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

Abdullah Al Theyab, Robert Clapp

ABSTRACT

Nvidia's graphics processing units (GPU) powered with CUDA, the supporting API extension to the C programming language, has allowed a significant speedup to reverse time migration (RTM) computation. FDTD GPU kernel, in particular, is an order of magnitude faster than similar CPU kernels (Micikevicius, 2009). 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 kernels take the majority of the migration time because of the larger output volume. We examine the performance of a general 2D RTM algorithm with subsurface offset gather computation fully offloaded to a single GPU device. We also show an optimized imaging kernel that utilizes the shared memory to reduce the cost to about 50% of the naive imaging kernel.

INTRODUCTION

Reverse time migration (RTM) is a full wave equation imaging technique that constructs an image that best represent the subsurface seismic properties. Seismic data is migrated given an estimate of the wave propagation velocity in the subsurface. Estimates of the velocity field can be inaccurate at the first imaging attempt, and subsurface offset gathers can provide a measure of the errors in velocity estimation Biondi and Symes (2004). They also can give AVO information if amplitudes are handled properly. RTM and subsurface offset gathers generation are known to be computationally expensive and 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 CPUs. 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 CPUs because thread switching is expensive. This is not the case on GPUs, which can run a massive number of threads concurrently to hide memory latency. This makes GPUs very attractive hosts for this type of computational problems.

Micikevicius (2009) 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 condition.

In this report, we review briefly the theory of RTM for acoustic medium. We detail the GPU implementation of RTM with subsurface offset gathers generation. We analyze the performance of GPU kernels on a single device, avoiding the complexity introduced by communications between GPU and the host. We optimize the imaging kernel by using shared memory to reduce the cost of subsurface offset generation. The developed RTM kernels were run on Tesla 10 series GPUs.

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 (Zhang and Sun, 1993). However, those modification are not discussed here as they do not relate to the focus of this study i.e. optimizing the RTM kernels.

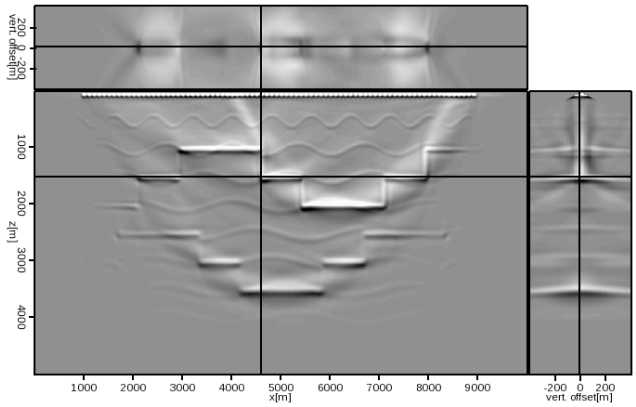

Figure 1: Vertical subsurface offset gather.

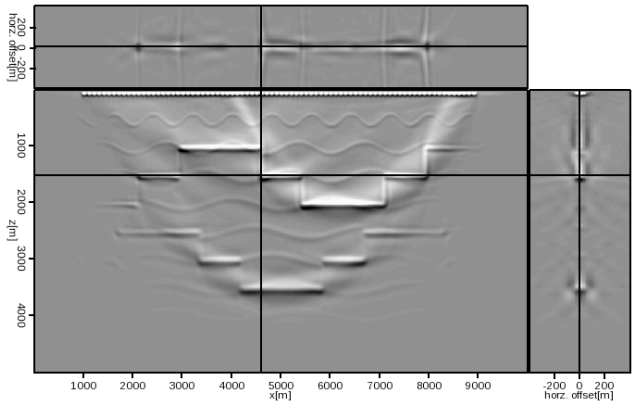

Figure 2: Horizontal subsurface offset gather.

CUDA PROGRAMMING MODEL

Compute Unified Device Architecture (CUDA) is a programming platform for massively parallel computation. Codes built with CUDA can run on Nvidia's graphics 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 are grouped together in thread blocks. Threads within a thread block can share data using shared memory and synchronize using barrier. That however is not possible between thread blocks. Calling a kernel from the host code will launch the thread blocks in an 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 number of 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 (NVIDIA, 2008) 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 in size.

