Parallelization of shortest path algorithms

Ruijian An

I. Introduction

Given a weighted graph G = (V, E) and a source vertex *s*, single source shortest path problem(SSSP) asks the shortest distance from the source vertex to all vertices in the graph[2]. SSSP is a fundamental part of graph theory and has broad applications in road networks[1], DNA microarrays[3] and many other problems which can be abstracted as graphs. Many algorithms, such as Dijkstra's algorithm and Bellman-Ford algorithm, are proposed to solve SSSP. To minimize the running time, Dijkstra's algorithm utilizes a priority queue, so it is sequential in itself. On the other hand, Bellman-Ford algorithm does much more work but contains a lot parallelism. Confronted with huge graphs like social network and road network, it is almost impossible to process vertices one by one using Dijkstra's algorithm. However, modern GPGPU provides a brand new solution to SSSP. In the report, I present my current work on parallelization of the Bellman-Ford algorithm. At last, parallel algorithms for all-pair shortest path problem would be implemented and compared.

II. Related work

In 1950s, Bellman-Ford algorithm is proposed by Bellman and Ford. In the algorithm, each vertex in the graph has an additional attribute *d* to record the tentative distance from source vertex to itself. The algorithm also employs a technique called *relax*, which is used to process edges. If we relax an edge *(u,v)* whose weight is *w(u,v)*, we update the *v.d* with *min(v.d, u.d + w(u,v))*. The Bellman-Ford algorithm relaxes each edge for $|V|$ - 1 times, so the running time is O($|V||E|$).

With the development of modern GPU, many parallel versions are proposed. The main parallelism is embedded in the step which relaxes all edges in the graph. P.Harsih proposes to first visit all vertices in parallel and then to relax outgoing edges[6]. In this way, many threads might read and write *v.d* for some vertex *v* concurrently, so atomic operation is necessary to avoid race conditions. However, atomic functions serialize the contentious updates from multiple threads, thus increasing the running time. Instead of relaxing leaving edges, G.Hajela relaxes all incoming edges after visiting all vertices in parallel[7]. For each vertex *v*, this method first computes the minimum of *u.d + w(u, v)* for all *v*'s neighbours *u*, then only this minimum would be used to update *v.d*. In this case, atomic operation is avoided because there is no concurrent read and write by different threads, but either a loop or a nested kernel is introduced. It is not obvious which method achieves better result, so more details on these approaches and a comparative discussion are included in later sections.

To optimize the performance of the parallel version, several techniques are employed to increase work efficiency. F.Busato uses *frontier propagation* and *edge classification* to largely reduce unnecessary work[10]. *Frontier propagation* takes advantage of a queue which only contains active vertices so that the unchanged vertices in last iteration would not be processed. *Edge classification* divides edges into several classes and relaxation of different class edges is optimized based on their features. A.Davidson tests the performances of two techniques – *workfront sweep, near-far* to improve work efficiency[9]. The idea of *workfront sweep* is basically the same as *frontier propagation*. Motivated by Dijkstra's algorithm and delta stepping algorithm, *Near-Far Pile* splits the vertices into *the Near Set* and *the Far Set*. The method first processes the near set and then the far set. In this way, much work to process the far set is discarded. Among all methods, *near-far* achieves the best result.

III. Parallel implementation

In my implementation, I combine the existing techniques and make some improvement. Most parallel versions above choose to map a thread to each vertex and then relax incoming/outgoing edges. However, such way causes many potential inefficiencies. First, the degree of vertices is different, which means each thread might face the imbalanced work problem. Second, parallelism hidden in the algorithm is not fully exploited. After visiting vertices in parallel, either a loop or a nested kernel is necessary to relax edges related to it[10]. Instead, I choose to map threads to edges. Usually, general or dense graphs with huge number of vertices have many more edges(which is O($|V|^2$) than vertices(which is O($|V|$)). This method solves the imbalanced work problem and introduces more parallelism. In the following parallel version, we assume the input graph is a complete graph. Another versions for general or sparse graphs will be discussed later in *Discussion* section.

First, we need to choose a proper graph representation. Traditionally, a graph is represented by either an adjacency matrix or an adjacency list. Also, compact adjacency lists are commonly used in parallel graph algorithms[6][9]. CUDA can directly map threads to 1D, 2D arrays due to the syntax of kernel call, so the adjacency matrix representation is the best choice for mapping to edges. Also, adjacency matrix is easy to generate and to test. Contrarily, it is hard to form a one-to-one correspondence between edges and threads for an adjacency list representation due to different degree of vertices. If we opt for a compact adjacency list, additional techniques like *scan, reduce* are needed, which complicate the problem.

