CS 515 Programming Languages and Compilers I

Project 2: Parallelizing Single Source Shortest Path (SSSP)

In this project, you will parallelize single source shortest path (SSSP) algorithm on GPUs. A SSSP algorithm finds the shortest path between two vertices in a graph. It is an important problem that admit many potential algorithmic choices. SSSP can find its uses in many applications including route planning, AI, 3D modeling, and social networks [2]. The parallelism in SSSP can be exploited using the technique for parallelizing directed acyclic graph (DAG) as we discussed in class.

You need to submit both code and report. The code comprises 60% and the report comprises 40% of project 2 grade. We will provide real world graphs as input sets. You need to evaluate your implementations on each of the graph and report the performance results for different stages of computing.

You will receive **0 credit** if we cannot compile/run your code on ilab machines or if your code fails correctness tests. All grading will be done on ***ilab*** **machines**.

The interface of your implemented code needs to meet the requirements defined in Section 3.2. We have provided sample code package for you to start with. The following three sections describe the three different versions of SSSP.

1 Bellman-ford Algorithm

The input file is a plain text file that contains only edges. Each line in the input file corresponds to an edge in the graph and has at least two vertex indices separated by space or tab. The first number specifies the start point of the edge of interest and the second one specifies the end point (the node that is pointed to in the edge). The third number (optional) contains the weight of the edge, if not specified, the edge weight is set to 1 by default.

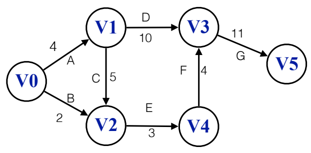
An example graph and edge list is provided in Figure 1 (a) and (b) respectively. The source node by default is always the node with the index 0 in the graph. Note that the edge list in the input file is not necessarily sorted by starting node index or destination node index. You might need to sort the list of edges as a preprocessing step. A potential data structure you can use (after sorting the edge list by starting node index) is presented in Figure 1 (b), in which the outgoing edges from a node v is represented as a set of edges from index A[v] to A[v+1]-1 in the edge list E. You can also sort the edges by the destination node index, and represent contiguous segments of incoming edges each of which corresponding to edges pointing to the same node. This is similar to the compressed sparse row (CSR) format for sparse matrices.

The sequential bellman-ford algorithm checks all edges at every iteration and updates a node if the current estimate of its distance from the source node can be reduced. The number of iterations is at most the same as the number of vertices if no negative cycle exists. Our input graphs do not contain negative cycles. The complexity of the sequential bellman-ford algorithm is $\mathcal{O}(\mid V \mid \cdot \mid E \mid)$. The pseudocode for sequential bellman-ford algorithm is described as below:

Listing 1: Sequential Bellman-ford

```
for each vertex v in vertices:
        distance[v] := inf;
distance[source] = 0;
for i from 1 to size(vertices)-1 {
  for each edge (u, v) with weight w in edges {
      if distance[u] + w < distance[v]:
        distance[v] := distance[u] + w;
        }
  if (no node's␣distance␣changed)␣break;
}
```

(a) Original graph



(b) Data structure

Edge list **E**:	*Outgoing edges* ***A***:	Vertex value ***V (initially)***:
0 1 **4**	A[0]: 0	V[0]: 0
0 2 **2**	A[1]: 2	V[1]: ∞
1 2 **5**	A[2]: 4	V[2]: ∞
1 3 **10**	A[3]: 5	V[3]: ∞
2 4 **3**	A[4]: 6	V[4]: ∞
3 5 **11**	A[5]: 7	V[5]: ∞
4 3 **4**		

(c) Implementation 1

	Node values						*Edges visited*
Init.	0	∞	∞	∞	∞	∞	{A, B, C, D, E, F, G }
Iteration 1	0	4	2	∞	∞	∞	{A, B, C, D, E, F, G }
Iteration 2	0	4	2	14	5	∞	{A, B, C, D, E, F, G }
Iteration 3	0	4	2	9	5	25	{A, B, C, D, E, F, G }
Iteration 4	0	4	2	9	5	20	{A, B, C, D, E, F, G }

