Shortest path ray tracing on parallel GPU devices[a]

[a]Published in SEG Expanded Abstracts, 3470-3474, (2013)

Jorge Monsegny, UP Consultorias and William Agudelo, ICP-Ecopetrol

ABSTRACT

A new parallel algorithm for shortest path ray tracing on graphics processing units was implemented. This algorithm avoids the enforcing of mutual exclusion during path calculation that is found in other parallel graph algorithms and that degrades their performance. Tests with velocity models composed of millions of vertices with a high conectivity degree show that this parallel algorithm outperforms the sequential implementation.

INTRODUCTION

Shortest path ray tracing is a method to trace rays using a graph that represents the velocity model. The vertices of the graph have defined locations in the velocity model. The edges between vertices have weights associated with the traveltime of a seismic ray that crosses from one of its adjacent vertices to the other. It is an instance of the single source shortest path problem (SSSP) that is classic in graph theory. Finding the shortest path from one vertex to another using these weights is an aproximation to the seismic ray between them by Fermat's principle (Moser, 1991). This approximation will get closer to the seismic ray when the vertex and edge coverage is dense, although this is computationally expensive. This method was introduced in Nakanishi and Yamaguchi (1986) and Moser (1991).

This ray tracing method has some advantages (Moser, 1991). It can calculate raypaths from one vertex to all other vertices in the velocity model simultaneously. Traditional restrictions of other methods like shadow zones and difracted raypaths are properly handled, and the velocity model can be complex. As the resulting raypath is only an aproximation, it can serve as a very good starting point to other more precise methods.

One of the main drawbacks of this method is its calculation velocity (Leidenfrost et al., 1999). If it is properly implemented with priority queues its computational complexity is $O(n \log n)$ where n is the number of vertices. Nevertheless, some implementations report almost an order of magnitude more time consumed that other alternatives like finite differences eikonal solvers and wavefront construction (Leidenfrost et al., 1999).

Graphics processing units (GPUs) are a commodity programable hardware that have found a place in scientific computation. They can run concurrently millions of threads with very cheap context switch and high computational throughput. It is possible to decompose the shortest path calculation in a such a way that a GPU can handle this calculation. Indeed, some have used GPUs to solve the SSSP problem in general graphs with millions of vertices, although some approaches require atomic memory operations (Harish and Narayanan, 2007), which are expensive in GPU devices. It will be show that there is a solution that does not need this kind of memory operations.

METHOD

The velocity model is represented as a weighted graph $G = (V, E, W)$ where V is the set of vertices, E the set of edges and W the set of edge weights. Each vertex depicts a location (x, z) in the velocity model in such a way that all locations form a regular rectangular grid that covers the velocity model extension. Figure 1 shows a three layer velocity model. The black dots are the vertex locations. We can also set a double index (i, k) to each vertex based in its column and row in the rectangular grid. As there is a one to one correspondence between vertices and indices, hereafter we will not distinguish between them. The horizontal and vertical separations between locations are dx and dz, respectivelly. For simplicity, we will set $dx = dz$.

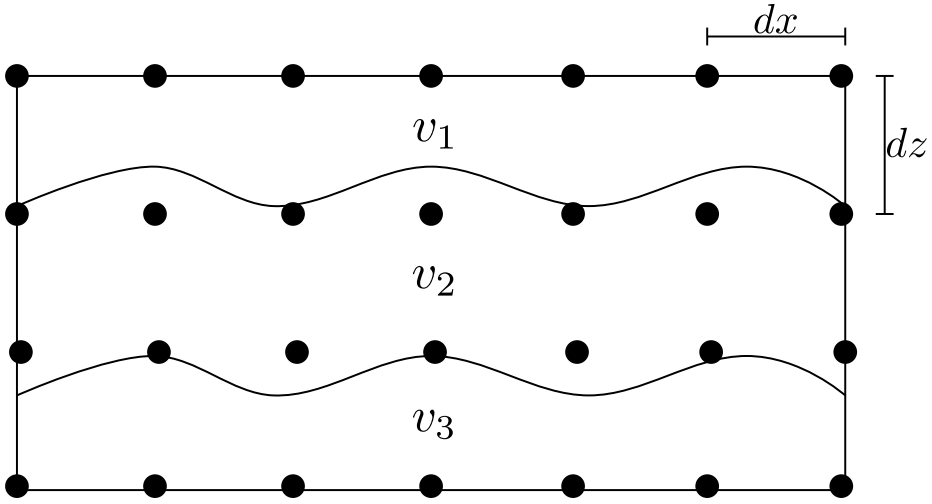

Figure 1: Rectangular grid of vertices in a velocity model.

