A Parallel Implementation of Shortest Path Ray Tracing on GPU

Somrath Kanoksirirath

ABSTRACT

This senior project is still in process.

INTRODUCTION

Many problems in physics associate with shortest paths; for instance, Fermat's principle, Hamilton's principle and the eikonal equation. Especially, in geophysics, the eikonal equation plays a crucial role in prediction of source-reciever paths and traveltimes of seismic P-wave in isotropic media (Shearer (2009)). There are many numerical methods for finding the seismic traveltimes, such as Finite-difference method (Vidale (1988)), Fast marching method (Sethian and Popovici (1999)), Fast sweeping method (Zhao (2004)), and Shortest path raytracing (Moser (1991)).

According to Rawlinson and et al. (2007), it can be shown that the eikonal equation is equivalent to shortest path problem.

$$T = \int_L s \, \mathrm{d}l \tag{1}$$

where T is traveltime, L is raypath or the shortest path from source to a receiver, s is slowness or inverse of local wave speed which depends on only medium, and $\mathrm{d}l$ is small displacement.

In order to apply shortest path solvers, we have to convert our problem to a graph or a network. Accoring to Mak and Koketsu (2011), there are two popular network configurations: cell model and grid model—both were suggested by Moser (1991). Each model has its own weakness. A task-parallel implementation of cell model using OpenMP was developed by Giroux and Larouche (2013), while a parallel implementaton of grid model on graphic processing units (GPU) was introduced by Monsegny and Agudelo (2013).

Our implementation is similar to Monsegny and Agudelo (2013) except that we introduce what we call update map and employ edge reduction to reduce computations. In addition, unlike Monsegny and Agudelo (2013), we reject the weight precalculation so that enough GPU memory is avaliable for a very large graph.

METHODS

To remind, we deal with a domain or field where slowness can be derived. A point source position or an initial condition is selected. We assume that no shortest paths run across the area outside the domain—which is equivalent to absorbing boundary condition. Our task is to find the shortest path and its associate traveltime from the point source to all positions in the domain. Next, the graph is constructed.

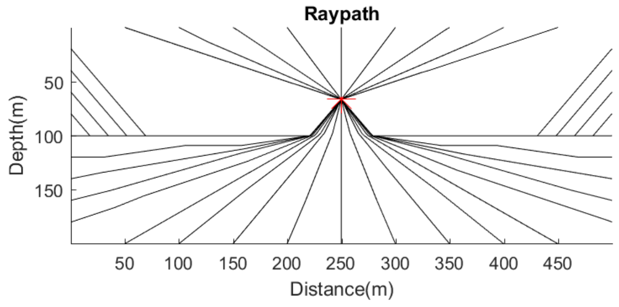

Figure 1: Tempory: Domain+BC+Source+Cells+Nodes.

Graph

Nodes

According to the grid model, the domain is first discretised into small cells. Inside each cell, slowness is uniform. Practically, by interpolation, the number of cells can be larger (or smaller) than the number of input data points. Nodes or vertices of the graph are constructed at the center of each cells (Figure 1).

Edges

Next, neighbors of a node in two dimensions are defined by two parameters, R_x and R_y (both are referred as radius). As shown in Figure 2a, shortest paths from a node to its nearby neighbors are approximated to be straight lines, these straight lines are edges of our graph. Hence, traveling from source to a receiver so far away needs to be done by passing through these short straight lines consecutively. Illustrated by Figure 2b, the larger the size of neighbors or the radiuses—R_x and R_y, the greater the accuracy of the grid model due to more angular choices. Nevertheless, if R_x or R_y is too big, or local slowness values vary suddenly, the approximation above that shortest paths to neighbors are straight lines might be inaccurate. This contradiction is the most prominent weakness of grid model. To be practical, the graph needs to have a large number of nodes with proper radiuses.

