Performance of RTM with ODCIGs computation fully offloaded to GPU

Abdullah Al Theyab and Robert G. Clapp

ABSTRACT

Nvidia's graphics processing units (GPU) powered with Compute Unified Device Architecture (CUDA), the supporting API, have allowed a significant speedup to finite difference time domain (FDTD) seismic modeling and, consequently, to reverse time migration (RTM). To utilize the power of GPUs for velocity analysis, we implemented kernels for generating offset-domain common image gathers (ODCIGs). With 4GB of memory, a single Tesla 10 series GPU can perform the 2D RTM with generation of ODCIGs. Computing the ODCIGs takes the majority of the algorithm execution time because of the large volume of output. We examine the performance of a general 2D RTM algorithm with ODCIGs computation fully offloaded to a single GPU device. We optimized the imaging kernel utilizing the available shared memory on the GPU to double the throughput of the kernel.

INTRODUCTION

Reverse time migration (RTM) is a full wave equation imaging technique that constructs an image that best represents the subsurface structure. Seismic data are migrated using 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). In addition, they can give amplitude versus offset (AVO) information if amplitudes are handled properly. Algorithms for RTM and generation of ODCIGs are known to be computationally expensive and sometimes unaffordable. Therefore, an efficient implementation of RTM with ODCIGs generation algorithm is vital for minimizing the time required for creating a complete image.

Reverse time migration falls into the computational class of convolution with a stencil. The 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 of spatial and temporal locality and/or the data prefetch capabilities of modern CPUs. Another way of hiding memory latency is to have more threads than cores to execute some threads while other threads are waiting for memory access. The performance gain given 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 capability makes GPUs very attractive hosts for stencil computational problems.

Micikevicius (2009) has shown an order of magnitude increase in performance for the GPU's FDTD kernel as compared to the kernel's performance on multi-core CPUs. The reported performance numbers are optimized for 3D by using equal grid spacing in all dimensions and without using the surface and absorbing boundary conditions.

Bus connection speed can be a performance bottleneck for mixed CPU-GPU processing. Generating subsurface offset gathers on the host using wavefields computed on the GPU will slow down the algorithm significantly. Fortunately, this can be avoided by fully offloading RTM and ODCIGs generation to GPUs.

In this report, we review briefly the theory of RTM for an acoustic medium. We detail the GPU implementation of RTM with generation of ODCIGs. We then analyze the performance of GPU kernels on a single device. Finally, we optimize the imaging kernel by using shared memory to double the throughput. The developed 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 the response at many receiver stations $(r_1, r_2, ...)$, with each receiver r_i positioned at $\mathbf{x}_{r_i}$. The data from one experiment are collected into a *shot gather* $D_s(r_i, t)$. The 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 with a zero source term. The first wavefield is the *forward* propagated wavefield $P_f(\mathbf{x}, t)$, which is simulated using the boundary condition

$$P_f(\mathbf{x}, t) = \delta(\mathbf{x} - \mathbf{x}_s) \int_0^t f_s(t')dt' \tag{2}$$

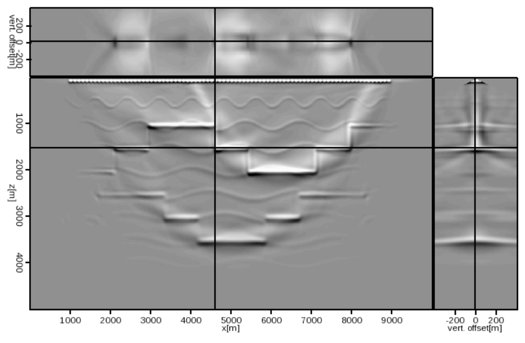
and a zero boundary condition above the Earth's surface (Zhang and Sun, 1993). Here, $f_s(t')$ is the source signature. The second wavefield is the *backward* propagated wavefield, computed using the upper boundary condition