Algorithm 1: MAIN_ BELLMAN_FORD

```
Input: dist, src_idx, matrix
flag = 0;
mask1 = $[0, \dots, 0]$;
mask = $[0, \dots, 0]$;
CUDA_INITIALIZATION(dist, src_idx, mask, mask1);
for $i \leftarrow 1$ to $|V| - 1$ do
    CUDA_RELAXATION(dist, matrix, mask, mask1, flag);
    if $flag == 0$ then
        break;
    end
end
CUDA_SWAP(mask, mask1, flag);
```

The input of the main function is *dist, src_idx, matrix*. The length of *dist* is equal to the number of vertices in the graph and *dist[i]* represents the tentative shortest distance from source vertex to vertex *i*. *Src_idx* is the index of the source vertex. *Matrix* is an adjacency matrix where *matrix*$[i][j]$ stores the weight of edge *(i, j)*.

This parallel version utilizes three additional structures *mask, mask1, flag*. First, *mask, mask1* aim to save unnecessary relaxations. *Mask* records whether a vertex is active, so the length of both arrays is equal to $|V|$. If vertex *i* is active/inactive, then *mask*$[i]$ = 1/0. In algorithm 3, we first check whether the starting vertex of an edge is active, and we only relax the edges related to active vertices, which saves much unnecessary work. A vertex *i* is marked as active once *dist*$[i]$ is updated. However, we cannot simply set *mask*$[i]$ to be 1 and set its starting vertex back to be inactive. As a result, *mask1* is needed to avoid potential conflicts. We read from *mask*, update information in *mask1* and swap (algorithm 4) *mask* with *mask1* after each iteration.

Second, *Flag* aims to break the outer loop once all vertices have found their shortest path. *Flag* is initialized to be 0 and would be set to 1 in algorithm 3 if any vertex's tentative distance is updated. After each iteration, if *flag* is still 0, then the main algorithm can terminate early.

Algorithm 2: CUDA_INITIALIZATION

```
Input: dist, src_idx, mask, mask1
idx = getThreadIDx;
dist[$idx$] = INFINITY;
mask[$idx$] = 0;
mask1[$idx$] = 0;
if $idx == src\_idx$ then
    dist[$idx$] = 0;
    mask[$idx$] = 1;
end
```

Algorithm 3: CUDA_RELEXATION

```
Input: dist, matrix, mask, mask1, flag
i = getThreadIDx;
j = getThreadIDy;
if mask[$i$] == 1 then
    temp = atomicMin(dist[$j$], [$i$] + matrix[$i$][$j$]);
    if temp > dist[$j$] then
        mask1[$j$] = 1;
        flag = 1;
    end
end
```

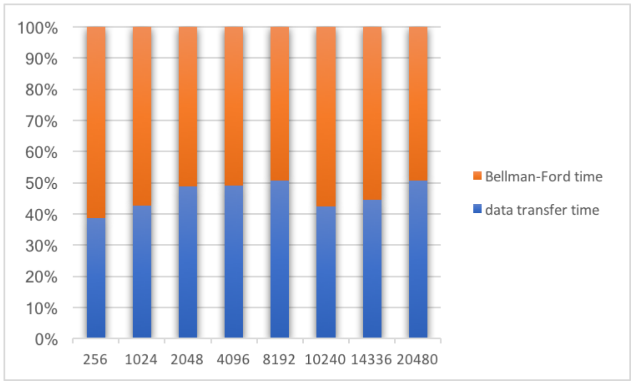
Algorithm 4: CUDA_SWAP

```
Input: mask, mask1, flag
i = getThreadIDx;
temp[$i$] = mask1[$i$];
mask[$i$] = temp[$i$];
mask1[$i$] = 0;
flag = 0;
```

IV. Discussion and analysis

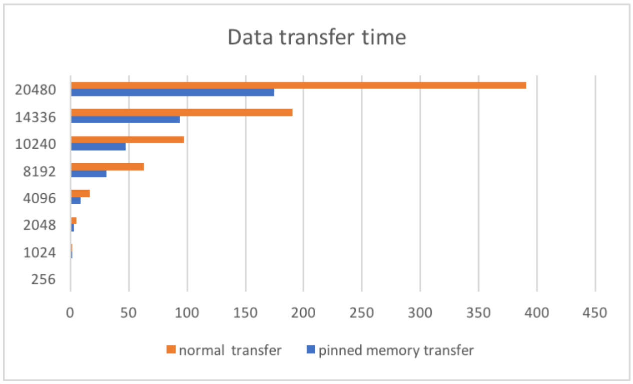
A. Improvement on data transfer

Although many efforts are made to optimize the relaxation step, data transfer also has determining influence on the overall performance. The figure on the right shows the percentage of data transfer time in the total running time, which almost reaches 50%, so improvement should be made to minimize the transfer cost.