Graph edges are defined between neighboring vertices. There are various ways to accomplish this. One way is to choose a rectangular neighborhood around the current vertex and define an edge between every other vertex inside the neighborhood and the current one. In Figure 2 displays two of such neighborhoods, one of radius 1 and the other of radius 2. A neighborhood with radius r contains $(2r + 1)^2 - 1$ vertices.

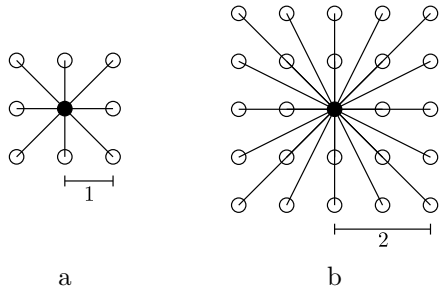

Figure 2: Two vertex neighborhoods. (a) radius 1 (b) radius 2.

Weight precalculation

The weight of each edge is the traveltime between its adjacent vertices. This traveltime is calculated by integrating the velocity model along the spatial location of the edge. As the graph weights are going to be used several times during the ray tracing we choose to precalculate them. Algorithm 1 is a kernel function that performs this precalculation concurrently.

Algorithm 1 PrecalculateWeights(v,w)

1: $(col, row) = getThreadIdx$
2: **for all** neighbors (i, k) of (col, row) **do**
3: $\quad w[col][row][i][k] = calctt(v, col, row, i, k)$
4: **end for**

As this kernel is launched for each vertex, the first step is to obtain the indices (col, row) of the current thread. Next we calculate the traveltime from vertex (col, row) to each of its neighbors (i, k). The array w will keep these values.

Sequential main function

Algorithm 2 is the sequential main function that call the parallel kernels to perform the ray tracing. Its parameters are the indices (si, sk) of the source vertex. The array v holds the velocity values at each vertex location, obtained from the velocity model. The arrays tt and $auxtt$ will be used to store the traveltime from source vertex to each other vertex. At the beginning they are set to ∞ (or another big value) except for the traveltime of source vertex that is set to $tt[si][sk] = 0$. The arrays pr and $auxpr$ will contain the indices of the predecesor vertex of each vertex along the ray. Their starting values are $(-1, -1)$ for all vertices. At this point we call the kernel function to precalculate the weights of each vertex. The ray tracing is conducted next by calling the two function kernels Relaxation and WriteBack in a loop while the boolean variable *stop* is false. The reason to have divided the work among these two kernels is that it is necessary to synchronize all threads after the work done by

first kernel and this global synchronization is only possible by having the rest of the code in another kernel.

Algorithm 2 MainFunction(si,sk)

1: Store the velocity at each vertex (i, k) in $v[i][k]$.
2: Set $tt[i][k] = auxtt[i][k] = \infty$ for each vertex (i, k).
3: $tt[si][sk] = 0$
4: Set $pr[i][k] = auxpr[i][k] = (-1, -1)$ for each vertex (i, k).
5: PrecalculateWeights(v,w)
6: $stop = false$
7: **while** *stop* is false **do**
8: $stop = true$
9: Relaxation(tt,auxtt,w,pr,auxpr)
10: WriteBack(tt,auxtt,pr,auxpr,stop)
11: **end while**

