ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/320138488

Accelerating Johnson's All-Pairs Shortest Paths Algorithm on GPU

Conference Paper · September 2017

CITATIONS
0

READS
334

3 authors, including:

Oğuzhan Taştan
Middle East Technical University

4 PUBLICATIONS **0** CITATIONS

SEE PROFILE

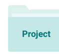
Alptekin Temizel
Middle East Technical University

73 PUBLICATIONS **596** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

IRIS Towards Natural Interaction and Communication View project



Aflatoxin detection on Siirt pistachio species with hyperspectral imaging View project

All content following this page was uploaded by Oğuzhan Taştan on 02 October 2017.

The user has requested enhancement of the downloaded file.

Accelerating Johnson's All-Pairs Shortest Paths Algorithm on GPU

Oğuzhan Taştan[1], Oğul Can Eryüksel[1], Alptekin Temizel[2]

1. Department of Computer Engineering
Middle East Technical University
{oguzhan.tastan,ogul.eryuksel}
@ceng.metu.edu.tr

2. Graduate School of Informatics
Middle East Technical University
atemizel@metu.edu.tr

Abstract

In graph theory finding shortest paths from each node to all the others is a common problem, known as all-pairs shortest path (APSP). However, it is challenging to process large graphs containing hundreds of thousands nodes and vertices in feasible time for real world applications. In this paper, we present a parallel implementation of Johnson's algorithm, which solves APSP problem on recent GPU architecture. Proposed algorithmic and architectural optimizations results in more than 4.5 times speed up of all-pairs shortest path calculation for large graphs with respect to the CPU. The source code is publicly available at https://github.com/ouzan19/JohnsonAlgoCUDA .
.

Keywords— CUDA, GPU, Johnson's Algorithm, APSP, Parallel Processing.

1. Introduction

In real world, many problems require computation of the shortest paths, such as query processing in spatial databases [1], Internet route planners [2], web searching [3], VLSI chip layout [4], power allocation in wireless sensor networks [5], and regenerative braking energy for railway vehicles [6].

There are three main solutions to the all-pairs shortest path problem. First of all, Dijkstra's algorithm [7] solves single source shortest path problem by updating the tentative distances of the neighboring nodes according to the edge costs starting from a source node with the time complexity of O (V^2) where V is the number of nodes in the graph. The improved version of the algorithm [8] which exploits priority queue runs in O ($V \log (V)$). The approach can be used to solve the all-pairs shortest path problem by applying Dijkstra's algorithm for each node in the graph which results in O ($V^2 \log (V)$).

However, the algorithm runs with only positive weighted edges. In order to handle the negative weighted graphs as in [5] and [6], Floyd-Warshall [9] algorithm can be used. The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices with time complexity of O(V^3). An alternative to Floyd-Warshall algorithm is Johnson's algorithm [10] which reweights the edges of the graph by using Bellman-Ford [11] algorithm so that no negative weighted edge exists and then applies the Dijkstra's algorithm for each node. Thus, the time complexity of the Johnson's algorithm is O($VE + V^2 log(V)$) where V is the number of nodes and E is the number of edges in the graph.

Since the algorithms to solve shortest path problems are computationally costly, the problem cannot be solved sequentially in a feasible time for real-world problems having large graph sizes. In order to overcome this problem, parallel computing is required.

Some parallel implementations of shortest path algorithms have been studied previously. Bader et al. [12] provided parallel algorithm for shortest path problem by making use of CRAY supercomputer. Crauser et al. [13] have implemented PRAM version of Dijkstra's algorithm. Although these methods produce significantly good results, they make use of expensive hardware.

Another alternative is to use hardware accelerators such as Graphic Processing Units (GPUs) which provide high computational cost at low cost. GPUs have massively parallel architectures and can be programmed with frameworks such as CUDA [14] or OpenCL [15].

In this paper, we present three different parallel implementations of Johnson's all-pairs shortest path algorithm on GPU. We compare these three implementations in terms of their execution times performance and discuss the advantages and disadvantages of the implementations.