(d) Implementation 2

Node values (checked for update)						*Edges visited*
0						{ }
	4	2				{ A, B }
	4	2	14	5		{ C, D, E }
			9		25	{ F, G }
					20	{ G }

Figure 1: SSSP Example

The sequential bellman-ford algorithm does not have as good asymptotic complexity as the Dijkstra's algorithm. However, it is amenable to parallelization. The first two required versions of parallel implementations are both based on the bellman-ford algorithm. An example for the bellman-ford algorithm is presented in Figure 1 (c). In this example, it takes four iterations before the distance value of every node in the graph stops changing. The node values at the beginning of every iteration is provided in Figure 1.

In the parallel bellman-ford algorithm, we exploit the parallelism of edge processing at every iteration. For example, in Figure 1 (c), each of the seven edges is checked at every iteration, we can distribute these 7 edges evenly to different processors such that each processor is responsible for the same number of edges.

The pseudo code of the first parallel implementation is given below:

Listing 2: Parallel SSSP Implementation 1

```
1  kernel edge_process(L, distance_prev, distance_cur)
2  {
3    load = L.length % warp_num == 0 ? L.length/warp_num: L.length/warp_num+1;
4    beg = load*warp_id;
5    end = min(L.length, beg + load);
6    beg = beg + lane_id;
7    for ( i = beg, i < end,  i += 32 ) {
8      u = L[i].src;
9      v = L[i].dest;
10     w = L[i].weight;
11      if ( distance_prev[u] + w < distance_prev[v]
12        atomicMin(&distance_cur[v], distance_prev[u] + w);
13
14   }
15 }
16 ...
17
18 for (i from 1 to size(vertices)-1 ) {
19   edge_process(L, distance_prev, distance_cur);
20   if ( no node is changed) break;
21   else swap(distance_cur, distance_prev);
22 }
```

The prev node list array distance_prev and current node list array distance_cur in Listing 2 represent respectively the result from the last iteration and the updated result at the end of the current iteration. Every iteration here corresponds to one kernel invocation (Line 19 at Listing 2). This is an out-of-core implementation, which means once these nodes are updated, their distance is not immediately visible to other co-running threads in the same kernel invocation. We switch the pointers to these two arrays in Line 21 of Listing 2 before calling the kernel in the next iteration.

What to Report?

- Execution configuration: you need to report results for the five following block size and block number configurations: (256, 8), (384, 5), (512, 4), (768, 2), (1024, 2). The edge list will be evenly divided to different thread warps.
- Edge list organization: You need to report the running time for the three cases: when the edge list is sorted by source node index, and when the edge list is sorted by destination node.
- In-core configuration: you need to implement the in-core method and report the running time for all graphs. Note that in Listing 2, we used two arrays for the node list, one represents the state before the iteration and the other represents the state after the iteration. You will need to use one single array "distance", and remove distance_prev & distance_cur such that the original two lines:

```
    if ( distance_prev[u] + w < distance_prev[v] )
       atomicMin(&distance_cur[v], distance_prev[u] + w);
```

should look like this:

```
    temp_dist = distance[u] + w;
    if ( temp_dist < distance[v] )
        atomicMin(&distance[v], temp_dist);
```

You need to make other changes correspondingly, for example, the parameters to the kernel function need to be updated, the code to swap the two pointers need to be removed. The code to check if any node is updated needs to updated.

You also need to report the running time of every graph when the edge order is when the edge list is sorted by starting node, and when the edge list is sorted by destination node.

- Using shared memory: you will need to implement a segment-scan type method and report the running time. You will only need to run the segscan method on the edge list sorted by destination index, in which the incoming edges for the same node can be processed by consecutive threads (or iterations). You will only need to use the out-of-core processing mechanism. Since finding the minimum of a segment of values is similar to computing the sum of a segment of values, you can use the similar segment scan code in project 1 to find the minimum distance for every segment before you atomically update using atomicMin.