Relaxation

Algorithm 3 is the kernel Relaxation. It takes care of finding the smaller traveltime to each vertex up to the current iteration. This kernel is launched for each vertex, so in a similar way to the kernel function that precalculates the weights, it starts by obtaining the current vertex indices (col, row). The following step is to iterate each neighbor (i, k) of vertex (col, row). During each iteration it calculates the following expression:

$$newtt \quad = \quad tt[i][k] + w[col][row][i][k] \tag{1}$$

Figure 3 shows schematically this calculation. It is the current smaller traveltime from source (si, sk) to neighbor vertex (i, k) plus the traveltime from this vertex to vertex (col, row). This is a candidate smaller traveltime from source vertex to (col, row) passing through (i, k). If this traveltime is smaller than the current smaller traveltime from source vertex to vertex (col, row), we have to actualize its value and record that this new smaller traveltime is reached through vertex (i, k)

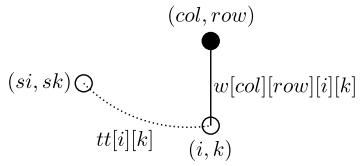

Figure 3: Traveltime compared during relaxation kernel.

It should be noted that during this operation each kernel thread, although reads information from many differente vertices, only modifies information of its own vertex

(col, row) and no other kernel thread modify this vertex. This is the main difference with the shortest path algorithm of Harish and Narayanan (2007), that at this point calculates the expression:

$$newtt \quad = \quad tt[col][row] + w[col][row][i][k], \tag{2}$$

i.e. the sum of the traveltime from source to vertex (col, row) and the traveltime from this vertex to vertex (i, k). Figure 4 displays schematically this calculation. This value is a tentative smaller traveltime from source to vertex (i, k) passing through (col, row). If this is small that the current smaller traveltime to vertex (i, k) it modifies traveltime and predecessor values for vertex (i, k). The problem is that another kernel thread might be, concurrently, attempting to modify information of the same vertex (i, k) (because vertices usually belong to many neighborhoods) and this can lead to race conditions.

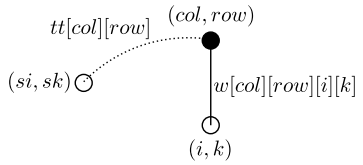

Figure 4: Traveltime compared in Harish and Narayanan (2007).

Algorithm 3 Relaxation(tt,auxtt,w,pr,auxpr)

```
1: (col, row) = getThreadIdx
2: for all neighbors (i, k) of (col, row) do
3:    newtt = tt[i][k] + w[col][row][i][k]
4:    if newtt < auxtt[col][row] then
5:       auxtt[col][row] = newtt
6:       auxpr[col][row] = (i, k)
7:    end if
8: end for
```

The solution of Harish and Narayanan (2007) is to use an atomic function that compares and possibly stores in one single instruction the smaller traveltime in $tt[i][k]$. This strategy avoids the race condition but has two drawbacks. First, atomic functions slow down kernel execution. And second, it only allows to write a single value in an atomic operation, but we need to write two values: traveltime and predecessor. To solve this last problem it is possible to implement and use mutexes to do all the actualizations avoiding the race conditions, but this only slows down even more the kernel execution, as mutexes are usually implemented using atomic functions.

Another detail to take into account is that during the traveltime and predecesor actualization we have been setting the auxiliary arrays *auxtt* and *auxpr* instead of

the arrays *tt* and *pr*. The reason is that in the current kernel iteration other threads are still using the current values of *tt* and *pred* and we only want to use the new values in the next invocation of this kernel for consistency.

Writing back

Algorithm 4 is the kernel WriteBack. This kernel actualizes the arrays *tt* and *pr* from auxiliary arrays *auxtt* and *auxpr* if during last execution of Relaxation a smaller traveltime was discovered. Additionally, if at least one vertex has changed its traveltime, the variable *stop* is set to false. This will make the main function (Algorithm 2) to execute one more time the while loop.