The rest of the paper is organized as follows: related work in parallel shortest path implementations are summarized in Section 2. The CUDA programming model is presented in Section 3. In Section 4, algorithmic overview is explained. Two different parallel implementation of the algorithm is given in Section 5. Experimental setup and results are presented in Section 6 and concluding remarks are given in Section 7.

2. Related Work

With the advance of general purpose GPU computing, the massively parallel shortest path algorithms extensively studied during last years. Harish et al. [16] implemented Floyd-Warshall algorithm in CUDA by using V^2 threads in each of V iterations. Okuyama et al. [17] improved Harish's method by using the on-chip memories. Lund et al. [19] implemented the algorithm with multi-stage CUDA kernels.

Dijkstra's algorithm and Bellman–Ford algorithm are two well-known algorithms for solving the single-source shortest path problem. Meyer et al. [20] implemented delta stepping method to make Dijkstra's algorithm parallel. Arranz et al. [21] presented a method based on the technique proposed by Crauser et al. [13]. Harish et al. [16] implemented the ijkstra's algorithm in CUDA. Singh et al. [22] implemented the modified version of Dijsktra's algorithm as node-based and edge-based approach.

Agarwal et al. [23] implemented parallel version of Bellman-Ford algorithm by using two-flag approach which finds those edges which should be relaxed at next iteration. This reduces the number of iterations, thus the execution time. However, there exists much branch divergence in the implementation of kernels. Busato et al. [24] avoided this by using queue data structure and also improves it by using edge classification method, dynamic parallelism and dynamic virtual warps technique.

Pogorilyy et al. [18] implemented the parallel Johnson's algorithm with delta stepping by formalizing it in terms of Gluskov's modified systems of algorithmic algebra.

They handled the frontiers using a queue at the expense of atomic operations. In this paper, apart from queue-based approach, flag-based and prefix-based approaches have also been implemented and discussed by exploiting advanced GPU features to improve the performance. It includes parallel implementations of both Bellman-Ford and Dijkstra's algorithm.

We have adopted the queue method and edge classification optimization from Busato et al. [24].We have implemented the parallel version of Dijkstra's algorithm based on Crauser et al. [10]. Different to the work of Arranz et al. [21], we prevent branch divergence due to frontier flag usage by using the proposed queue structure to keep track of frontier nodes.

3. CUDA Overview

SPMD (Single Program Multiple Data) model of the GPU allows the same sets of instructions of a program (kernel) to be executed on different data items in parallel. GPU uses a large number of light-weight threads which are mapped to the different cores of the GPU. GPUs could be programmed using established programming frameworks such as the popular CUDA framework by NVIDIA [25].

NVIDIA GPUs have multiple SMs, which are composed of multiple independent processing elements. GPUs have multiple levels of memory for each processing element. A fast, private register memory, shared memory which is accessible to all processing elements in any SM, and global, constant and texture memories which are present on device DRAM are accessible to all processing elements of the GPU.

In CUDA [26], the programmers define a set of instructions under a device kernel function and these instructions are executed by all threads on the GPU.A block is a group of threads which can be assigned to cores under an SM. Multiple threads can be assigned to a core and similarly multiple blocks can be assigned to an SM and each thread gets a unique thread ID in an SM. Blocks can be further grouped to form a grid, where each block gets a unique Block ID. These Thread IDs and Block IDs are used by a thread to uniquely identify the data item on which it is supposed to work.

4. Johnson's Algorithm

The Johnson's algorithm includes three main steps. In the first step, an extra node, called q, is added to the graph with zero weighted edges to all other existing nodes. Then, the shortest path from q to all other nodes is calculated by the modified version of Bellman-Ford algorithm [11]. In the second step, the costs of the edges are re-weighted by using the costs of the shortest path calculated in the first step. Then, in the last stage, the extra node is removed from the re-weighted graph and Dijkstra's algorithm [7] is used to find the shortest paths from each node s to every other vertex in the re-weighted graph.