WAVE PROPAGATION KERNELS

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

$$P^{t+1} = v^2 \Delta t^2 \left(f^t + \nabla^2 P^t \right) + 2P^t - P^{t-1}. \tag{5}$$

We use similar approach to the one implemented by Micikevicius (2009), but we generalize it to variable grid spacing for each dimension. We also have separate kernels for source function injection/boundary condition. 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 Micikevicius (2009).

The problem grid is divided into a grid of 16x16 blocks as illustrated in Figure 3. Each grid block is assigned to a thread block of the same size; i.e. each thread performs the finite differencing on a single point on the grid. Since data sharing is allowed between threads within a thread blocks, shared memory is used to keep local copies of P^t that are needed for computing the spatial derivatives. Each thread within a thread block loads the corresponding point on the grid of P^t into shared memory. Because the derivative stencil requires eight neighboring points in each spatial dimension, some of the threads are assigned to load the halos, i.e. the surrounding points of the thread block.

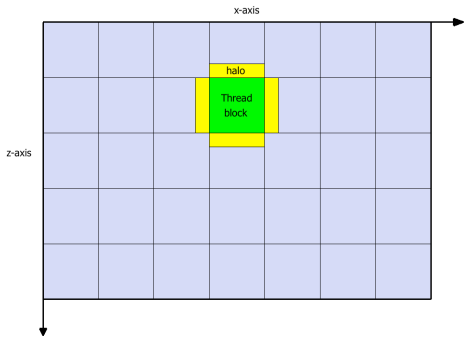

Figure 3: Thread blocks of wave propagation kernel are mapped to areal blocks of the domain. A thread block has to read the assigned grid points (green area) and halos (yellow) from neighboring points to shared memory.

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 that computes $\mathbf{I}(\mathbf{x}, h_z)$:

```
__global__ void img_hz_kernel(float *p1 , float *p2){
  int i=blockIdx.x*blockDim.x+threadIdx.x;            /* z image location */
  int j=blockIdx.y*blockDim.y+threadIdx.y;            /* x image location */
  for(int h=hmin; h<hmax; h++)
    img_zh[iloc(i,j,h)]+=p1[loc(i-h,j)]*p2[loc(i+h,j)];/* imaging condition*/
}
```

Grid blocking of image locations (z, x) is similar to the one shown for wave propagation. The iteration occurs along the offset axis, h, which means that thread block *extent* is the whole offset axis. This is a naive implementation of the imaging condition because of redundant reads.

The global memory access of the naive imaging kernel is illustrated in Figure 4(a), where for each point in the 3D output space, two values from P_f and P_b are needed. The imaging kernel can be improved to eliminating redundant reads from within a thread block.

For the sake of simplicity, we consider a 1D thread block of dimension n that have an offset extent of n point; i.e. each thread block will compute a tile of size n^2 from the output space (z, h_z). For the naive imaging kernel, the number of global memory accesses is $4n^2$ total accesses; two reads from the wavefields and one read and

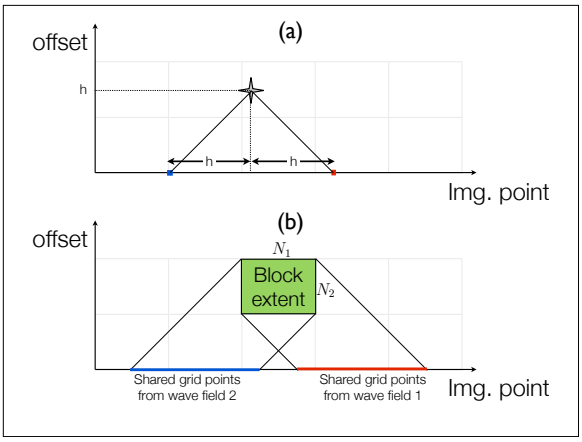

Figure 4: Memory access pattern of imaging: (a) to compute the image value (at the star), two points from P_f (blue) and P_b (red) are needed. (b) Thread block advancement along the offset axis and data sharing between threads within the thread block. All data access from with the thread block will fall within the shared areas from the forward and backward propagated wavefields.