Algorithm 4 WriteBack(tt,auxtt,pr,auxpr,stop)

1: $(col, row) = getThreadIdx$
2: **if** $tt[col][row] > auxtt[col][row]$ **then**
3: $\quad tt[col][row] = auxtt[col][row]$
4: $\quad pr[col][row] = auxpr[col][row]$
5: $\quad stop = false$
6: **end if**
7: $auxtt[col][row] = tt[col][row]$

Implementation improvements

One important thing to avoid in programming parallel algorithms for GPUs is thread divergence. Thread divergence occurs when threads executing concurrently follow different execution paths. This behaviour can not be avoided in the conditional structures (if) of Algorithms 3 and 4. Iteration structures (forall) of Algorithms 1 and 3 can also suffer from thread divergence because threads in charge of vertices near the border of the velocity model have fewer neighbor vertices to process. One possible solution is to complete the neighborhoods of these vertices by extending the velocity model with dummy vertices. The weights between these dummy vertices and the normal ones have sufficiently big values to ensure that they are not going to form part of any shortest ray path. This is shown in Figure 5 where the white vertices have been added to the model and the dotted edges have big weights. Kernels in Algorithms 1 and 3 are still only launched for normal model vertices.

Another improvement is to use the device shared memory with data that is going to be read multiple times, because this memory is faster than the global one. Shared memory is only shared inside groups of threads called blocks. To take advantage of it the model is divided into rectangular vertex areas with enough information to be completely and independently processed by each block. Due to the dependency of a vertex on its neighbor vertices, these areas should contain some vertices from adjacent

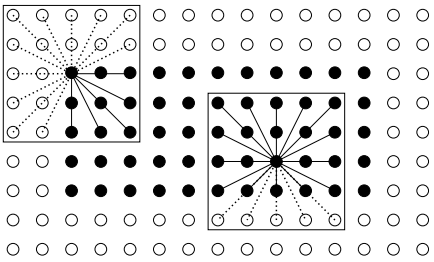

Figure 5: Model extension with dummy vertices to diminish thread divergence.

areas, i.e. the rectangular areas overlap. Figure 6 shows a velocity model and a group of overlapping areas. The stripped zones are the vertices processed by each block. The nonstripped border zones are vertices from adjacent areas that are needed in the current block. This area has a thickness equal to the neighborhood radius r. Grey areas correspond to the dummy vertices added before to reduce thread divergence. This approach is used in Micikevicius (2009) to compute 3D finite differences, but with a different overlapping shape.

In Algorithm 1 velocity values in array v are read multiple times, hence that array is a good candidate to be loaded in shared memory. Before the iteration structure in this kernel function each thread reads once from global memory the velocity of its vertex in array v and stores it in a shared array (this is the stripped zone in Figure 6). Some threads are also assigned to copy the overlaping zones of v, i.e, the nonstripped zone in Figure 6. The same technique can be applied in Algorithm 3, but with the array tt of traveltimes. There is no need to copy the weight array w to shared memory in this kernel function because each of its elements is read once.

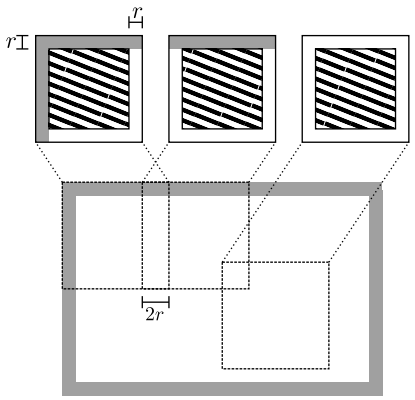

Figure 6: Model partitioning in overlaping areas.

RESULTS AND DISCUSSION

It is performed a test to compare the velocities of GPU parallel and CPU sequential shortest path ray tracing algorithms with different model sizes and different neighbor-

hood radii. The sequential and parallel versions were executed in an intel® icore™ i7 CPU equipped with a nvidia® geforce® gt 650m GPU. Although it is not a high-end GPU device it can be useful to show the performance of the parallel ray tracer and better performance is expected from more advanced GPUs. This GPU has the following relevant specifications:

Global memory 2048 MBytes.

Computing cores 384.

Memory bus width 128 bytes.

Shared memory per block 49152 bytes.

Threads per block 1024.