The adjacency list representation is used to store the graph. With adjacency lists, a graph $G(E,V)$ represented as follows: Vertices of graph are represented as array, say V_a; another array of adjacency list stores the edges with edges of vertex $i+1$ immediately following the edges of vertex i for all i in V. Each entry in the vertex array V_a corresponds to starting index of its adjacency list in the edge array E_a. Each entry of the edge array E_a refers to a vertex in the vertex array V_a (Figure 2). With the help of this representation, when processing nodes in parallel, we can have coalesced access to the adjacencies of a node in the graph.

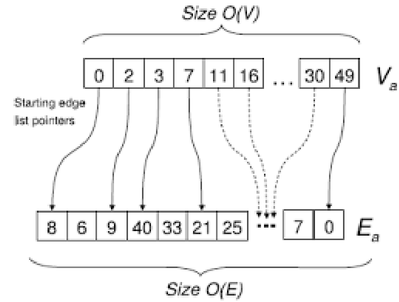


Figure 2: Graph representation with vertex list pointing to a packed edge list (adapted from [16])

4.1 Bellman-Ford Algorithm

The Bellman-Ford algorithm first initializes the distance to the source to 0 and all other nodes to infinity. Then, relaxing operation is performed for each edge V times, where V is the number of vertices in the graph. The relax operation for an edge from u to v with the weight w is follows:

if distance[u] + w < distance[v]:
distance[v] := distance[u] + w
predecessor[v] := u

We use the modified version of the algorithm [24] with two modifications: active vertices and edge classification. The active vertices approach is based on the idea that the vertices whose cost value has not been updated in previous iteration are ignored in the current iteration. This is achieved by using a queue structure and decreases the number of relax operations, thus the execution time diminished.

In edge classification approach, the edges are classified into four categories and processed differently to decrease the number of relax operations. The categories are as follows:
Self-loop edge class includes the edges whose both ends are the same vertices. There is no need to relax these edges.
Source edge class includes the edges which are the source edges. For this kind of edges, relax operation is replaced by direct update by cost of source vertices.
In-degree edge class includes the edges whose in-degree is equal to 1. Since these kinds of edges are visited only once, there is no need for atomic operations
Out-degree edge class includes the edges whose out degree is equal to zero. These edges are ignored during algorithm iterations and assigned at the end of the algorithm without using atomic operations.

The pseudo-code for the modified version of Bellman-Ford algorithm is shown in Algorithm 1.

4.2 Reweigthing

In this step, the weights of the edges are modified using the values calculated in the previous step. An edge, from u to v with weight w, is updated by $w+h(u) -h(v)$ where $h(.)$ function gives the cost of the shortest path between q and the given vertices, which was calculated in the previous step. This step is required to convert negative-weighted graph into nonnegative-weighted graph so that the Dijkstra's algorithm can run in the next step.

4.3 Dijkstra's Shortest Path Algorithm

The Dijkstra's algorithm first initializes the distance to the source to 0 and all other nodes to infinity. Then, it creates two sets, one is called visited and the other one is unvisited and

marks the source node as visited. Starting from current node; for all neighbors in the unvisited set, it performs relax operation. When all neighbors of the current node are done, it marks the current node as visited and deletes it from unvisited set. Then, a node is selected from the unvisited set as the current node and the relaxation process is repeated. The algorithm finishes when the unvisited set has no element.

```
for each u in V do
        d(u) = INF
d(s) = 0
F1  ← {s}
F2  ← { }
while F1 is not empty ; do
        for vertices in F1 do
                u  ← DEQUEUE(F1)
                for vertices v in adj [u] do

                if( u == v OR out-degree(v) == 0) then
                        skip
                else if(u == s OR in-degree(v) == 1) then
                        d(v) = d(u) + w
                                ENQUEUE(F2; v)
                        else
                        d(v) = min(d(v),d(v)+w);
                                ENQUEUE(F2; v)
                end
        end
        SWAP(F1, F2)
end
```

Algorithm 1: Pseudo code for optimized sequential Bellman-Ford algorithm [24]

5. Parallel Implementations of the Algorithm

In this work, we have implemented three different versions of parallel Johnson's algorithm. The difference between versions is the way of keeping track of frontier vertices in the Dijkstra part of the algorithm. One of them, called flag-based, uses a Boolean array which stores a Boolean value, which indicates whether the node is frontier or not, for each node in the graph. The second, called Q-based, puts the frontiers into a queue using atomic operations. And the other, called prefix-based, puts the frontiers into queue using prefix sum. The first two steps of the versions are common.