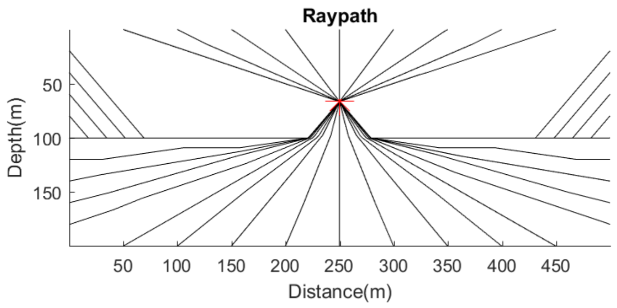

Figure 2: Tempory: (a) Cells+Neighbor+Edges+Reduction+Example of reduction, (b) An example of outcomes due to different radiuses.

Dummy nodes

Obviously, nodes near the border of the graph will have fewer neighbors, in other words, fewer edges. This will complicate our code and, also, lead to more severe problem in parallel computing on GPU called therad divergence (Monsegny and Agudelo (2013)). A simple solution is adding enough cells to all borders (Figure 5)—extending R_x cells on the left and right as well as R_y cells on the top and bottom. To remain the same outcome, their slowness values are infinity. They will be referred as dummy nodes.

Edge reduction

As demonstrate in Figure 2a, the edge connecting to the second neighbors on the right can be removed, as it is equivalent to two consecutive edges—one connecting the node to the first neighbor on the right and another one connecting the first and the second neighbors on the right. In general, those removable edges are the edges that link the node (0,0) to any of its neighbors (i,j) such that the greatest common divisor of i and j is unequal to one—GCD$(i, j) \neq 1$.

Weight

The equation 1 can be discretized to compute the weight of edge as shown below.

$$T = \sum_L s_\mathrm{n}\, dl_\mathrm{n} \tag{2}$$

where, in this content, L is the straight line pointing to the neighbor, dl_n is length of the line segment within cell n, and s_n is slowness value that cell n is holding. Figure 3 provides an example of an edge connecting a node (0,0) to its neightbor (3,1). As mentioned earlier, we decide to recalculate the weight of an edge whenever we need it, since saving every weights of our dense graph would require too large GPU memory space. We propose an efficient algorithm for finding the weight in Appendix A.

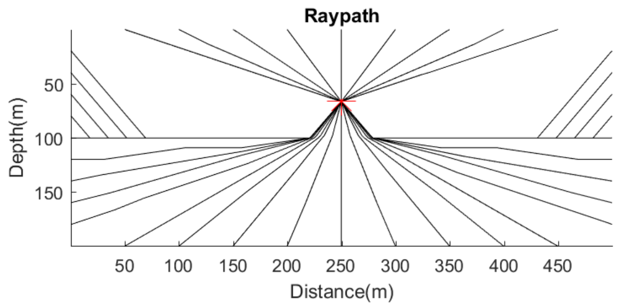

Figure 3: Tempory: Edge (3,1)+Cells+$s_\mathrm{n} + dl_\mathrm{n}$.

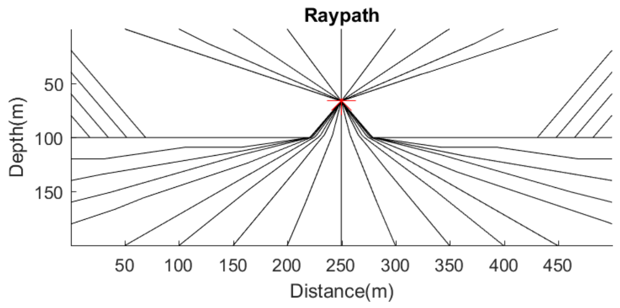

Figure 4: Tempory: Work group + Upload shared memory (four shaded rectangles).

Single-Source Shortest Path algorithm

Before describing our algorithm, we assume that readers deeply understand the Bellman-Ford algorithm as well as the famous Dijkstra's algorithm. Also, OpenGL terminology is adapted here (see Segal (2013)), even though some of CUDA terminology are provided (see Harish and Narayanan (2007)).

Parallel version: Work group

First, we divide nodes of the graph into work groups, the work group size is precisely equal to the neighbor's size. If the number of nodes (without dummy nodes) in two dimensions are $Nx \times Ny$, this restricts that Nx must be divided by $2R_\mathrm{x} + 1$ and, likewise, Ny must be divided by $2R_\mathrm{y} + 1$. This work group is equivalent to OpenGL compute shader's work group and CUDA's thread block, while each node will be each OpenGL's invocation or CUDA's thread.