Streams and pinned memory can be used to optimize data transfer. First, data transfer can overlap kernel execution by using streams. Without specifying the stream, all device operations run in the default stream. Also, we can create non-default streams. Streams can be specified in the device operation call as a parameter. Device operations in the same stream will be executed in the prescribed order while those in different streams can operate concurrently. To optimize the basic parallel version, two streams *stream1, stream2* are created, and the adjacency matrix is divided evenly into two parts. First part of the matrix is transferred to device and then is processed in *stream1*. Concurrently with processing the first part, the second part is transferred to GPU in *stream2*.

As a result, part of the transfer time is saved. Meanwhile, the work to process the second part is dramatically reduced since the vertices in the first part has found smaller tentative shortest distances. The more pieces the matrix is partitioned into, the more overlap we get. However, it is not worthwhile to break the matrix into many pieces since CUDA is less efficient in transferring small data and too many operations in one stream serialize the code.



Second, pinned memory is useful. By default, CPU memory allocation is pageable. Unfortunately, GPU is unable to access pageable memory directly. Whenever a data is to be transferred from the host to the device, it is first copied to pinned host memory which serves as a staging area. Then the data is transferred from pinned memory to the device. The cost of the first step can be avoided. I test the transfer time of the input matrix whose size ranges from 256 * 256 to 20480 * 20480. In the figure on the left, we can see that the pinned memory outperforms the normal transfer. As the matrix size grows, the advantage of pinned memory gets larger.

B. Version for larger graphs

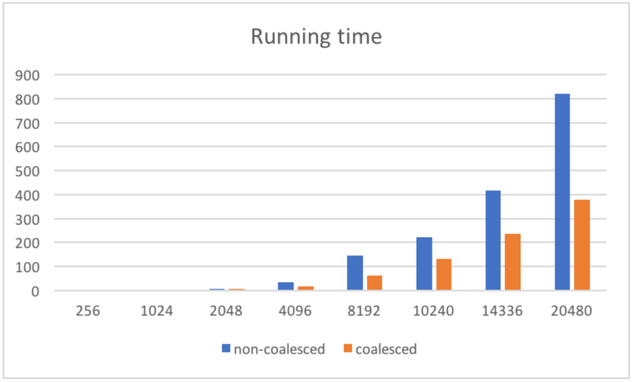
In this part, I present a variant designed for huge graphs. The basic parallel implementation above maps each thread to an edge in the graph. However, the maximum number of threads which can run concurrently is limited by the hardware. For example, GTX 750 Ti has 5 streaming multiprocessors(SM) and maximum number of threads per SM is 2048 according to the device query, which adds up to 10240 parallel threads at max. As a result, once kernel launches more threads than the limit, additional blocks must wait to be executed sequentially, causing inefficiency. On the other hand, the computation power of a thread is unclear, so we do not know the optimized number of edges a thread can process. The detail performance of different edges per thread would be presented later.

Algorithm 5: CUDA_RELEXATION_MULTI_EDGES

```
Input: dist, matrix, mask, mask1, flag
i = getThreadIDx;
j = getThreadIDy;
while $i < |V|$ do
    while $j < |V|$ do
        if $mask[i]$ == $1$ then
            temp = atomicMin(dist$[j]$, dist$[i]$ + matrix$[i][j]$);
            if $temp > dist[j]$ then
                mask1$[j]$ = 1;
                flag = 1;
            end
        end
        $j$ += $gridDim.y * blockDim.y$;
    end
    $i$ += $gridDim.x * blockDim.x$;
end
```

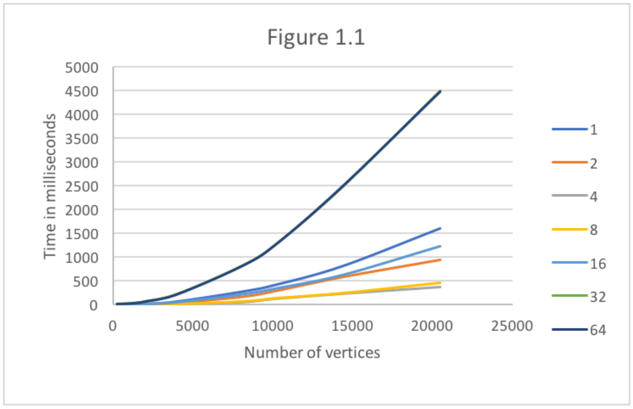
In this variant, we launch fewer threads than the number of edges in the graph. First, each thread starts on a different data index. Second, after each iteration, we increment each index by the total number of threads in the grid, which is *gridDim.y(or x) * blockDim.y(or x)*.