5.1 Parallel Bellman-Ford Algorithm

In the algorithm, each edge can be processed independent of others; therefore, they can be processed in parallel. One thread will be assigned to each edge and all threads are synchronized, and then the next iteration begins. Main kernel for Bellman-Ford algorithm is as in the Algorithm 2. Each thread is responsible for one vertex and each vertex first initialize the corresponding index of node weight array, then all threads other than the first terminates. The first thread controls the main algorithm. It first put the source node into the queue, and then relaxes the nodes in the queue by calling another kernel which is shown in the Algorithm 4. This child kernel simultaneously extracts the nodes from queue, relaxes them and puts the updated ones into another queue. Then, main kernel swaps the queues, and repeats the process until the queue is empty.

The update and enqueue operation in the relax kernel should be atomic because many threads try to update the same queue and the same node.

Since the nodes have different number of adjacent vertices, the fair work load balancing may not be achieved during the algorithm if the adjacent vertices are processed in a sequential manner in a kernel. Instead, we exploit the dynamic parallelism feature of Maxwell architecture which allows dynamically creating threads at run time without the need of kernel returns [27]. Thanks to this feature, we can invoke a child kernel which relaxes adjacencies of the node in the main kernel; therefore, all adjacent nodes in the main kernel are processed simultaneously which results in fair work load.

```
BEGIN
    tid = blockIdx.x * blockDim.x + threadIdx.x;
    // initialize
    if( tid == s ) then
      V[tid] = 0;
    else
      V[tid] = INF;
    endif

    __syncthreads();

    // until Q is empty, relax nodes in the Q
    if(tid == 0) then
          ENQUEUE(Q1,s);
      while(! isEmpty(Q1)) do
                    relaxQueue<<<    >>>(E,V,Q1,Q2,s);
                    SWAP(Q1,Q2);
          End while
    Endif
END
```

Algorithm 2: Pseudo kernel for Bellman-Ford. *V* represents the node weight array, *E* represents the edge costs, s is the source node, *F1* and *F2* are the queues. *ENQUEUE* function puts the given node into given queue.

5.2 Reweigthing

In this step, all edge costs are updated completely independent of each other's. Each thread is responsible from one node. Each thread updates the costs of out-going edges of corresponding node by using the node weights calculated by Bellman-Ford algorithm in previous step according to the formula given in previous section as shown in Algorithm 3.

```
BEGIN
    tid = blockIdx.x * blockDim.x + threadIdx.x;
    for each v in ADJ(tid)
        E(tid,v) = E(tid,v) + V(tid) – V(v);
    end for
END
```

Algorithm 3: Reweighting kernel.

5.3 Parallel Dijkstra Algorithm

Unlike Bellman-Ford algorithm, it is not straight forward to parallelize Dijkstra's algorithm because of its sequential nature. Although there can be different approaches, we parallelize the inner operations of the Dijsktra's algorithm. At outer loop, the algorithm chooses a node to compute new distance values. Inner operations relax outgoing edge values to update node labels. If we can select frontier nodes that can be updated separately without affecting the correctness of the algorithm, the algorithm can be parallelized efficiently. The method to define the frontier set is explained in [21].