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

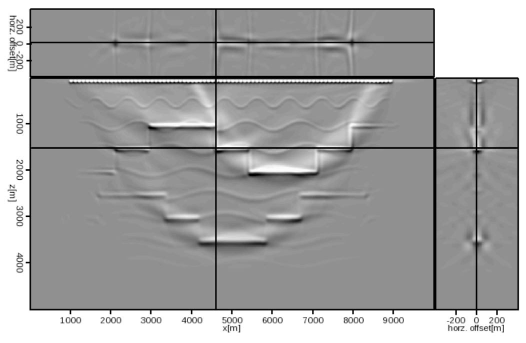
The final image is computed using

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

where $\mathbf{h}$ is the subsurface offset (Biondi and Symes, 2004). In practice, $\mathbf{h}$ is sampled in the horizontal direction, producing horizontal ODCIGs $\mathbf{I}(\mathbf{x}, h_x)$, and in the vertical direction to produce the vertical ODCIGs $\mathbf{I}(\mathbf{x}, h_z)$. For a 2D imaging problem, the horizontal and vertical ODCIGs are two 3D image volumes. Figures 1(a) and 1(b) show respectively slices through the vertical and horizontal subsurface offset cubes for a 2D synthetic dataset migrated using the correct velocity model.



(a)



(b)

Figure 1: **(a)** Vertical ODCIGs. **(b)** Horizontal ODCIGs. The energy is focused at the zero-offset because the data were migrated using the correct velocity. [**CR**]

CUDA PROGRAMMING MODEL

Compute Unified Device Architecture (CUDA) is the GPU supporting API extension to the C programming language. A GPU has many multiprocessors, each with its own set of stream processors and shared memory. A GPU also has 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 by using shared memory and can synchronize by using barriers. However, data sharing and synchronizing are 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 limits the maximum number of threads per thread block. Moreover, the fixed size of the fast shared memory and the number of registers limit the number of the thread blocks that run concurrently on a single multiprocessor. For stencil computation, 2D thread blocks are mapped to the data grid, which means that each thread block handles a tile of grid points. The optimal thread block size is 16x16, which is determined by the device instruction set (NVIDIA, 2008) and the available shared memory.

Many considerations have to be taken into account when optimizing CUDA kernels. Coalesced global memory access can radically reduce memory access instructions. Shared memory usage can also significantly reduce global memory access. However, memory bank conflicts can hinder the performance gain from using the 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 were run on a Tesla 10-series GPU (Tesla S1070). This GPU contains 30 multiprocessors, each with 8 streaming processors and 16 KB of shared memory. The memory bandwidth is 104 GB/s, and the global memory is 4GB in size.

PERFORMANCE METRICS

The total cost of the 2D RTM with ODCIGs generation algorithm for a grid with the size $N_x \times N_z$ and N_t time steps is

$$C_{\text{total}} = (C_{\text{forward-FDTD}} + C_{\text{backward-FDTD}})N_x N_z N_t N_{shots} + C_{\text{Imaging}} N_I N_x N_z N_h N_{shots}, \tag{5}$$

where $C_{\text{forward-FDTD}}$, $C_{\text{backward-FDTD}}$, and C_{Imaging} are the costs for forward wave propagation, backward wave propagation, and imaging, respectively. N_I is the number of imaging steps, and N_h is the number of points in the subsurface offset axis. FDTD cost includes the cost of setting the surface boundary condition. Because of the asynchronous execution of backward propagating kernels and imaging kernels, it is difficult to isolate their costs. We estimate that $C_{\text{forward-FDTD}} = C_{\text{backward-FDTD}}$, where $C_{\text{forward-FDTD}}$ can be calculated by timing the forward propagation part of the algorithm. Therefore the throughput metrics for wave propagation and imaging kernels are

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

Both are expressed in millions of output points per second (Mpts/sec).

WAVE PROPAGATION KERNELS

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

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

We use an approach similar to the one implemented by Micikevicius (2009), but we generalize it to accommodate variable grid spacing for each dimension. We also have separate kernels for the source function injection. The waves incident on the grid boundaries are attenuated by adding an absorption term to the wave equation. This term is active around the neighborhood of the side and bottom boundaries. These inclusions to the algorithm increase the number of floating point operations and the number of memory accesses, which in turn reduce the throughput of FDTD by a few hundreds of Mpts/sec as compared to the reported performance measures by Micikevicius (2009).