There are two advantages of this implementation. First, the number of edges processed by a thread is determined by the *gridDim* and *blockDim*, so it is easy to modify the number of edges per thread and to test the performance. Second, global memory access is coalesced in this way. In a nutshell, memory coalescing means the consecutive threads access consecutive memory address. In the variant above, all the threads *i* access the first *gridDim.x * blockDim.x* elements in parallel and move on to the next *gridDim.x * blockDim.x* elements after one iteration.



In the figure above, I compare the performance between coalesced memory access and non-coalesced access. Usually, data transfer between CPU and GPU costs much time as said in prveious section. Similarly, it is expensive to access global memory, so a proper access pattern would lead to a considerable speedup.

In the following part, I test the performance of different versions where each thread processes 1, 2, 4, 8, 16, 32, 64 edges.



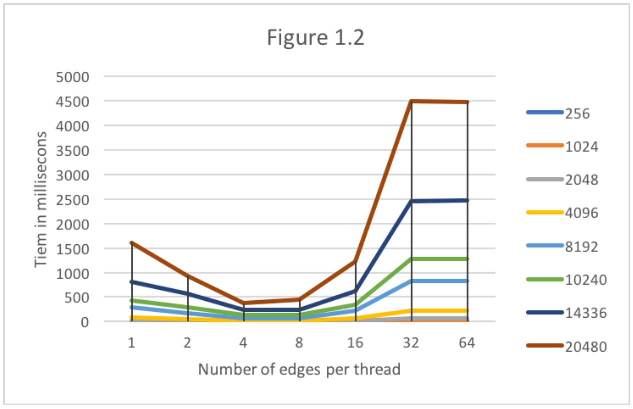


Figure 1.1 shows the running time increases with the number of vertices. Also, different versions' performances vary dramatically. Figure 1.2 shows the trend in a straightforward way. Fixed the size of the input matrix, the running time increases after a initial decrease. First, we can guess that a thread is capable of relaxing more than one edge efficiently. Second, the number of blocks has an influence on the performance when it exceeds the number of SMs. When the number of edges per thread increases from 1 to 4, the number of blocks largely decreases and hence blocks needn't wait for SM to process them in order. Also, relaxing 4 edges is within one thread's computation power, so the overall running time decreases. However, when the number of edges increases from 4 to 64, each thread relaxes huge number of edges serially, which dominates the running time. Although the number of blocks decreases, a thread becomes less efficient when relaxing huge number of edges.

C. Version for sparse/general graphs

In section III, adjacency matrix representation is chosen to model complete graphs since it is well suited for parallel architecture. However, general graphs is far more common in many applications, so we need a modification for them.

Algorithm 6: CUDA_RELEXATION_GENERAL_GRAPH

```
  Input: dist, s, e, w, mask, mask1, flag
1 i = getThreadIDx;
2 if $mask[s[i]] == 1$ then
3     temp = atomicMin(dist$[e[i]]$, dist$[s[i]]$ + w$[i][j]$);
4     if $temp > dist[e[i]]$ then
5         mask1$[e[i]]$ = 1;
6         flag = 1;
7     end
8 end
```

To preserve the advantage of processing edges in parallel, I choose to use three arrays: *s, e, w* to store edges. *s* stores the starting vertices of edges while *e* stores the end vertices of edges, so (*s[i], e[i]*) is an edge whose weight is *w[i]* in the graph. In the variant, 1D block and 1D grid are launched to process three arrays, but there is a limit for it. For example, GTX 750 Ti can launch 2147483647(blocks per grid) * 1024(threads per block) threads in total. Under such circumstances, each thread need to be responsible for several edges, just like the multiple edges version in section IV B. It is similar to a common approach in some papers, where each thread processes the adjacency list of a vertex, but variable degree of vertices causes imbalanced work problem. A.Davidson[9] proposes *Cooperative Blocks*, which utilizes the shared memory so that threads in a block can process a set of n vertices in parallel. However, this is not a problem for variant proposed since we can always divide the edges evenly to each thread.

D. Different ways to relax edges

In the initial Bellman-Ford algorithm, relaxation of all edges in the graph is the most expensive step, so the parallelization of this step determines overall performance. However, there are many ways to parallelize it.