The CUDA atomic intrinsic functions are provided here:
http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

2 Work Efficient Algorithm

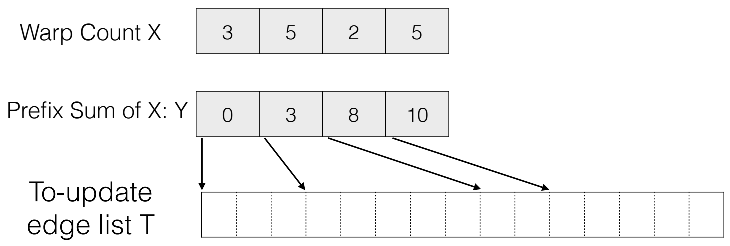


Figure 2: Use prefix sum to calculate the index of an edge in the work list

We can improve *Implementation 1* by reducing the edges that need to be processed at each iteration. For any directed edge, if the starting point of the edge did not change, it will not affect the point-to end of the edge (the destination node on the other end). Thus, we can skip this edge when performing the comparison and updating "if (distance[u] + w < distance[v]) ..." computation.

In this implementation, you will need to filter out the edges whose starting point did not change in the last iteration and only keep the edges that might lead to a node distance update. We refer to the edges that need to be checked as **to-process** edges. To achieve this, you can use a parallel scan operation.

Recall that the exclusive scan is an operation that scans a sequence numbers of x_0, x_1, x_2, ..., x_n, and produces a second sequence numbers of y_0, y_1, ... y_n such that: $y_0 = 0$, $y_1 = x_0$, $y_2 = x_0 + x_1$, and so on. Similar to reduction, the scan operation can be performed with logarithmic parallel steps [1]. Assuming x_i stores the number of to-process edges identified by warp i and T stores all the to-process edges, y_i gives the total number of to-process edges gathered by warps from $warp_0, warp_1, .., warp_{i-1}$, and thus gives the index of the first to-process edge found by warp i in the list T. That is, the first to-process edge detected by warp i will be stored at $T[y_i]$. An example is given in Figure 2.

You may implement the scan operation for gathering to-process edges in three steps. In this project, we assume that the total number of threads will not be greater than 2048. We also assume that one thread warp will gather to-process edges from a contiguous range of edges from the edge list L, such that the warp i will process the range from index (L.length/warp_num) * i to index L.length/warp_num * (i+1) - 1 (inclusive).

1. Every warp counts the number of edges that need to be processed in its assigned edge range. x[warpid] specifies the number of to-process edges that are identified by warp warpid.

   Specifically you can use warp voting functions for fast identification of the to-process edges for threads in the same warp. The "unsigned int __ballot(int predicate)" function in CUDA returns a value with the N-th bit set, where N is the lane index in the warp if the predicate is non-zero. If you combine the __ballot function with __popc function, you can count the number of threads in a warp which have the predicate set to non-zero. In our example, you can use "mask = __ballot(current_edge.src == changed); count = __popc(mask);" to get count, which is the number of lanes in the warp that has seen the predicate (current_edge.src == changed) set to true.

   More information about the ballot function can be found here:
   https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/

2. Since we assume the total number of threads is $\leq$ 2048, the number of warps is $\leq$ 64. In this step, we launch a kernel that has the same number of threads as the number of warps in the last step (and the block number is 1). At this step, we perform a block level parallel scan to get the offset of every warp's first to-process edge in the final to-process edge list T.

3. We let every warp copy its identified to-process edge index to the array T. If a warp's offset is y[i], and it has detected k to-process edges (not necessarily contiguous), then it will copy these k edges to $T[y[i]]$:$T[y[i] + k - 1]$.

   Specifically, you can also use the ballot and the integer intrinsic functions to help you determine the index of the to-process edge in the final list T.

   After "mask = __ballot(current_edge.src == changed); localid = __popc(mask<<(32-lane)); T[localid + cur_offset] = current_edge.id;", you can successfully copy the to-process edge to the right location. In this code snippet, cur_offset represents the offset for this current warp and current loop iteration since one warp might process many 32-edge segments in a loop. Before the first iteration of this loop starts, the cur_offset should be y[i]. The cur_offset should be updated as more and more edges are processed in following iterations, it can be incremented by "__popc(mask)" at the end of every iteration.