While finding the shortest paths, slowness values and current candidates of traveltime will be read many times. Most of them are required for many invocations in the same work group. These values, therefore, are uploaded to shared memory of each work group in order to accelerate our program. The transfered variables are covered by shaded rectangles shown in Figure 4. Each invocation uploads four times, some are uselessly uploaded twice.

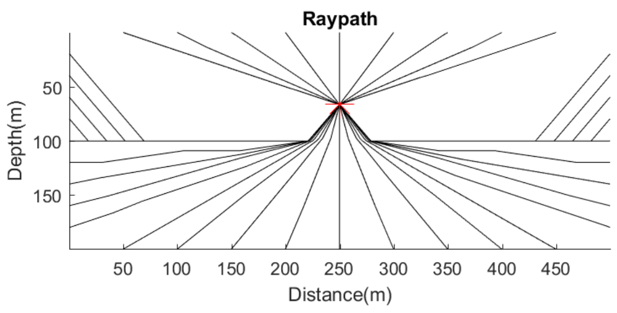

Figure 5: Tempory: Nodes+Dummy nodes+Work group+Dummy work group.

Parallel version: Update map

Although the main idea of the Bellman-Ford algorithm and the Dijkstra's algorithm are completely different, their implementation are very similar. While the Dijkstra's algorithm (Agrawal (2016)) pushes active nodes into a min priority queue and pop out the next accepted node to proceed—accepting the least candidate of traveltime among active nodes—until no active node is presented, the improved Bellman-Ford algorithm (Wikipedia (2017)) relaxes this process by proceeding with all nodes in the graph until no active node ramains. We improve the performance of the Ballman-Ford algorithm for solving our uniform rectangular graph by keeping track of active work groups—which is larger but cover all active nodes—then proceed as before until no work group is active.

To determine active work groups, the update map is employed to keep marks that indicate which work group containing at least one updated node; in which an updated node means that its traveltime is updated by the last OpenGL's dispatch or CUDA's kernal call. Then, a work group is labaled as active, if any of nine (nearby and itself) work groups contains an updated node. We refuse to chase after individual active node, because GPU memory is valuable and, in parallel computing, invoking an invocation will alter all invocations in the same work group nevertheless. One weakness of an actual implementation of the update map is that two additional synchronizations—OpenGL's barrier() function or CUDA's __syncthreads() function—are needed. Moreover, dummy work groups which always inactive are needed at all four borders as shown in Figure 5.

Parallel version: Bellman-Ford algorithm

The only differences between our parallel version and the general Bellman-Ford algorithm are work group, update map and parallel execution. The pseudo code of our parallel implementation is shown in Algorithm 1. It can be divided into two parts: initialization (lines 1–6) and main process (lines 7–20). Only the main process is executed in parallel. Adding and removing dummy objects are trivial and, thus, excluded.

According to Algorithm 1, lines 1–5 are the same as universal implementation of the Bellman-Ford algorithm, while in line 6, only the work group, which the source node is resided, is labeled updated.

Next, the main process consists of four stages: (1) checking whether the work group is active (line 7), (2) upload local traveltime and slowness to shared memory (line 8), (3) updating candidate of traveltime of all nodes in active work groups (lines 9–18) and (4) Marking on the update map. Three synchronizations of invocations in the same work group are required within the first stage and between the other stages. The synchronization after the second stage is inevitable even without the update map.

Algorithm 1 Bellman-Ford algorithm with Update map

```
Variables:
    $Tt[X]$ = Candidate of traveltime of node X
    $pN[X]$ = Relative position of previous node to X
    $Umap[G]$ = Is any updated node in work group G?
    $isUpdated$ = Is any node in the work group updated?
Function:
    Weight($S$,$X$,$Y$) = Weight of the edge $X$–$Y$ in field $S$
    UpShmem = Upload local $Tt$ and $S$ to shared memory
 1: for each node $X$ do
 2:     $Tt[X]$ = Infinity
 3:     $pN[X]$ = NULL
 4: end for
 5: $Tt[Source] = 0$
 6: $Umap[Source \in G]$ = True (Otherwise, False)
 7: for work group $G$ which $Umap[G]$ == True do
 8:     UpShmem($G$)
 9:     for each node $X \in G$ do
10:         for each neighbor $Y$ of node $X$ do
11:             $tempTt = Tt[Y]$ + Weight($S, X, Y$)
12:             if $tempTt < Tt[X]$ then
13:                 $Tt[X] = tempTt$
14:                 $pN[X] = Y$
15:                 $isUpdated$ = True
16:             end if
17:         end for
18:     end for
19:     $Umap[G] = isUpdated$
20: end for
```