First, we have two choices: mapping threads to edges or mapping threads to vertices. In a general or dense graph, the number of edges is probably larger than that of vertices. As a result, mapping threads to edges introduces more parallelism and eliminates imbalanced work problem. However, the number of streaming multiprocessors and maximum number of threads on a SM is limited, which directly limits the size of the graph as said in section IV B. To solve this problem, it is necessary to process a chunk of edges per thread. On the other hand, mapping thread to vertex can easily handle a much larger graph.

Second, an edge *(u,v)* can be considered as either an incoming edge or an outgoing edge. If we map a thread to an edge, this problem makes little difference. Nevertheless, if a thread is mapped to a vertex, two ways has big impact on the implementation. I present the pseudocode of two implementations and compare the performance on several graphs.

Algorithm 7: CUDA_RELEXATION_INCOMING

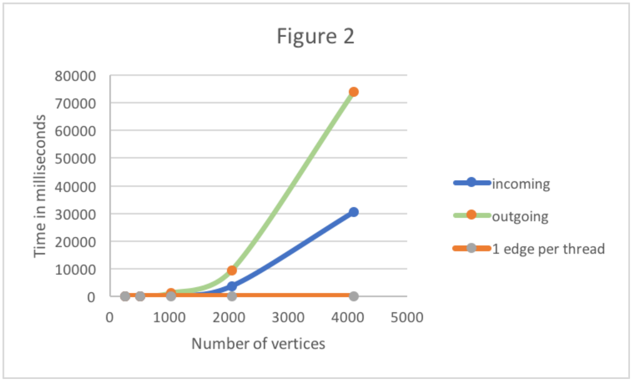
```
Input: dist, matrix, mask, mask1, flag
idx = getThreadIDx;
minimum = dist[$idx$];
for $i \leftarrow 0$ to $|V| - 1$ do
    if mask[$i$] == 1 then
        if dist[$i$] + matrix[$i$][$idx$] $<$ minimum then
            minimum = dist[$i$] + matrix[$i$][$idx$];
            mask1[$i$] = 1;
            flag = 1;
        end
    end
    dist[$idx$] = minimum;
end
```

Algorithm 8: CUDA_RELEXATION_OUTGOING

```
Input: dist, matrix, mask, mask1, flag
idx = getThreadIDx;
if mask[$idx$] == 1 then
    for $i \leftarrow 0$ to $|V| - 1$ do
        temp = atomicMin(dist[$i$], dist[$idx$] +
          matrix[$idx$][$i$]);
        if temp $>$ dist[$i$] then
            mask1[$i$] = 1;
            flag = 1;
        end
    end
end
```

Both versions use a loop to traverse its neighbours. However, the incoming version uses the loop to find the minimum and update *dist[idx]* for all vertices in parallel while the outgoing version uses an atomic operation to update the *dist* for its neighbours in parallel. It is not obvious which method is more efficient, so I test the performance on different graphs.

In figure 2, I present the running time curve for incoming version, outgoing version and the basic parallel version. As expected, the basic parallel version is far more efficient than the version mapping threads to vertices. Both incoming version and outgoing version employ a loop, but there is a huge gap between the performance of two versions. I think the gap is due to the atomic operations. In the incoming version, the minimum is computed inside the kernel and *dist[i]* is updated only once. On the other hand, the outgoing version accesses the *dist[i]* atomically for every iteration and at the same time other threads have to wait for next execution. Also, every time a smaller value is found, the *dist[i]* would be updated, so it is less efficient.



V. All pairs shortest path problem

A. Problem statement and proposed algorithms

Similar to single-source shortest path problem, all-pairs shortest path problem asks for the shortest path distance between any two vertices in the graph. Many methods are proposed to solve this problem.

First, it can be solved by running single-source shortest path algorithms for $|V|$ times[2]. So, the Dijkstra's algorithm and the Bellman-Ford algorithm would run in O($|V|^2\, lg|V|$ + $|V||E|$) and O($|V|^2|E|$) respectively.