There were built 16 velocity models with dimensions ranging from 128×128 vertices to 1600×1600 vertices to test the GPU parallel program performance. All models were a simple vertical gradient with $0.5km/s$ on the top and $4km/s$ on the bottom.

The GPU function kernels were made for a block size of 16×16 threads to meet the maximum number of threads per block and the maximum shared memory per block. The measured times account for all GPU operations: data transfers to and from device, weight precalculation and ray tracing.

The sequential version of the shortest path ray tracer was also tested using the same set of velocity models to provide a point of comparison. This sequential version was implemented with the standard library priority queue that provides a fast implementation of this data structure that dominates the velocity of this algorithm. Both ray tracers, sequential and parallel, were compiled with -O4 optimization flag.

In Figure 7 are shown the time results for three neighborhood radii: $1, 2$ and 3. The CPU version outperforms the parallel one in the first two radii and is almost as good in the third. The reason is that in calculations with low radius values there are not enough operations to keep the GPU busy while waiting for data transfers from global to shared memory.

Figure 8 displays the time results for other three neighborhood radii: $4, 5$ and 6. In these cases, the parallel implementation is faster than the sequential one, with a speedup factor between 2 and 3 for the highest radius. In these cases the high number of operations per thread was enough to hide the latency of global memory. Despite the low capacity of the GPU device used in the test, this is a significant improvement over the sequential solution. More powerful GPU devices can deliver better improvements.

One important thing to consider involves the memory requirements. The most expensive piece of data is the weight array that has $((2r + 1)^2 - 1)/2$ elements for

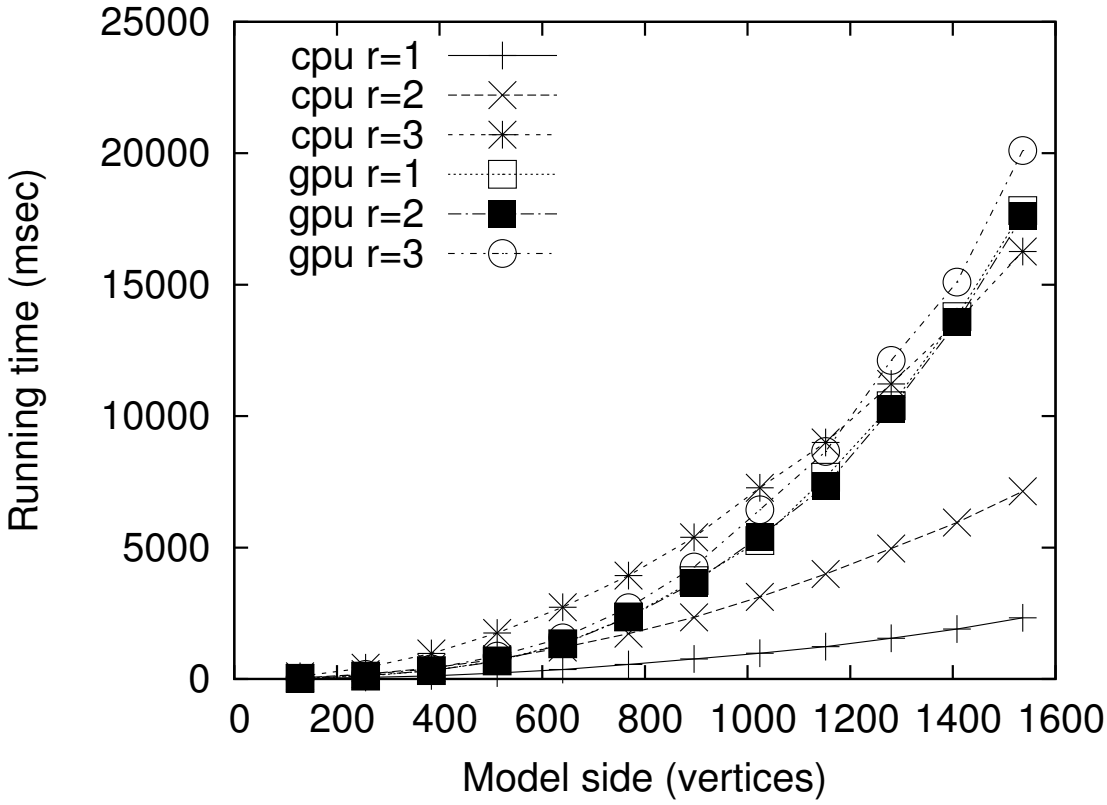

Figure 7: CPU vs. GPU for three neighborhood radius: $1, 2$ and 3.