At each iteration of Dijkstra's algorithm, we need to identify the nodes which will be inserted into the frontier set, then we can relax those nodes in parallel. Crauser et al [13] defined an algorithm that augments the frontier set with nodes with bigger tentative distance. At each iteration *i,* algorithm

for each node of the unsettled set, $u \in U_i$, the sum of; its tentative distance and the minimum of the cost of its outgoing edges. Then, it chooses the minimum of those values. Finally, those nodes, whose tentative distance are lower or equal than the minimum value, are inserted into the frontier set. The process repeated until the threshold reaches infinity as shown in Algorithm 5.

```
BEGIN
    tid = blockIdx.x * blockDim.x + threadIdx.x;
    nodeId = DEQUEUE(Q1, tid);

    for each v in ADJ(nodeId)
      if (nodeId == v) then
              continue;
      endif
      cost =  EDGECOST(nodeId,v);
      BEGIN ATOMIC
      if( cost + V[v] < V[nodeId]) then
              V[nodeId] = cost + V[v];
                outDegree = ADJ(v).size();
              if( outDegree > 0)then
                      ENQUEUE(Q2,v)
              endif
       endif
      END ATOMIC
    end for
END
```

Algorithm 4: Pseudo kernel for relax operation. DEQUEUE function extracts from given queue corresponding to given index. ADJ function returns the neighbors of the given node. All other variables are the same with Algorithm 2.

Arranz et al. [21], describe a modified version of method of Crauser et al. They introduce a new concept of Δ_i as the limit value computed in each iteration *i* that holds any unsettled node *u* with its node weight less than Δ_i can be settled safely. Their method proceeds as follows: First, for each node in graph, the algorithm calculates minimum edge cost among its outgoing edges. Second, for each iteration *i* of the external loop, having all tentative distances of the nodes on the unsettled set, it calculates a threshold value that is the minimum value of sum of node weight and corresponding value calculated in the first step among the unsettled nodes. Finally, it possible to put into the frontier set every node whose node weight is greater than the threshold value calculated in previous step.

Minimum function called in main kernel is the modified version of the advanced reduce3 method of CUDA SDK [28] in order to calculate the threshold value. The update function is shown in the Algorithm 7.

The problem with this implementation is that the flags of the nodes in the frontier set are not in consecutive locations in the flag array and this causes branch divergence. In order to avoid branch divergence, we have embedded the idea of frontier propagation using queue data structure, as in the parallel Bellman-Ford algorithm, into the parallel Dijkstra's algorithm. To implement queue in GPU, we use two different approaches, namely Q-based and prefix-based approaches.

In the main part of Q-based approach, instead of setting the frontier flag of the source node, it is put into the queue. The relax and update kernel of Q-based parallel Dijkstra's algorithm is shown in Algorithm 8 and Algorithm 9 respectively. The cost of this modification to the flag-based algorithm is the atomic operations when queue is being filled. Although it gets rid of the branch divergence significantly, all threads needs to update the queue sequentially in order to avoid race conditions. The time performance comparison of two approaches is presented in the Section 6.

```
BEGIN
        tid = blockIdx.x * blockDim.x + threadIdx.x;
        // initialize
        if( tid == s ) then
          V[tid] = 0;
          F[tid] = 1;
         U[tid] = 0;
        else
          V[tid] = INF;
          F[tid] = 0;
          U[tid] = 1;
        endif
        if(tid == 0) then
        Δ = ∞
         while (Δ<∞) do
              relax(U, F, V,E);
              Δ =  minimum(U, V);
              update(U, F, V, Δ) ;
         end while
        endif
END
```

Algorithm 5: Pseudo kernel for main algorithm of flag-based Dijkstra. *V* represents the node weight array, *E* represents the edge costs, *s* is the source node, *U* is the unsettled flag array, *F* is the frontier flag array, *Δ* is the threshold value.

```
BEGIN
    tid = blockIdx.x * blockDim.x + threadIdx.x;
    if(F[tid] == True) then
        for each v in ADJ(tid)
            if(U[v] == True) then
                V[v] = AtomicMin(V[v]  , V[v]+E(tid,v));
            endif
        end for
    endif
END
```

Algorithm 6: Pseudo kernel for relax operation in flag-based Dijkstra. ADJ function returns the neighbors of the given node. The other variables are the same with Algorithm 5.

```
BEGIN
        tid = blockIdx.x * blockDim.x + threadIdx.x;
        F[tid] = False
        if (U[tid] == True and V[tid]≤Δ) then
              U[tid] = False
              F[tid] = True
        end if
END
```

Algorithm 7: Pseudo kernel for update function of flag-based Dijkstra. The variables are the same with Algorithms 5 and 6.