More details about the integer intrinsic functions can be found here:
http://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__INT.html

Note that we will focus on improving performance of the computation stage not the performance of the filtering stage. Here we have describe one approach for the filtering stage. This approach needs to scan all edges in the edge list before filtering is done. It may not be the most efficient way. You can implement the filtering stage in your own way as long as the relative ordering of the to-process edges is the same as they are in the complete list L. Another possible approach is to record which nodes have been updated during the last computation stage, get the outgoing edges from each of these nodes (use the sort-by-source edge list data structure) represented a contiguous range of edges in the edge list [v.beg, v.end), gather these ranges $[v_i.beg, v_i.end), i = 0...k$ (assuming k nodes have been updated), concatenate these edges into a single list of to-process edges. You can also use prefix sum to get the total number of edges to process and then let individual thread (warp) process a subset of them. Finally, you can also implement the filtering stage on CPU.

After filtering out the edges that do not need to be considered, you will evenly distribute the to-process edges among different thread warps and perform the same iterative process as described in *Implementation 1*.

What to Report?

- Use the same execution configuration as specified in Implementation 1.
- You need to report the running time for the computation stage and the filtering stage separately.
- You need to report the results for the edge list sorted by starting point for the out-of-core and in-core processing kernel.
- You need to perform data analysis, summarize what you have observed and discuss (reason about) the differences in various configurations.

3 Optimization (Optional – 10% Extra-Credit)

In the third implementation, you will design and implement your optimization strategy. You can use a priority queue to improve the convergence speed as in the sequential Dijkstra's method. You can also optimize the filtering process (vertex-edge process) on GPUs in the work efficient version (currently in Section 2 only part of the work-efficient version is required to be implemented on GPU). The vertex-to-edge expansion process in Figure 2 can be efficiently implemented using parallel prefix sum. You can also use any other strategy you design by yourself.

You will need to specify clearly which optimization you have applied. Your implementation does not necessarily need to always run faster than the first two implementations. However, you will need to describe why you think your proposed optimization might help, and provide some intuition, i.e., use a simple example. This optional optimization component is due before the end of this semester.

Here we provide two potential optimizations on locality and priority queue that you can implement.

3.1 Potential Optimization 1: Locality Enhancement

You can use graph partition method to divide one kernel invocation into multiple kernel invocations for last level cache performance optimization. The GPU we have on ilab cluster each has 256KB last level cache. Not the entire data set can fit into cache at one time. Therefore, we can divide one kernel execution (if there is sufficient workload) into multiple kernel execution such that each kernel's (frequently-used) data set size fit into cache.

To partition the workload, you can partition the edges of the graph. We will provide a balanced graph edge partition library based upon request. Note that the provided library only performs balanced partition, if you want to keep partition the tasks into variable size components, you might need to perform hierarchical partitioning, for instance, use bisection at one time, and only split the individual sub-partition that you want to split.

Note this optimization may only apply to implementation 1, since only in implementation 1, the edges to be processed in every original kernel invocation are the same.

3.2 Potential Optimization 2: Priority Queue

In our first two implementations in this project, we treat every edge as having the same priority. The Dijkstra's algorithm and the near-far tile method [2] let edges that potentially point to smaller distance nodes have higher priority. The basic idea is to maintain a priority queue. At one time, we take the first n tasks from the priority queue, process them, which further generates new tasks that can be inserted into the priority queue (which might require some reordering operations in the queue or deletion of tasks in the queue).

If choosing the priority queue based approach, you will need to try at least one heuristic for the priority function. The near-far tile method uses the distance as the heuristic, that is, the edges in the work list that come from the nodes which are bound by a distance have higher priority.

Another potential heuristic you can try is to set the nodes that have the least number of un-updated predecessors nodes as having high priority. Specifically, if a node's all predecessors nodes have been updated in previous iteration(s), this node will have higher priority and the incoming edges to these nodes will be placed at the beginning of the priority queue. Next we place the edges that point to these nodes that just have only one un-updated (immediate) predecessor nodes in the queue – here an immediate predecessor node points to the node of interest. And next the edges that point to the nodes with two un-updated (immediate) predecessor nodes. Note that when ordering the edges in the list based on the priority order, the list should be the actual work-efficient work list, that is, we don't consider any edge whose starting point has not been changed.