The problem grid is divided into a grid of 16x16 blocks as illustrated by Figure 2. 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 block, 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.

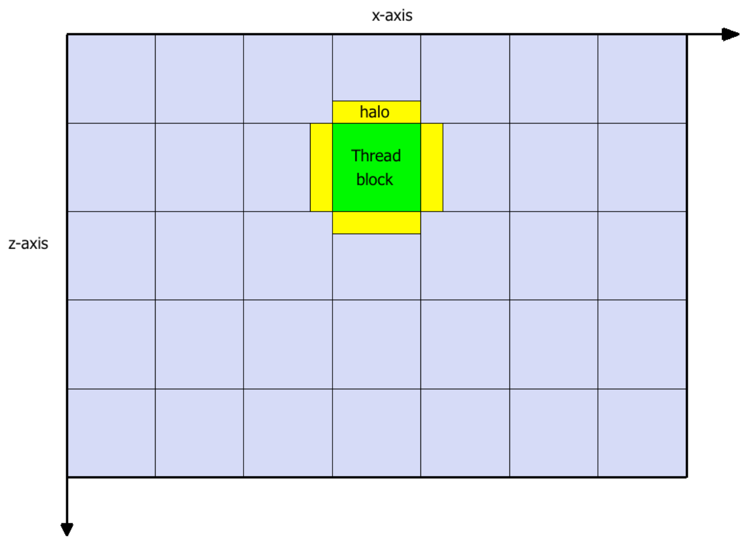


Figure 2: Thread blocks of the wave propagation kernel are mapped to areal blocks of the domain. A thread block has to read the assigned grid points (green area) and halos (yellow) from neighboring points to shared memory. [**NR**]

IMAGING KERNEL

The imaging equation (4) is used in practice to generate two ODCIG cubes with the two x- and z-spatial dimensions and a 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 the thread block *extent* is the whole offset axis. This is a naive implementation of the imaging condition because of the redundant reads from global memory. The global memory access pattern of the naive imaging kernel is illustrated in Figure 3(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 so that redundant reads from within a thread block are eliminated.

For the sake of simplicity, we consider a 1D thread block of dimension n that has an offset extent of n points; 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 $4n^2$; i.e., to update a point in the output volume (offset gathers), two reads from the wavefields, and one read and one write for updating the offset gathers are needed. Figure 3(b) shows areas that will be accessed from the wavefields. The width for each area is twice the width of the thread block n. The common areas can be copied to the shared memory, so that each thread will copy two values from the shared area. After that, the thread block advances along the offset axis and uses 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)];
```

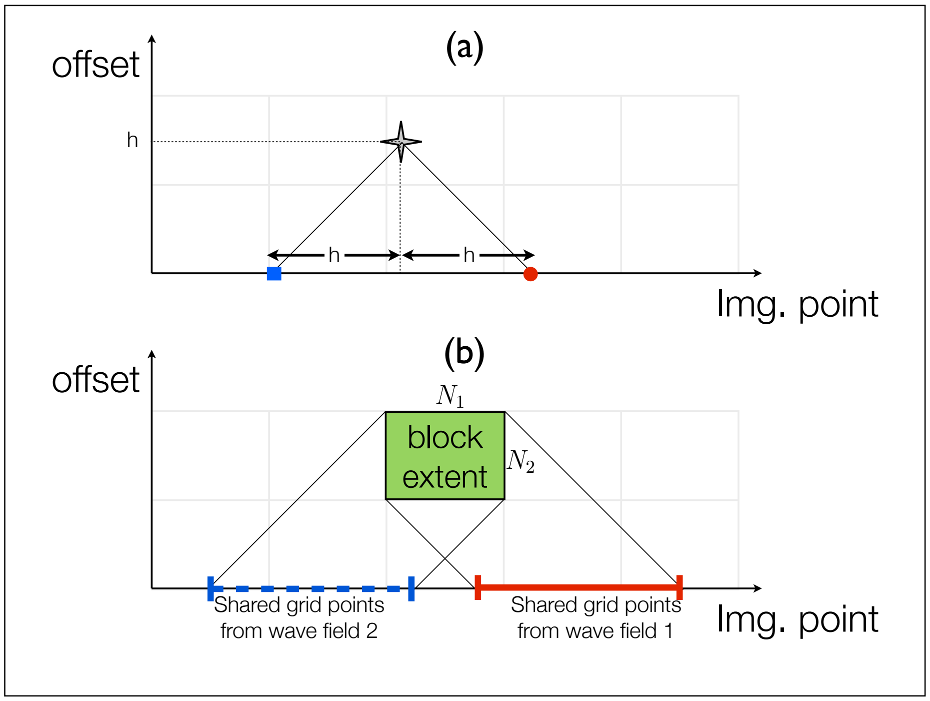

(a)
offset
h
h
h
Img. point
(b)
offset
N_1
block extent
N_2
Shared grid points from wave field 2
Shared grid points from wave field 1
Img. point

Figure 3: The memory access pattern of imaging: (a) To compute the image value (at the star), two points from P_f (square) and P_b (circle) are needed. (b) This diagram shows the thread block advancement along the offset axis and data sharing from within the block. All data access from with the thread block will fall in the shared areas from P_f (dashed) and P_b (solid). **[NR]**

