errors.

RTM in particular uses the full wave equation unlike the other imaging algorithms which use approximations of the wave equation. This give RTM superior accuracy, but renders it expensive or sometimes unaffordable. Therefore, having an efficient RTM algorithms is vital for minimizing the turnaround time for creating a complete image.

Reverse time migration falls into the class of stencil computation. Those

stencil computation workload can be divided among many processing units in an embarrassingly parallel fashion. However, the main performance limitation on modern computer architectures is the memory latency. Cache aware algorithms minimize data traffic by taking advantage spatial and temporal locality and/or data prefetch capabilities of modern CPU's. Another way for hiding memory latency is to have more threads than cores and execute some threads while the other threads are waiting for memory access. The performance gain by this technique is not significant on CPU's because thread switching is expensive. This is not the case on GPUs, which can run a massive number of threads concurrently to hid memory latency. This makes GPU's very attractive hosts for this type of computational problems.

[1] have shown an order of magnitude increase in performance for GPUs FDTD kernel compared to multi-core CPUs. The reported performance numbers are optimized for 3D using equal grid spacing in all dimensions, without using an absorbing boundary condition, and without taking into account volume injection and/or boundary condtion.

Our goal is to develop an efficient RTM algorithm utilizing the advent of Tesla 10 series GPUs. In this report, we review briefly the theory of RTM for acoustic medium. We detail the GPU implementation of RTM with subsurface offset gather generation. We analyze the performance of general GPU kernels on a single device, avoiding the complexity introduced by communications between GPU and the host.

2 Governing equations

Wave propagation in an acoustic medium is described by

$$\left(\nabla^2 - \frac{1}{v^2(\mathbf{x})}\frac{\partial^2}{\partial t^2}\right) P(\mathbf{x}, t) = -f(\mathbf{x}, t). \tag{1}$$

where P is the pressure at a point $\mathbf{x}$ in the medium at time t. $v(\mathbf{x})$ is the wave propagation velocity field, and $f(\mathbf{x}, t)$ is the source term. A seismic experiment is conducted in the field by exciting a seismic source $f(\mathbf{x_s}, t)$ and recording at many receiver stations $(r_1, r_2, ...)$ where each receiver r_i is positioned at $\mathbf{x}_{r_i}$. That data are collected from one experiment into a *shot gather* $D_s(r_j, t)$. This experiment is repeated many times with different source locations to make a collection of shot gathers.

To build a subsurface image, each shot is migrated independently by simulating two wavefields using equation 1. The first wavefield is the *forward* propagated wavefield $P_f(\mathbf{x}, t)$ in which we try to mimic what has happened in the field. This is done by using the source term

$$f_f(\mathbf{x}, t) = f_s(t)\delta(\mathbf{x} - \mathbf{x}_s), \tag{2}$$

and a zero boundary condition above the Earth surface. $f_s(t)$ is the source signature that is usually given by data acquisition design. The second wavefield

is the *backward* propagated wavefield computed using the source term

$$f_b(\mathbf{x}, t) = \sum_j \delta(\mathbf{x} - \mathbf{x}_{r_j}) D_s(r_j, t_{max} - t). \tag{3}$$

The final image is computed using

$$I(\mathbf{x}, \mathbf{h}) = \sum_s \sum_t P_f(\mathbf{x} + \mathbf{h}, t_{max} - t; s) P^b(\mathbf{x} - \mathbf{h}, t; s), \tag{4}$$

where $\mathbf{h}$ is the subsurface offset. In practice, $\mathbf{h}$ is sampled in (1) the horizontal direction producing horizontal subsurface offset gathers $\mathbf{I}(\mathbf{x}, h_x)$, and (2) the vertical direction to produce the vertical subsurface offset gathers s $\mathbf{I}(\mathbf{x}, h_z)$. For a 2D imaging problem, the horizontal and vertical offset gathers are two 3D image volumes. Figures 1 and 2 show slices through the vertical and horizontal subsurface offset cubes respectively.