In this exploration, you can maintain the priority queue (insertion and deletion) in GPU or CPU. As mentioned before, our focus at this project is not how to reduce the cost of parallel priority queue implementation. Rather we would like to explore heuristics that will lead to a good parallel SSSP implementation. Once you discover a good heuristics, you can try to improve the parallelization of priority queue on GPUs.

4 Compilation

We provide a sample code package for you to start with. It is here:
/ilab/users/zz124/cs515_2017/projects/proj2/code_sample.

Compilation: A Makefile is included in the sample code package. If you want to add new files, please make sure your Makefile is updated. Please do not change the final executable name – sssp. The grader will compile your code with the following commands:

- `make sssp` should compile everything into a single executable called **sssp**. The usage will be described in execution interface.
- `make clean` should remove all generated executable and intermediate files.

Run Your Code: Your code will need to take different number of command line options defined as follows:

Usage: Required command line arguments (in order):
Input file: E.g., –input in.txt
Block size: E.g., –bsize 512
Block count: E.g., –bcount 4
Output path: E.g., –output output.txt
Processing method: bmf (bellman-ford), tpe (to-process-edge), opt (one or two optimizations)
Sync method: –sync incore, or outcore

The sample code package provided to you have some code for parsing and evaluating the command options. Feel free to reuse them or modify them.

Program Output: The program output at standard out needs to report the kernel execution time in the following format.

"The total computation kernel time on GPU [gpuDeviceName] is xxxx milli-seconds", and
"The total filtering kernel time on GPU [gpuDeviceName] is xxxx milli-seconds".

You can use the same sample timing function as provided in project 1. Feel free to use any other type of timing function if you think is more accurate.

The program also needs to output a vector file called "output.txt" which lists the shortest distance to every node from the first node to last node in the format of "<vertex id>: <distance>". By default, node 0 is the source node, and thus the first value in the "output.txt" should always be "0:0". We will use this vector to test the correctness of your implementation. A reference binary for shortest path implementation will be provided later in piazza.

Machine to Use: Please use the same set of GPUs as in Project 1. The list of machines is provided here:

http://report.cs.rutgers.edu/mrtg/systems/ilab.html

We will compile and test your code on these ilab machines only.

Performance Debugging: You can use *nvprof* to check various performance metrics for a given CUDA program. You can run it from command line, i.e., "nvprof –metrics l2_l1_read_hit_rate a.out [paramlist]" will give you the l2 hit rate for the program a.out running on the input parameters [paramlist].

More information about nvprof is here:

docs.nvidia.com/cuda/profiler-users-guide/

5 Input Graph List

The following graph files will be used to test your sssp implementations.

- RoadNetCA: http://snap.stanford.edu/data/roadNet-CA.html
- LiveJournal: http://snap.stanford.edu/data/soc-LiveJournal1.html
- Pokec: http://snap.stanford.edu/data/soc-pokec.html
- HiggsTwitter (the social network one in the list): http://snap.stanford.edu/data/higgs-twitter.html
- WebGoogle: http://snap.stanford.edu/data/web-Google.html
- Amazon0312: http://snap.stanford.edu/data/amazon0312.html

More graphs are available from http://snap.stanford.edu/data/index.html

You can use other graphs to test your code and reason about the pros and cons of different implementations. The format of these graph files is described in Section 1. Some graph file I/O code is provided in the code package for you.

6 What to Submit?

You will need to submit a package called sssp_gpu.tgz including all the source code files, the Makefile, as well as a README for any special instructions on how to compile and run your program. You will also need to submit a report called sssp_report.pdf that describes the pros and cons of each method. The report needs to have at least six pages, not including references. Please use the latex or word template from here: https://www.usenix.org/conferences/author-resources/paper-templates . Please do not modify the template format.

Please do not include any graph files in your submission. If you used any graph not in the specified list and described it in your report, please specify the source (i.e., a web link).

References

[1] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[2] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 349–359, May 2014.