In prefix-based approach, the elements are added into the queue by indexes which is previously calculated using prefix sum at the start of each iteration. Thus, there is no need for atomic operations (when filling queue) since the indexes are pre-defined. However, computation cost of the prefix sum is introduced in this method. The update kernel of prefix-based

algorithm is shown in Algorithm 10 and the relax kernel is the same with that of Q-based approach.

```
BEGIN
    tid = blockIdx.x * blockDim.x + threadIdx.x;
    nId = Q[tid];
    for each v in ADJ(nId)
        if(U[v] == True) then
            V[v] = AtomicMin(V[v] , V[v]+E(nId,v));
        endif
    end for
END
```

Algorithm 8: Pseudo kernel for relax operation of Q-based parallel Dijkstra. *Q* is the queue data structure. All other parameters are the same with Algorithm 5.

```
BEGIN
    tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (U[tid] == True and V[tid]≤Δ) then
        U[tid] = False
        BEGIN ATOMIC
            ENQUEUE(Q,tid);
        END ATOMIC
    end if
END
```

Algorithm 9: Pseudo kernel for update function of Q-based Dijkstra.

```
BEGIN
    tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (U[tid] == True and V[tid]≤Δ) then
    U[tid] = False
        Q[ prefixSum[tid] ] = tid ;
    end if
END
```

Algorithm 10: Pseudo kernel for update function of prefix-based Dijkstra.

To extend the single source parallel Dijkstra's algorithm to the all-pairs shortest path algorithm, we have simply applied the algorithm for each node in the graph.

6. Experimental Setup and Results

The experiments have been conducted on 64-bit Windows 10 Pro operating system with Intel Core i7-6700K CPU @4.00 GHz and 16GB DDR4 RAM. The GPU used is NVIDIA GeForce GTX 960. The dataset used in the experiments is composed of eight graphs with different number of nodes and edges (Table 1).

In this work, we compare four different implementation of Johnson's all-pairs shortest path algorithm in terms of execution time with given dataset, namely sequential, flag-based parallel implementation, Q-based parallel implementation and prefix-based parallel implementation. The execution times are shown in Figure 3.

Additionally, speed-up rates of different methods are shown in Figure 4. It is observed that Q-based approach provides higher speed-up than flag-based approach for all graphs. This is clearly caused by the fact that branch divergence has more severe drawbacks than atomic operations have. On the other hand, prefix-based approach, which is expected to be faster, has less speed-up than the other approaches. This is caused by the low input size with respect to the graphs in [24]. We could not supply higher sized inputs because of memory constraint in GPU since the problem is all-to-all shortest path. Q-based approach can speed up the Johnson's algorithm more than 4.5 times.

In Figure 3, it can be clearly seen that, larger graphs are needed to benefit from the parallel GPU implementation. While there is no significant advantage is observed until number of edges is around 150000, GPU implementation has lower execution time with larger graphs.

Table 1: Properties of the graphs used in the experiments

Graph	# of nodes	# of edges
Graph1 [29]	6301	20777
Graph2 [29]	36682	88328
Graph3 [29]	62586	147892
Graph4 [30]	58228	428156
Graph5 [31]	75879	508837
Graph6 [32]	77360	905468
Graph7 [33]	265214	420045
Graph8 [32]	82168	948464

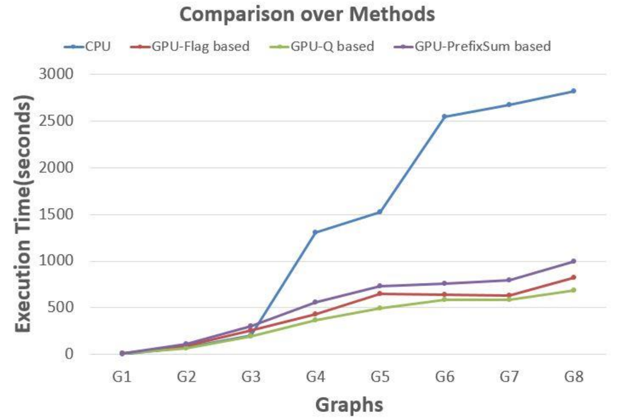


Figure 3: Execution times on different graphs.