```
  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 number of global memory accesses 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 (naive) to 576 (improved).

KERNELS' PERFORMANCE

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

Figure 5 shows the execution time for RTM without ODCIGs generation, RTM with ODCIGs generation using the naive imaging kernel, and RTM with ODCIGs generation using the improved imaging kernel. Computing ODCIGs is very costly because of the large volume that is updated at every imaging step. The floating-point operation count is very small for the imaging kernel. This indicates that most of the execution time for ODCIGs generation is spent accessing and writing to the device global memory. The memory access time is so dominant that constructing angle-domain common image gathers (ADCIGs) at every time step could be considered.

SUMMARY AND FUTURE DIRECTIONS

In this paper, we have shown the implementations of the naive imaging kernel. We also demonstrated a strategy for improving the imaging kernel that doubles the through-

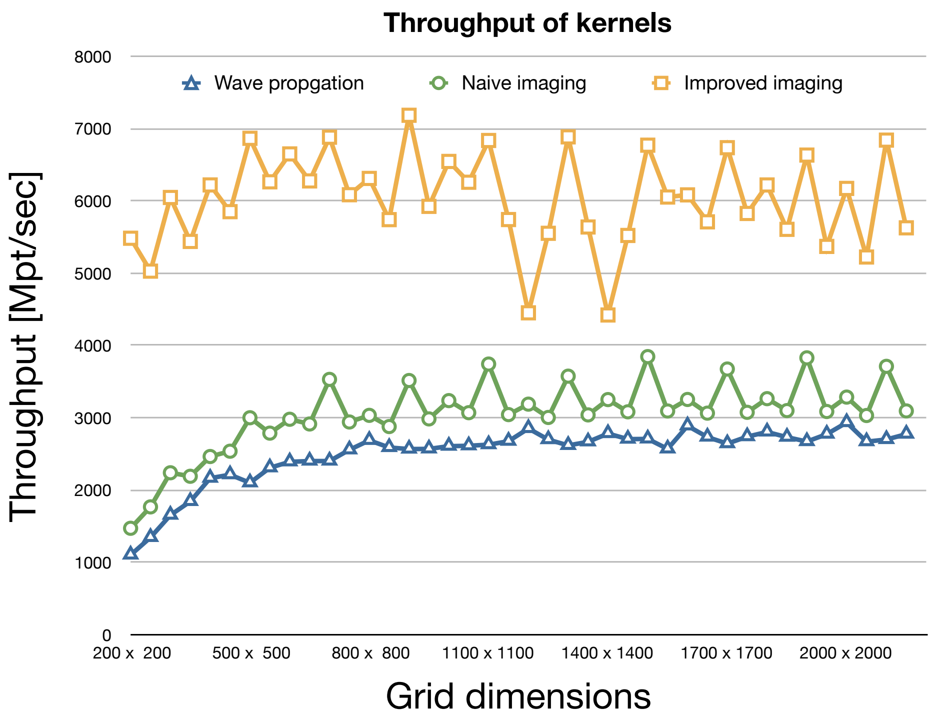


Figure 4: Throughput of the implemented kernels. Improving the imaging kernel using shared memory doubles the throughput of the imaging kernel. [**NR**]