For the first stage, the nine nearby marks from the update map are uploaded to shared memory, then synchronize so that all the invocations in the same work group can see the nine marks and then recognize whether its work group is active or inactive.

The second stage is straight forward, each invocation in an active work group will upload four traveltimes and four slowness values to its shared memory as already shown in Figure 5.

For the third stage, only line 9 and line 15 differ from the general Bellman-Ford's pseudo code. For line 9, instead of checking all nodes in the graph, only nodes in active work groups are reviewed. For line 15, if any candidate of traveltime in an active work group is changed, a variable *isShortest* in the shared memory will become True. If no candidate is altered, it remains False.

The four stage is to mark the update map, whether any node in the work group is updated or not.

Two remarks are need to be provided. First, the runtime of all implementations of the Bellman-Ford algorithm depends on the number of edges of the longest shortest path in the graph. This will explains why our parallel program is slowed down when edge reduction is applied. (Now, we have no result, this is my hypothesis.)

Second, the $pN[X]$ appeared in Algorithm 1 is an interger. Although, in line 13, the character Y indicates the relative position (i, j) of neighbor in two dimensions to the node X, we can let $pN[X] = (2R_\text{x}+1)(j+R_\text{y})+(i+R_\text{x})$ which we can recover $i = \text{mod}(pN, 2R_\text{x}+1) - R_\text{x}$ and $j = ((pN - i - R_\text{x})/(2R_\text{x}+1)) - R_\text{y}$.

Serial version: Dijkstra's algorithm

To investigate the improvement of our parallel implementation, we code another serial version. The Dijkstra's algorithm using STL priority queue, which is a derivative of Agrawal (2016), is implemented as shown in Algorithm 2.

Since the STL priority queue cannot directly substitute the priority queue appeared in the Dijkstra's pseudo code, as the STL priority queue does not have decrease_key feature. One possible solution is to push more than one copy pair of a node and its (different) candidate of traveltime into the STL priority queue. Because the newer pair of a node always holds less prospective traveltime than the older pair of the same node due to line 19, the newest pair of any active node, therefore, will be popped out first as desired. Thus, what remains is to discard all the older pairs popped out later. This can be achieved by introducing an array, $flag[X]$.

Similar to the update map, the array keep one flag per node. The flag of a node is set, when its first pair is popped out from the STL priority queue. After its flag is set, all operations associate with the node will be ignored. In other words, only the node that its flag is not set will be proceed, as can be seen in lines 13–15 and line 17. In addition, (line 4–8) certainly dummy nodes must always be ignored. The rest of Algorithm 2 is the same as the general Dijkstra's pseudo code.

Three remarks are needed to be provided. First, the WeightP function in Algorithm 2 mean that, different from the Weight function in Algorithm 1, all line segments and their corresponding neighbors of each edge pointing to a neighbor (Figure 2a) are precalculated. The WeightP function only assembles all the required components and produce the weight. This modification to the weight algorithm in Appendix A is trival.

Second, the runtime of the Dijkstra's algorithm depends on the sorting algorithm for line 13. According to Agrawal (2016), the time complexity of our implement using STL priority queue is $\mathcal{O}(E\log V)$ where E is the total number of edges and V is the number of nodes in the graph.

Third, in our actual code, the array $flag[X]$ is nested in the integer array $pN[X]$, because state of a $flag[X]$ can be represented by sign of an integer. We, therefore, define $pN[X] = flag[X] \times [(2R_\text{x}+1)(j+R_\text{y})+(i+R_\text{x})+1]$ which we can derived back $flag[X] = \text{sign}(pN[X])$, $i = \text{mod}(|pN[X]|, 2R_\text{x}+1) - R_\text{x} - 1$, and $j = ((|pN| - i - R_\text{x} - 1)/(2R_\text{x}+1)) - R_\text{y}$. Unlike parallel version, the pN is additionally shifted by +1 so that no $pN[X] = 0$.