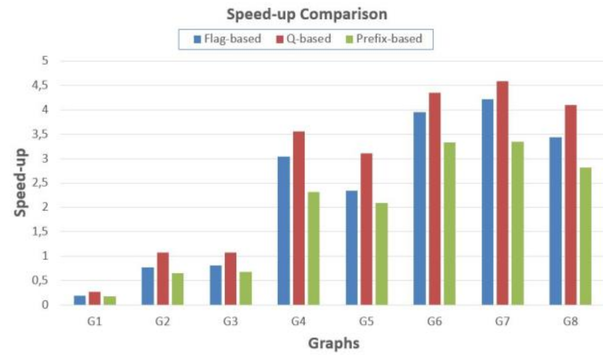


Figure 4: Speed-up rates on different graphs.

7. Conclusions and Discussion

In this paper, we proposed an efficient GPU implementation of Johnson's all-pairs shortest path algorithm. The dominant part of Johnson's algorithm is Dijkstra's part and in this part we combined queue-based approach from Busato et al. [24] and Δ- stepping approach from Crauser et al. [13] in order to make Dijkstra's algorithm parallel by avoiding branch divergence. The results show that although the proposed method suffers from atomic operations, it is more efficient than the flag-based methods due to lack of branch

divergence. However, as the input size increases, the performances difference between the two gets smaller.

The proposed method can be preferable when the input graph is negative-weighted because Dijsktra's algorithm cannot run with negative-weighted graphs. In addition, it can be more efficient than Floyd-Warshall algorithm if the input graph is sparse.

As a future work, the performance comparison between proposed method and GPU implementation of Floyd-Warshall algorithm can be done.

8.References

[1] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query processing in spatial network databases," in VLDB'03. Berlin: VLDB Endowment,2003,pp.802–813.[Online].Available: http://dl.acm.org/citation.cfm?id=1315451.1315520

[2] Rétvári, G., Bíró, J. J., & Cinkler, T. (2007). On shortest path representation. IEEE/ACM Trans. Netw., 15(6), 1293-1306.

[3] Barrett, C., Jacob, R., & Marathe, M. (2000). Formal-language-constrained path problems. SIAM Journal on Computing, 30(3), 809-837.

[4] Ashok Jagannathan. Applications of Shortest Path Algorithms to VLSI Chip Layout Problems. Thesis Report. University of Illinois. Chicago. 2000.

[5] Zhang, X., Yan, F., Tao, L., & Sung, D. K. (2014, August). Optimal candidate set for opportunistic routing in asynchronous wireless sensor networks. In Computer Communication and Networks (ICCCN), 2014 23rd International Conference on (pp. 1-8). IEEE.

[6] S. Klamt and A. von Kamp, "Computing paths and cycles in biological interaction graphs," BMC Bioinformatics, vol. 10, no. 6, pp. 1–11, 2014.

[7] Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs", Numerische Mathematik 1: 269–271. doi:10.1007/BF01386390.

[8] Fredman, Michael Lawrence; Tarjan, Robert E. (1984). Fibonacci heaps and their uses in improved network optimization algorithms. 25th Annual Symposium on Foundations of Computer Science. IEEE. pp. 338–346. doi:10.1109/SFCS.1984.715934

[9] Floyd, Robert W. (June 1962). "Algorithm 97: Shortest Path". Communications of the ACM 5 (6): 345. doi:10.1145/367766.368168.