It must be mentioned that the procedure above give a kinematically correct image. Slight modifications to the equations provide accurate amplitudes [3]. However, those modification are not discussed here as they do not relate to the focus of this study i.e. optimizing the algorithm.

3 CUDA programming model

Compute Unified Device Architecture (CUDA) is a programming platform for massively parallel computation. Codes built with CUDA can run on graph-

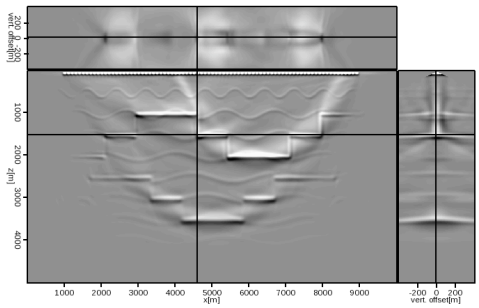

Figure 1: Vertical subsurface offset gather.

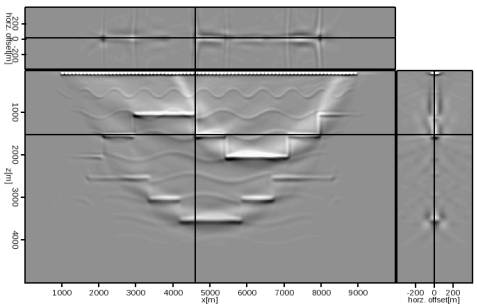

Figure 2: Horizontal subsurface offset gather.

ics processing units (GPU). A GPU has many multiprocessors, each with its own set of stream processors and shared memory. A GPU also have a global DRAM memory usually with a size of several gigabytes. This memory is uncached and memory latency is hidden by executing a massive number of threads concurrently. Threads within a block can share data using the shared memory. Threads are grouped together in thread blocks. Threads within a thread block can share data and synchronize using barrier. That's however is not possible between thread blocks. Calling a kernel from the host code will launch the thread blocks in unspecified order.

The number of processing units limit the maximum number of threads per thread block. Moreover, the fixed size of the fast shared memory and registers limit the number of thread blocks running concurrently on the GPU. For stencil computation, 2D thread blocks are mapped to the data grid meaning that each thread block handles a tile of grid points. The optimal thread blocks size is 16x16, which is determined by the device instruction set [2] and the available

shared memory (maximizing the number of grid blocks running concurrently on a multiprocessor).

Many considerations has to be taken into account for optimizing CUDA kernels. Coalesced global memory access can radically reduced memory access instructions. Shared memory usage can also significantly reduce global memory access. However, bank conflicts can hinder the performance gain of using shared memory. Therefore, kernels should be designed to avoid or reduce simultaneous access to the same memory banks by the threads in a thread block.

The kernels used for our implementation was run on Tesla 10-series GPU's (Tesla S1060) which contains 30 multiprocessors with 8 streaming processor each, and 16 KB of shared memory. The memory bandwidth is about 100 GB/s, and the global memory is 4GB.

4 Performance metric

The total cost for 2D RTM algorithm for a grid size N_xxN_z and N_t time steps is

$$C_{\text{RTM}} = (C_{\text{forward}-\text{FDTD}}+C_{\text{backward}-\text{FDTD}})N_xN_zN_tN_{shots}+C_{\text{Imaging}}N_IN_xN_zN_hN_{shots}, \tag{5}$$

where $C_{\text{forward}-\text{FDTD}}$, $C_{\text{backward}-\text{FDTD}}$, C_{Imaging} are the costs for forward, bacward wave propagation, and imaging (both horizontal and vertical), respectivly. N_I is the number of imaging steps, and N_h is the number of points in the subsurface offset axis. FDTD costs include injection/boundary condition. Because of asynchronous execution of backward propagating kernels and imaging kernels, it is difficult to isolate their costs. However, we estimate that $C_{\text{forward}-\text{FDTD}} = C_{\text{backward}-\text{FDTD}}$, where $C_{\text{forward}-\text{FDTD}}$ can be calculated by timing the forward propagation part of the algorithm. Therefore, the throughput metrics for wave propagations and imaging kernels are