each model vertex. This imposes a very serious restriction to the model size because of the limited amount of memory in the GPU device. This can be amelioriated with a swapping scheme that loads this array by fragments or by using various GPU devices at the same time. Both possibilities need further experimentation. The possibility of calculating this array on the fly at each iteration is out of the question because it is very time consuming. On the other side, the weight precalculation can be better amortized if varius ray tracings with different sources are going to be performed on the same velocity model, so only one precalculation is needed for all of them.

A mild vertical gradient velocity model is shown in Figure 9. The velocity of this model varies with depth z: $0.5 + 0.005z$ in km/s. Figure 10 shows a composite image of traveltimes to all model positions and some selected rays using the GPU raytracer with size neighbor equal six, in just one run.

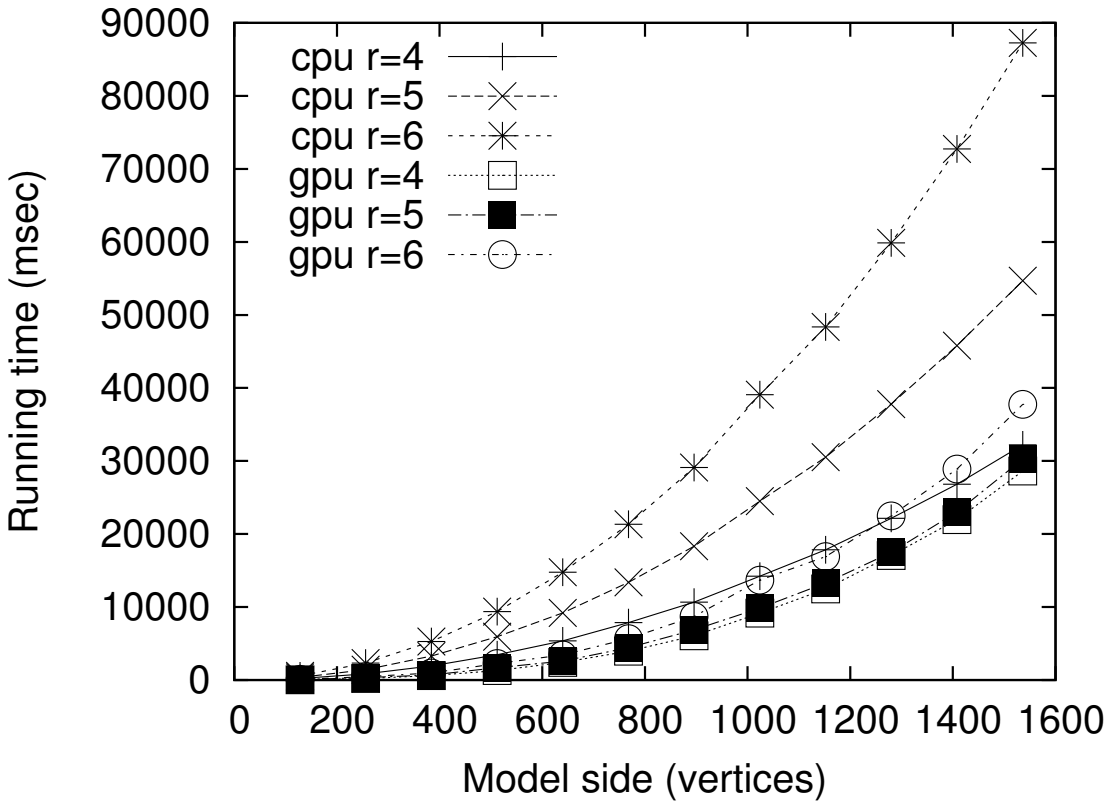

Figure 8: CPU vs. GPU for three neighborhood radius: $4, 5$ and 6.