[10] Johnson, Donald B. (1977), "Efficient algorithms for shortest paths in sparse networks", Journal of the ACM 24 (1): 1–13, doi:10.1145/321992.321993

[11] Bellman, Richard (1958). "On a routing problem". Quarterly of Applied Mathematics 16: 87–90. MR 0102435

[12] Bader, D. A., & Madduri, K. (2006, August). Parallel algorithms for evaluating centrality indices in real-world networks. In Parallel Processing, 2006. ICPP 2006. International Conference on (pp. 539-550). IEEE.

[13] Crauser A., Mehlhorn K., Meyer U., and Sanders P. 1998. A Parallelization of Dijkstra's Shortest Path Algorithm. MFCS'98- LNCS 1450, Lubos Prim et al. (Eds.), Springer-Verlag Berlin Heidelberg, pp. 722-731.

[14] NVIDIA CUDA: http://www.nvidia.com/cuda

[15] Khronos OpenCL : https://www.khronos.org/opencl

[16] Harish, P., & Narayanan, P. J. (2007, December). Accelerating large graph algorithms on the GPU using CUDA. In International Conference on High-Performance Computing (pp. 197-208). Springer Berlin Heidelberg.

[17] T. Okuyama, F. Ino, and K. Hagihara, "A task parallelalgorithm for finding all–pairs shortest paths using the gpu," International Journal of High Performance Computing and Networking, vol. 7, no. 2, pp. 87–98, 2012.

[18] Pogorilyy, S. D., Slynko, M. S., & Rustamov, Y. I. (2017). Research and development of Jonhson's algorithm parallel schemes in GPGPU technology. TWMS Journal of Pure and Applied Mathematics, 8(1), 12-21.

[19] Lund, B. D. & Smith, J. W. (2010). A Multi-Stage CUDA Kernel for Floyd-Warshall. CoRR, abs/1001.4108.

[20] U. Meyer and P. Sanders, "delta-stepping: a parallelizable shortest path algorithm," Journal of Algorithms, vol. 49, no. 1, pp. 114–152, 2003.

[21] H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "A New GPUbased Approach to the Shortest Path Problem," in The 2013 International Conference on High Performance Computing & Simulation, (HPCS 2013), p. To appear, 2013.

[22] D. P. Singh and N. Khare, "A study of different parallel implementations of single source shortest path algorithms," International Journal of Computer Applications, vol. 54, no. 10, pp. 26–30, September 2012, published by Foundation of Computer Science, New York, USA.

[23] P. Agarwal and M. Dutta, "New Approach of Bellman Ford Algorithm on GPU using CUDA", Int. Journal Computer Applications, vol. 110, no. 13, pp. 11-15, January 2015.

[24] F. Busato and N. Bombieri, "An efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures", IEEE Trans. Parallel Distrib. Syst., vol. pp, no.99, September 2015.

[25] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," ACM Queue, vol. 6, no. 2, pp. 40-53, 2008.

[26] NVIDIA Corporation, CUDA C programming guide (2013),http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[27]http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cuda-dynamic-parallelism

[28] M. Harris, Optimizing Parallel Reduction in CUDA, NVIDIA, 2008.

[29] M. Ripeanu and I. Foster and A. Iamnitchi. Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design. IEEE Internet Computing Journal, 2002.

[30] Cho, E., Myers, S. A., & Leskovec, J. (2011, August). Friendship and mobility: user movement in location-based social networks. In Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 1082-1090). ACM.

[31] Richardson, M., Agrawal, R., & Domingos, P. (2003, October). Trust management for the semantic web. In International semantic Web conference (pp. 351-368). Springer Berlin Heidelberg.

[32] J. Leskovec, K. Lang, A. Dasgupta, M. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. Internet Mathematics 6(1) 29--123, 2009.

[33] Leskovec, J., Kleinberg, J., & Faloutsos, C. (2007). Graph evolution: Densification and shrinking diameters. ACM Transactions on Knowledge Discovery from Data (TKDD), 1(1), 2.

View publication stats