put of the naive implementation. This implementation is applicable to 3D imaging problems, although the available memory for current GPUs is not sufficient to hold the generated image volumes. We are considering generating horizon-based ODCIGs for 3D wave propagation. Due to the large output volume, generating ODCIGs is costly as compared to RTM without ODCIGs generation. To reduce the cost, we are considering data compression and subsampling the ODCIGs, which will reduce the output volume that is updated at every imaging step. The cost of generating ADCIGs will be minimal compared to generating ODCIGs. Therefore, we are considering generating ADCIGs on the GPU.

ACKNOWLEDGMENTS

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.

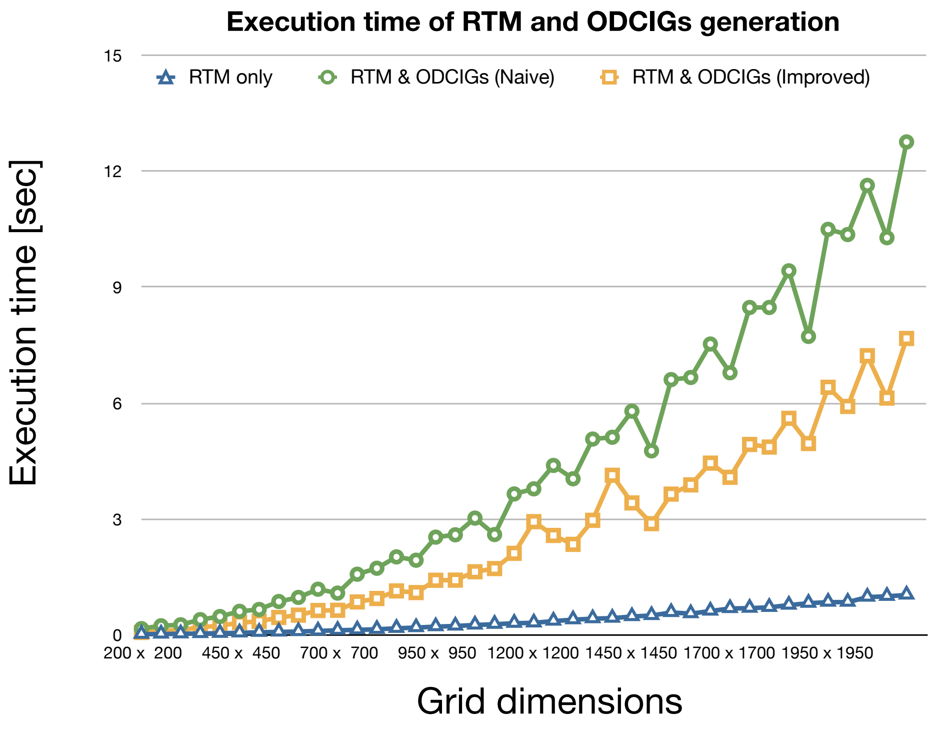


Figure 5: Execution times for migrating a single shot with 1000 time steps and 100 imaging steps. There are 81 points along the z- and x-offset axes for the generated ODCIGs. Because of the large volume of ODCIGs ($2 \times 81 \times$ grid size) that is updated at every imaging step, most of the computing time is spent on generating the gathers. [**NR**]

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.