$$\begin{aligned} C_{FD}^{-1} &= N_x N_z N_t/(\text{FDTD exec. time}), \\ C_{Imaging}^{-1} &= N_I N_x N_z N_h/(\text{RTM exec. time} - 2 * \text{FD exec. time}). \end{aligned}$$

both are expressed in Mpts/sec.

5 Wave propagation kernels

The wave equation is solved by explicite finite differencing that is second order in time and eighth order in space,

$$P^{\mathtt{t}+1} = v^2 \Delta t^2 \left(f^{\mathtt{t}} + \nabla^2 P^{\mathtt{t}} \right) + 2P^{\mathtt{t}} - P^{\mathtt{t}-1}. \tag{6}$$

We use similar approach to the one implemented by [1], but we generalize variable grid spacing for each dimension. We also have separate kernels for source function injection and recording. The energy of waves incident on grid boundaries are attenuated by adding an absorption term to the wave equation. That term is active around the neighborhood of the side and bottom boundaries. Those inclusions to the algorithm increase the number of floating point operations and memory accesses, which in turn reduce the throughput of FDTD few hundreds of Mpts/sec compared to the reported performance measures by [1].

The problem grid is divided into 16x16 blocks. Each grid block is assigned to a thread block of the same size; i.e. each thread performs the FDTD for a single point in the grid. Since data sharing is allowed between threads within a thread blocks, shared memory is used to keep local copies of $P^{\mathtt{t}}$ that is needed for computing the spacial derivatives. Each thread with a thread block loads the corresponding point on the grid of $P^{\mathtt{t}}$ into shared memory. Because the derivative stencil requires eight neighboring points in each dimension, some of the threads are assigned to load the halos, i.e. the surrounding points of the thread block.

6 Imaging kernel

The imaging equation 4 is used in practice to generate two image cubes with the two x- and z-spatial dimensions and a the third dimension along $h_x(h_z)$ for horizontal(vertical) subsurface offsets. The subsurface offset can have both positive and negative values. The following code snippet shows a simple kernel for imaging condition.

```
__global__ void img_kernel(int zmin, int xmin, int h, float *p1, float *
  int i=zmin+blockIdx.x*blockDim.x+threadIdx.x;     /* z image location *
  int j=xmin+blockIdx.y*blockDim.y+threadIdx.y;     /* x image location *
  img[iloc(i,j,h)]+=p1[loc(i,j-h)]*p2[loc(i,j+h)]; /* imaging condition*
}
```

This is a naive CUDA implementation of the imaging condition because no locality is taken advantage of because blocking is done based on image locations; i.e. x and z locations of the image points while looping along the offset axis. Blocking on the image point and horizontal offset slice and looping over the depth for the horizontal offset cube will perform better. That is due to the fact that there is not data dependencies in the dimension normal to the subsurface offset dimension. Secondly, blocking along the subsurface offset dimension enables data sharing between threads in a thread block. Figure 3 shows the data dependancy projection from the output to input dimensions. This data in the projection area (red and blue zones) can be shared between the threads in a

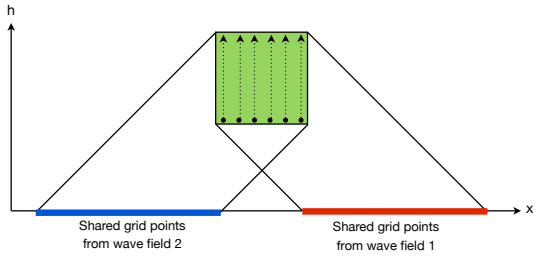

Figure 3: Thread block advancement along the offset axis and data sharing between threads within the thread block. All data access from with the thread block will fall within the shared areas from the forward and backward propagated wavefields.

thread block to make use of the fast shared memory and minimize data traffic from main memory.