Algorithm 9: APSP_ALGORITHM

```
Input: L
m = 1;
while $m <|V| - 1$ do
    for $i \leftarrow 1$ to $|V|$ do
        for $j \leftarrow 1$ to $|V|$ do
            for $k \leftarrow 1$ to $|V|$ do
                $L_{ij}$ = min($L_{ij}$, $L_{ik}$ + $L_{kj}$)
            end
        end
    end
    m = 2m
end
return L
```

Algorithm 10: SQUARE_MATRIX

```
Input: L
for $i \leftarrow 1$ to $|V|$ do
    for $j \leftarrow 1$ to $|V|$ do
        $L_{ij}$ = 0;
        for $k \leftarrow 1 m$to $|V|$ do
            $L_{ij}$ += $L_{ik}$ * $L_{kj}$
        end
    end
end
return L
```

Second, a dynamic-programming algorithm for all-pairs shortest paths problem is proposed by Corman[2]. The input of the algorithm is an adjacency matrix L where L_{ij} is either edge's weight w_{ij} or infinity if there is no edge between *i, j.*

By finding the *min*(L_{ij}, L_{ik} + w_{kj}) for all *i*'s neighbour *k*, a shortest path from *i* to *j* consisting of two edges would be found and we store the minimum back to L_{ij}. After repeating this process for $|V|$ times, the shortest path consisting of $|V|$ vertices is found. It is the actual shortest path since any path in a graph with $|V|$ vertices contains at most $|V|$ vertices. After substituting the minimum and addition with multiplication and addition, this process, line 2 - 5 in algorithm 9, becomes matrix multiplication, which is shown in algorithm 10. So, the repeated process of updating L_{ij} can be seen as multiplying the adjacency matrix with itself repeatedly until we get the $L^{|V|}$. We can speed up this process by squaring the adjacency matrix since only $L^{|V|}$ is needed. The fast algorithm is shown in algorithm 9. Compared to repeatedly multiplying the adjacency matrix with itself, the main change is to reduce the number of iterations of the outermost loop.

Algorithm 11: THE_FLOYD_WARSHALL

```
Input: L
1 for $k \leftarrow 1$ to $|V|$ do
2     for $i \leftarrow 1$ to $|V|$ do
3         for $j \leftarrow 1$ to $|V|$ do
4             $L_{ij}$ = min($L_{ij}$, $L_{ik}$ + $L_{kj}$)
5         end
6     end
7 end
8 return L
```

Third, the Floyd-Warshall algorithm is designed for all-pairs shortest paths problem. It also builds on the idea of dynamic programming. Instead of the number of vertices in the shortest path, the Floyd-Warshall algorithm considers the intermediate vertex of the shortest path. The vertices in the graph is numbered from 0 to $|V| - 1$, so $V = \{0, ..., |V| - 1\}$. First, consider all paths from vertex *i* to *j* whose all intermediate vertices are in the set $\{0, ..., k - 1\}$ and suppose path *q* is the shortest among them, then we consider the relationship between *q* and the shortest path *p* which corresponds to the set $\{0, ..., k\}$. If vertex *k* is not an intermediate vertex of *p*, then the weight of *q* equals the weight of *p*. If vertex *k* is an intermediate vertex, then *p* can be decomposed to two shortest paths, one path *m* from vertex *i* to vertex *k*, the other path *n* from vertex *k* to *j*. As a result, we only need to compare the weight of path *q* and the sum of weight of path *n* and *m*. Same as the matrix multiplication algorithm, the Floyd-Warshall algorithm's input is an adjacency matrix. As shown in algorithm 11, the outermost loop keeps extending the intermediate vertices set, and the shortest path distance between vertex *i* and *j*, L_{ij}, is updated by the min(L_{ij}, L_{ik} + L_{kj}). The algorithm's running time is O($|V|^3$), which is more efficient than the matrix multiplication algorithm whose time complexity is O($|V|^3 log|V|$). However, it is not clear which algorithm will achieve better results after parallelization.

B. Parallelization of proposed algorithms

First, run the Bellman-Ford algorithm on all vertices in parallel. In the previous sections, efficient parallel implementation of the Bellman-Ford algorithm is presented and discussed in detail. To achieve the task level parallelism, which is to process all vertices in parallel, streams can be used to overlap different kernel executions. Besides stream, dynamic parallelism, introduced by the Kepler architecture, enables the device to launch child kernels. For all-pairs shortest path problem, we can first launch a parent kernel and then call a child kernel to handle every vertex in parallel. However, these implementations might not get considerable speedup. Huge amount of blocks exceeding the limit of hardware can only wait until one streaming multiprocessor is available, which is explained in previous sections .