Algorithm 2 Dijkstra's algorithm with flag

```
Variables:
    $Tt[X]$ = Candidate of traveltime of node X
    $pN[X]$ = Relative position of previous node to X
    $flag[X]$ = Has any pair of node X popped out?
Function:
    WeightP($S$,$X$,$Y$) = Weight of edge $X$–$Y$ in field $S$
Data structure:
    $MinQ$ = Min ($Tt$) priority queue
 1: for each node $X$ do
 2:   $Tt[X]$ = Infinity
 3:   $pN[X]$ = NULL
 4:   if $X$ == Dummy node then
 5:     $flag[X]$ = True
 6:   else
 7:     $flag[X]$ = False
 8:   end if
 9: end for
10: $Tt[Source] = 0$
11: PUSH $Source$ in $MinQ$
12: while $MinQ$ is not empty do
13:   $X$ = Node extracted from $MinQ$
14:   if $flag[X]$ == False then
15:     $flag[X]$ = True
16:     for each neighbor $Y$ of node $X$ do
17:       if $flag[Y]$ == False then
18:         $tempTt = Tt[X] + \text{WeightP}(S, X, Y)$
19:         if $tempTt < Tt[Y]$ then
20:           $Tt[Y] = tempTt$
21:           $pN[Y] = X$
22:           PUSH $Y$ in $MinQ$
23:         end if
24:       end if
25:     end for
26:   end if
27: end while
```

RESULTS AND DISCUSSION

No result is avaliable. What will be investigated are described below.

Edge reduction & Update map

(Prove that they actually improve our program.)

Resolution and Radius

(Testing with R-gradient slowness model, we try different combination of $R_{\mathrm{x}}, R_{\mathrm{y}}$ and the number of cells or nodes, then predict the most effective radiuses (Accuracy per runtime).)

Implementations

(Apply three different programs—Serial, Parallel CUDA, Parallel OpenGL—to the same slowness model, number of nodes and radiuses, we find the best implementation.)

Further applications

(Other applications in Physics will be shortly discussed.)

CONCLUSION

Nothing can be concluded right now.

APPENDIX A: WEIGHT CALCULATION ALGORITHM

Some related facts, which are easy to prove, are stated below.

1. The crossing points can be specified by $Neighbor \cdot l$ where $l \in [0, 1]$ is length ratio and $Neighbor$ is an integer vector referring to the relative position. In Figure 3, $Neighbor = 3\,\hat{\imath} + 1\,\hat{\jmath}$.

2. In two dimensions, there are two types of crossing point: x=constant intersection and y=constant intersection. Although, some crossing points can be in both types (Figure 3), they do not affect our algorithm.

3. For each type in (2), the number of crossing points $nCross$ is equal to the absolute value of the respective component of the vector $Neighbor$. In Figure 3, there are 3 x=const. intersections and 1 y=const. intersections.

4. For each type in (2), the distance between any two adjacent crossing points is equal to the length of the vector $Neighbor$ divided by the number of cossing points $nCross$ discussed in (3). In Figure 3, the distance between two adjacent x=const. intersections is $(\sqrt{3^2+1^2})/3$. Thus, the difference of their length ratio dl is merely $1/nCross$ or $1/3$.

5. For each type in (2), the distance of the nearest crossing point to the neighbor node is half of the distance discussed in (4). Hence, the largest length ratio is $1 - 0.5\,dl$.

Obviously, from the above facts, length ratio l of all crossing point are easily obtained. What remains is to arrange them from longest to shortest, in order to find the line segments and their corresponding cell's index.