one write for updating the offset gathers. Figure 4(b) illustrates areas that will be accessed from the wavefields. The width for each area is twice the width of the thread block n. This common areas can be copied to the shared memory, so each thread will copy two values from the shared area. After that the thread block advance along the offset gather and use the data from shared memory, as shown using the following kernel:

```
__global__ void img_hz_improved_kernel(int h0 /* first offset */,
                                                    float *p1, float *p2){
  __shared__ float s_p1[BLOCK_SIZE][2*BLOCK_SIZE];
  __shared__ float s_p2[BLOCK_SIZE][2*BLOCK_SIZE];

  int i=hmax_gpu+blockIdx.x*blockDim.x+threadIdx.x;   /* z-location */
  int j=blockIdx.y*blockDim.y+threadIdx.y;            /* x-location */

  /* offset extent of the thread block */
  int hmin=h0-hmax_gpu;                               /* minimum offset */
  int hmax=hmin+BLOCK_SIZE;                           /* maximum offset */

  /* copy to shared memory */
  s_p1[threadIdx.y][threadIdx.x]=p1[loc(i-hmax, j)];
  s_p2[threadIdx.y][threadIdx.x]=p2[loc(i+hmin, j)];
  s_p1[threadIdx.y][BLOCK_SIZE+threadIdx.x]=p1[loc(i-hmin, j)];
  s_p2[threadIdx.y][BLOCK_SIZE+threadIdx.x]=p2[loc(i+hmax, j)];

   /* synchronize threads */
```

```
  __syncthreads();

  /* prepare addressing variables */
  int base=loc(i,j)+h0*n1*n2;
  int stride=n1*n2;

  /* internal loop along the offset axis */
  for(int h=0; h<BLOCK_SIZE; h++){
    zimg[base+h*stride]+=
        s_p1[threadIdx.y][BLOK_SIZE+threadIdx.x-h]
       *s_p2[threadIdx.y][threadIdx.x+h];
  }
}
```

The cost for the improved algorithm is $2n^2 + 4n$. For a thread block of width 16, the number of global memory accesses is reduced from 1024 to 576.

KERNELS PERFORMANCE

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

$$C_{\text{RTM}} = (C_{\text{forward}-\text{FDTD}} + C_{\text{backward}-\text{FDTD}})N_x N_z N_t N_{shots} + C_{\text{Imaging}} N_I N_x N_z N_h N_{shots}, \tag{6}$$

where $C_{\text{forward}-\text{FDTD}}$, $C_{\text{backward}-\text{FDTD}}$, C_{Imaging} are the costs for forward, backward wave propagation, and imaging (both horizontal and vertical), respectively. 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 propagation 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 millions of output points per second (Mpts/sec).

The throughput of RTM kernels are shown in Figure 5 and their relative execution times are shown in Figure 6. For comparison, thread blocks for imaging kernels were 16x16 for both kernels, and the improved kernel has offset extent of 16. The throughput of the improved kernel is double the throughput of the naive implementation. Modifying the block sizes yielded negligible return in performance. This optimized kernel is applicable to 3D. However, 4GB of GPU memory is not sufficient to host the 4D output volume.