For the naive imaging kernel, the number of device DRAM access for a thread block of size N_1x N_2 is $4N_1N_2$. The cost for the proposed algorithm is $2N_1N_2+2(N_1+N_2)$. For a 16x16 block, the number of global memory accesses is reduced from 1024 to 544.

7 Summery

In this report, we showed an efficient implementation of wave propagation kernel and a naive imaging kernel. We also showed a strategy for improving the imaging kernel that reduces the cost to about 60% of the naive implementation.

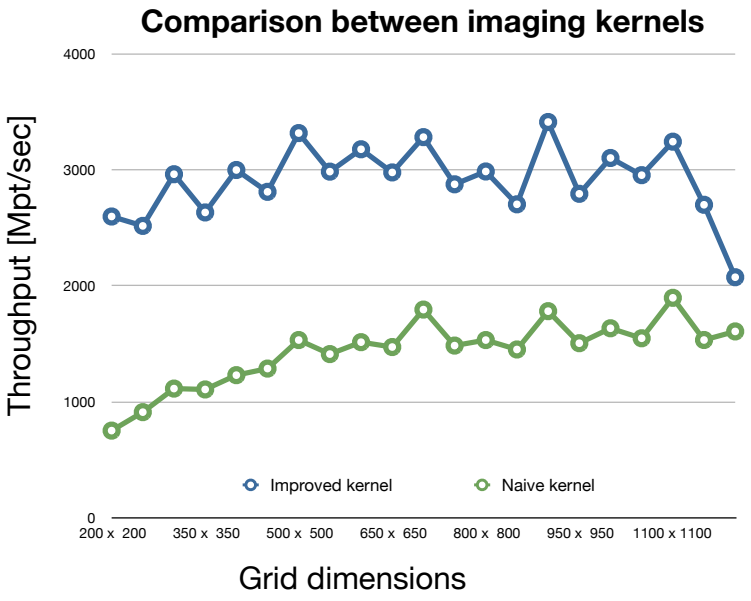

Figure 4: Performace of fully offloaded 2D RTM to a single GPU.

References

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

[2] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.

[3] Yu Zhang and James Sun. Practical issues in reverse time migration: true amplitude gathers, noise removal and harmonic source encoding. *first break*, 27:53–60, 1993.

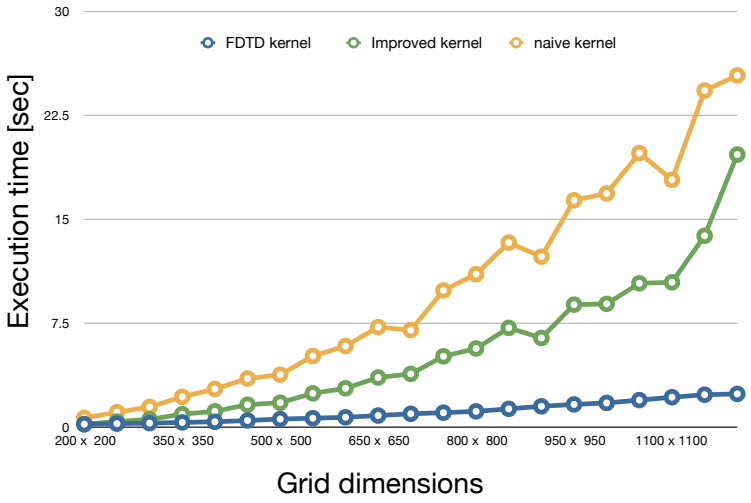

Figure 5: Execution time of RTM kernels for a single shot with 4000 time steps and 400 imaging steps and 81 points along the offset axis. Because of large output of imaging (81x grid size) per imaging steps compared to finite difference (20x grid size) most of the computing time is spent on generating subsurface offset gathers.