Algorithm A-1 Weight(S, X, Y) in GLSL syntax

```
Input:
    ivec2 $Node$ = Node $X$
    ivec2 $Neighbor$ = Neighbor node $Y$ relative to $X$
    $S[i][j]$ = Slowness value of cell $(i, j)$ relative to $X$
    const vec2 $Scale$ = Cell size
Output:
    $w$ = Weight of edge $X$–$Y$ in field $S$
Note:
    vec2 A in GLSL means float A[2] = {A.x, A.y} in C.
    abs($x$) will return absolute value of $x$.
    round($x$) will return nearest interger of $x$.
 1: ivec2 $nCross$ = abs($Neighbor$)
 2: const vec2 $dl = 1.0/nCross$
 3: float $w = 0$
 4: float $pl = 1.0$
 5: vec2 $l = 1.0 - 0.5\,dl$
 6: while $0 \neq nCross.x$ or $0 \neq nCross.y$ do
 7:     int $i = (l.x < l.y)\,?\,1 : 0$
 8:     vec2 $middle = 0.5\,(pl + l[i]) \cdot Neighbor$
 9:     ivec2 $index = Node + \mathrm{round}(middle)$
10:     $w = w + S[index.x][index.y] \times (pl - l[i])$
11:     $pl = l[i]$
12:     $l[i] = l[i] - dl[i]$
13:     $nCross[i] = nCross[i] - 1$
14: end while
15: $w\ = w + S[Node.x][Node.y] \times (pl)$
16: $w\ = w * \mathrm{lenght}(Neighbor \cdot Scale)$
```

In Algorithm A–1, the number of crossing points and the difference of two adjacent length ratio are computed (lines 1–2). The output weight is initialized (line 3). In line 4, the variable pl denoting previous length ratio is initially 1. The initial largest unvisited length ratio l of each type is calculated in line 5.

Next, the main loop in line 6 will iterate until no crossing points needed to be arranged. In line 7, the type i of the largest length ratio of all remaining crossing points is found. The cell's index is derived by rounding the middle point between the previous and the current crossing point, then adding with the node position in the graph (lines 8–9). Using the index, the slowness of the cell is obtained, then multiplied by the associate difference of length ratio, and added to the output weight. Whereas the actual line segment is obtained, when the weight is multiplied by the length of the vector $Neighbor$ in the actual scale at the last line.

To prepare for the next calculation, the current length ratio will be previous (line 11), the largest length ratio of the type will be shifted down (line 12), and the number of crossing point of the type will be reduced by 1 (line 13). The computation in line 15 accounts for the last line segment in the node's cell itself. In the last line, the length of $Neighbor$ in the actual scale is multiplied to the weight as already mentioned.

Finally, even though some crossing points can be counted as both types, when they meet themselves, $pl - l[i]$ in line 10 will becomes zero, thus contribute nothing to the weight. In addition, the pseudo code above is easy to be extended to three dimensions.

REFERENCES

Agrawal, S., 2016, Dijkstras Shortest Path Algorithm using priority queue of STL: GeeksforGeeks.org. (Accessed: 2018-01-10).

Giroux, B. and B. Larouche, 2013, Task-parallel implementation of 3D shortest path raytracing for geophysical applications: Computers & Geosciences, **54**, 130–141.

Harish, P. and P. Narayanan, 2007, Accelerating large graph algorithms on the GPU using CUDA, 197–208. Lecture Notes in Computer Science. Springer.

Mak, S. and K. Koketsu, 2011, Shortest path ray tracing in cell model with a second-level forward star: Geophys, J. Int., **186**, 1279–1284.

Monsegny, J. and W. Agudelo, 2013, Shortest path ray tracing on parallel GPU devices, 3470–3474. SEG.

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

Rawlinson, N. and et al., 2007, Seismic ray tracing and wavefront tracking in laterally heterogeneous media: Advances in Geophysics, **49**, 203–267.

Segal, M., 2013, OpenGL 4.4 API reference card. Khronos Group.

Sethian, J. and A. H. Popovici, 1999, 3-D traveltime computation using the fast marching method: Geophysics, **64**, 516–523.

Shearer, P., 2009, Introduction to seismology, second edition: Cambridge University Press.

Vidale, J., 1988, Finite-difference calculation of travel times: Bulletin of the Seismological Society of America, **78**, 2062–2076.

Wikipedia, 2017, Bellman-Ford algorithm—Wikipedia, The Free Encyclopedia. (Accessed: 2018-01-10).

Zhao, H., 2004, A fast sweeping method for eikonal equations: Mathematics of Computation, **74**, 603–627.