Algorithm 12: PARALLEL_FLOYD_WARSHALL

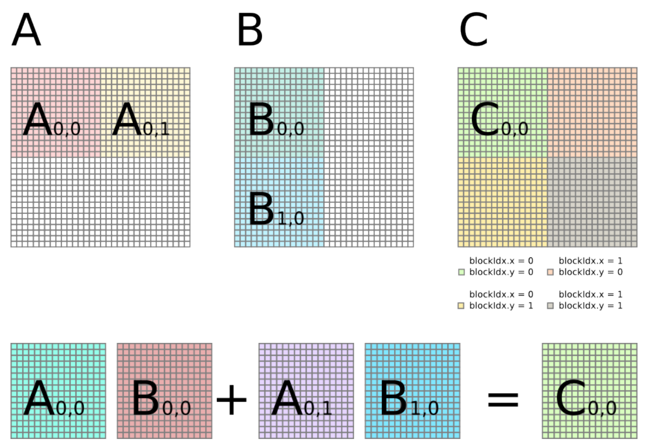
```
Input: L
1 i = getThreadIDx;
2 j = getThreadIDy;
3 L[$i$][$j$] = min($L[i][j]$, $L[i][k] + L[k][j]$)
```

Second, basic parallel Floyd-Warshall algorithm is proposed by P.Harish[6]. Due to the adjacency matrix representation, a 2D kernel can directly update all elements in parallel. Although the code is simple, the implementation is suffers from global data access. Each pair of threads requires 3 elements from global memory and one access to store the result back.

Algorithm 13: MATRIX_MULTIPLICATION

```
Input: L
1 i = getThreadIDx;
2 j = getThreadIDy;
3 for $k \leftarrow 0$ to $|V| - 1$ do
4     L[$i$][$j$] = min($L[i][j]$, $L[i][k] + L[k][j]$)
5 end
```

Third, the matrix multiplication algorithm, which runs in O($|V|^3 log|V|$). Similar to the parallel Floyd-Warshall algorithm, a 2D kernel, as in algorithm 13, can assign a thread to each element, but there is an loop inside the kernel. This basic implementation faces the same problem as the implementation proposed by P.Harish. However, this algorithm can be characterized as matrix multiplication, whose parallelization is studied in details. So, the following optimization is based on the previous improvement on parallel matrix multiplication[11][12][13].


A
B
C
$A_{0,0}$
$A_{0,1}$
$B_{0,0}$
$B_{1,0}$
$C_{0,0}$
blockIdx.x = 0 blockIdx.y = 0
blockIdx.x = 1 blockIdx.y = 0
blockIdx.x = 0 blockIdx.y = 1
blockIdx.x = 1 blockIdx.y = 1
$A_{0,0}$ $B_{0,0}$ + $A_{0,1}$ $B_{1,0}$ = $C_{0,0}$

Tiling is used to reduce the global memory transaction. As shown in the figure above[13], the target matrix *C*, which is *A* * *B*, is divided into 4 tiles. Each thread block computes one tile of the target matrix. Each tile is computed in several phases so that the input tiles can be transferred to the shared memory, reducing the global memory access. For example, to compute C_{00}, we first load the A_{00} and B_{00} to shared memory, then compute A_{00} * B_{00}. After another iteration, we add A_{00} * B_{00} to A_{01} * B_{10} to get the C_{00}. Algorithm 14 is the pseudocode for tiled matrix multiplication. First, divide the input matrix into tiles of size 32 * 32. In each iteration of the outer loop, two tiles, one column tile and one row tile, are transferred to the shared memory. The inner loop is responsible for calculating the minimum. In the end of the outer loop, all column tiles and row tiles' values are calculated, so we get the target tile's result.

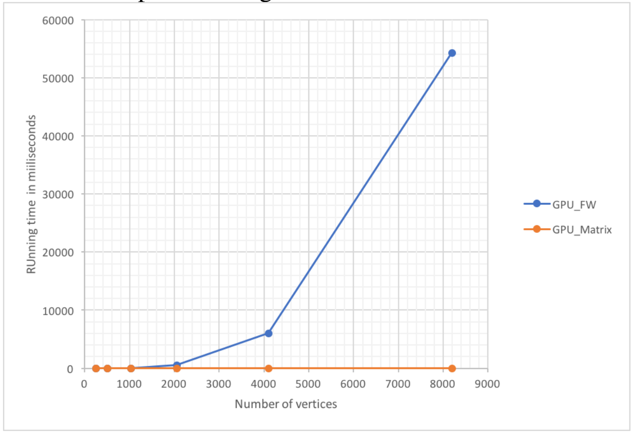
Algorithm 14: TILED_MULTIPLICATION

```
Input: L
bx = blockIdx.x;
by = blockIdx.y;
tx = threadIdx.x;
ty = threadIdx.y;
row = by * 32 + ty;
col = bx * 32 + tx;
__shared__ atile[32][32];
__shared__ btile[32][32];
tmp = L[row][col];
for $m \leftarrow 0$ to $|V|/32 - 1$ do
    atile[ty][tx] = L[row][m * 32 + tx];
    btile[ty][tx] = L[m * 32 + ty][col];
    __syncthreads();
    for $k \leftarrow 0$ to 31 do
        tmp = min(tmp, atile[ty][k] + btile[k][tx];
        __syncthreads();
    end
end
L[row][col] = tmp;
L$[i][j]$ = min($L[i][j]$, $L[i][k] + L[k][j]$);
```