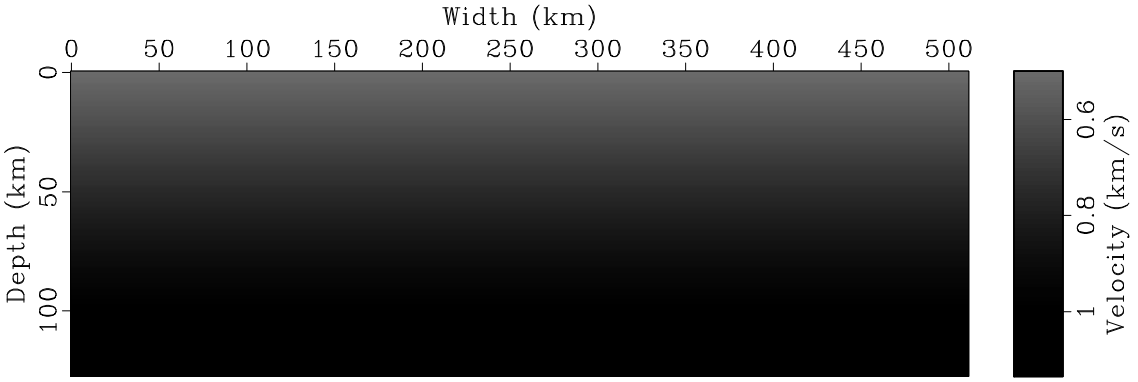

Figure 9: Vertical velocity gradient: $0.5 + 0.005z$ in km/s.

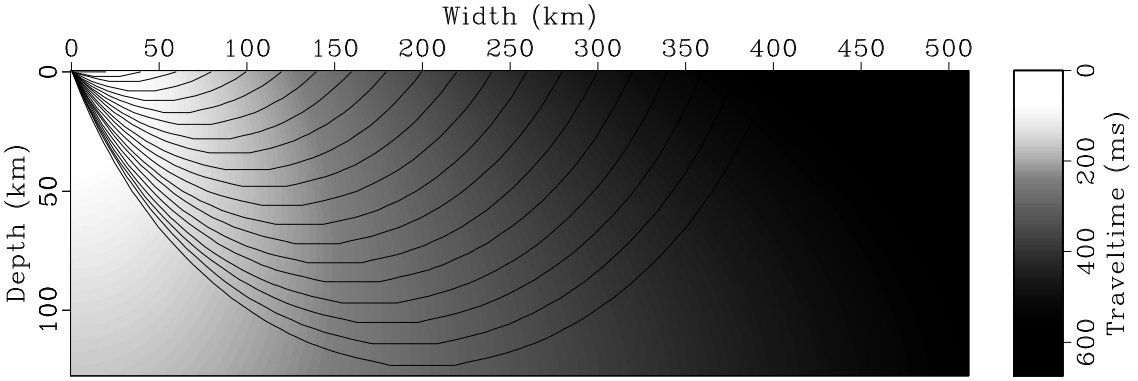

Figure 10: Traveltimes to all velocity model positions and some selected rays. Source is at $(0, 0)$ position.

REFERENCES

Harish, P., and P. J. Narayanan, 2007, Accelerating large graph algorithms on the gpu using cuda: Proceedings of the 14th international conference on High performance computing, Springer-Verlag, 197–208.

Leidenfrost, Ettrich, Gajewski, and Kosloff, 1999, Comparison of six different methods for calculating traveltimes: Geophysical Prospecting, **47**, 269–297.

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

Moser, T., 1991, Shortest path calculation of seismic rays: Geophysics, **56**, 59–67.

Nakanishi, I., and K. Yamaguchi, 1986, A numerical experiment on nonlinear image reconstruction from first-arrival times for two-dimensional island arc structure: Journal of Physics of the Earth, **34**, 195–201.