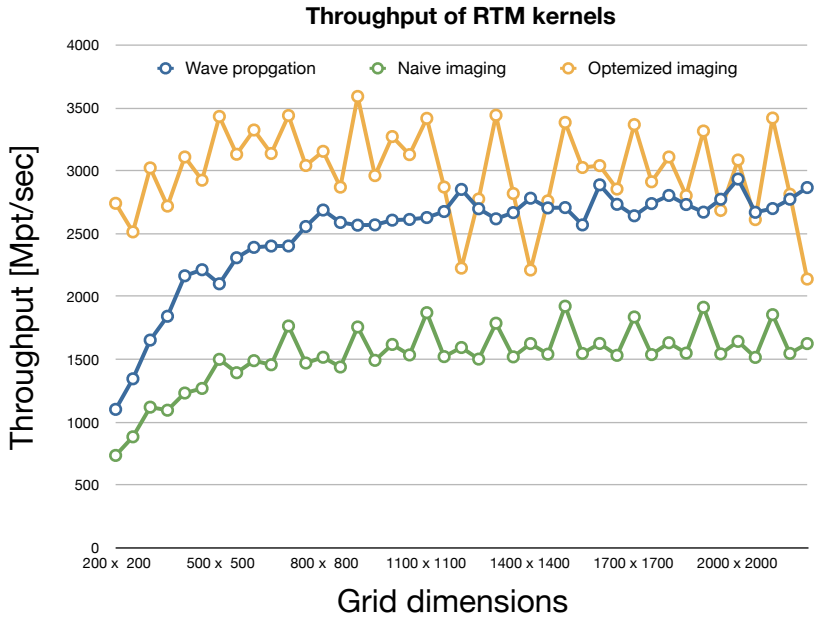

Figure 5: Performance of fully offloaded 2D RTM to a single GPU.

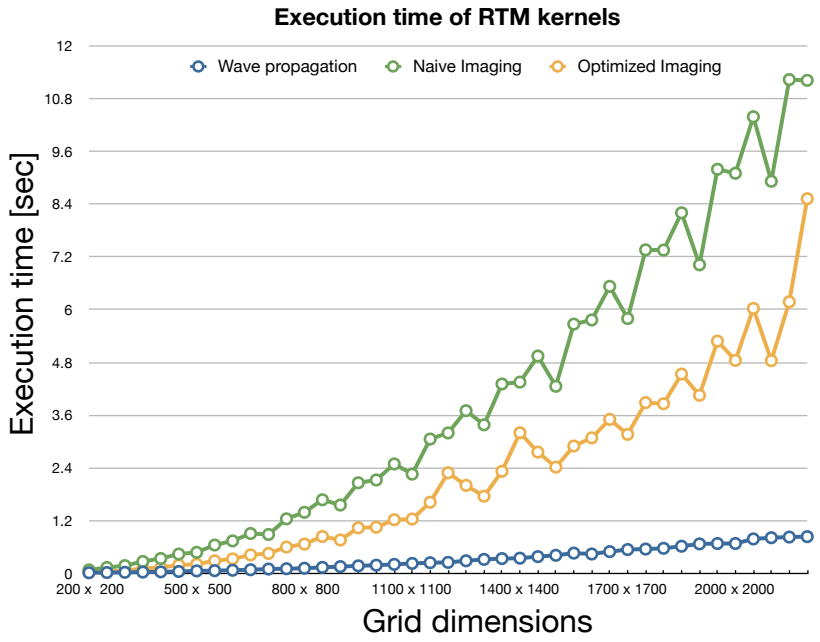

Figure 6: Execution time of RTM kernels for a single shot with 1000 time steps, 100 imaging steps, and 81 points along the offset axis. Because of large output of imaging (81x grid size) per imaging steps compared to finite difference (20x grid size) most of the RTM computing time is spent on generating subsurface offset gathers.

CONCLUSIONS AND FUTURE DIRECTION

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 50% of the naive implementation. This implementation is applicable to 3D, although available memory per GPU is not sufficient to hold image volumes. We consider generating horizon based image gathers instead.

ACKNOWLEDGMENT

We would like to thank Nvidia for providing the Tesla S1070 GPUs for SEP. We also thank Paulius Micikevicius for sharing his CUDA implementation of FDTD.

REFERENCES

Biondi, B. and W. W. Symes, 2004, Angle-domain common-image gathers for migration velocity analysis by wavefield-continuation imaging: Geophysics, **69**, 1283–1298.

Micikevicius, P., 2009, 3d finite difference computation on gpus using cuda: GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, 79–84, ACM.

NVIDIA, 2008, Nvidia cuda programming guide 2.0.

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