However, global memory access in the tiled matrix multiplication can be further improved. In C programming language, 2 dimensional array is row major. When *atile* is transferred to the shared memory, the access to input matrix *L* is coalesced since consecutive threads read from consecutive addresses. However, the access to the *btile* is not coalesced. To solve this problem, it is necessary to transpose *L* first. In 2009, G.Ruetsch proposes an efficient parallel implementation of matrix transpose[14]. After getting the transpose of *L*, *btile* can be accessed coalescely. Using transpose to increase the bandwidth utilization is initially proposed to optimize the CUDA matrix multiplication between two arbitrary matrices, so the only added work is to compute the transpose. Nevertheless, in the implementation for all-pairs shortest path problem, only one matrix *L* is transferred to the device. As a result, to take the advantage of coalesced memory access, we need another data transfer for matrix L^T. This a trade off between efficient memory access pattern and the data transfer.

C. *Performance*

In this section, I compare the performance between parallel implementations of the basic Floyd-Warshall algorithm and matrix multiplication algorithm.



The parallel matrix multiplication algorithm gets huge speedup over the naive parallel Floyd-Warshall algorithm. It is unexpected due to the sequential algorithms' time complexity, but it can be explained. First, the parallel matrix multiplication algorithm accesses the memory much more efficiently than the other algorithm. Second, the outer loop on the host for matrix multiplication algorithm only runs $log|V|$ iterations, which are smaller than the $|V|$ iterations for Floyd-Warshall algorithm.

VI. Conclusion

In this report, I present a modified Bellman-Ford algorithm which is well suited for parallel architecture. In this version, threads are mapped to edges to exploit more parallelism, which is more efficient than mapping to vertices. Also, variants for huge graphs and general graphs which preserve the advantages of the basic parallel version are proposed. Aside from the parallelization algorithm itself, streams and pinned memory are used to minimize the cost of data transfer. At last, I introduce the all-pairs shortest path problem and the parallelization of the proposed algorithms.

References

1. F. B. Zhan, and C. E. Noon, Shortest Path Algorithms: An Evaluation Using Real Road Networks, Transportation Science, 1996.

2. Cormen, T. H., Cormen, T. H. (2001). Introduction to algorithms. Cambridge, Mass: MIT Press.

3. K. Lee, J. H. Kim, T. S. Chung, B. S. Moon, H. Lee and I. S. Kohane, Evolution strategy applied to global optimization of Clusters in Gene expression data of DNA Microarrays, Proc. 2001 IEEE Congress on Evolutionary Computation, May 2001, vol. 2, pp. 845-850, doi:10.1109/CEC.2001.934278.

4. R. Bellman, On a Routing Problem, Quarterly of Applied Mathematics, vol. 16, pp. 8790, 1958.

5. L. A. Ford, Network Flow Theory, Report P-923, The Rand Corporation, 1956.

6. P. Harish, V. Vineet and P. J. Narayanan, Large graph algorithms for massively multithreaded architectures, Technical Report IIIT/TR/2009/74, International Institute of Information Technology Hyderabad, India, 2009.

7. Hajela, G. and Pandey, M. (2014) Parallel Implementations for Solving Shortest Path Problem Using Bellman-Ford. International Journal of Computer Applications (0975-8887), 95, 1-6.

8. S. Kumar, A. Misra, and R. Tomar, A modified parallel approach to single source shortest path problem for massively dense graphs using cuda, in Computer and Communication Technology (ICCCT), 2011 2nd International Conference on, sept. 2011, pp. 635-639.

9. A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel GPU methods for single source shortest paths. In Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium, pages 349-359, May 2014.

10. Busato, F. Bombieri, N. (2016). An Efficient Implementation of the Bellman-Ford Algorithm for Kepler GPU Architectures.. IEEE Trans. Parallel Distrib. Syst., 27, 2222-2233.

11. R. Hochberg. Matrix multiplication with cuda-a basic introduction to the cuda programming model. Shodor, 2012.

12. B. Scott. 2012. Matrix Multiplication Using Shared Memory.

13. Matrix multiplication in CUDA. 2014 http://www.es.ele.tue.nl/ mwi-jtvliet/5KK73/?page=mmcuda

14. G. Ruetsch and P. Micikevicius, Optimizing matrix transpose in CUDA, NVIDIA Technical Report, 2009.