PARALLEL IMPLEMENTATION AND ANALYSIS OF DIJKSTRA'S ALGORITHM USING OPENMP AND CUDA

Jianbo Pei and Jingyuan Wang

Department of Electrical Engineering and Computer Science
University of California, Irvine

ABSTRACT

The goal of our project is to parallelize Dijkstra's algorithm, a well-known algorithm for computing a single source shortest path in a graph. The parallelization is implemented using OpenMP and CUDA. The results of the two implementations demonstrates that both of them have a better performance than serial version. Comparing the two methods, CUDA shows a higher speed-up than OpenMP. The improvement rises with the increase of the number of nodes. The results satisfy our expectations and it shows noticeable efficiency for Dijkstra's algorithm.

1. INTRODUCTION

As we all know that finding the shortest path is a crucial problem in the graph theory. In a given graph, where graph can be undirected or directed, it treats all the destinations as nodes and the connection between any two nodes as the path with weight represents the distance. The task of finding the shortest path could be divided into two different categories based on the specific requirement. The first is to find the shortest path of one node to the other nodes. The second is to find the shortest paths between any given nodes.

There are many algorithms already proposed to solve the shortest path problem, such as Dijkstra's algorithm, Bellman-Ford algorithm, Floyd-Warshall algorithm, and Johnson's algorithm. Each algorithm was proposed to solve different kinds of shortest path problem. Among these algorithms, Dijkstra's algorithm has the big advantage of finding the shortest path from one node to the other nodes without changing the graph data structure. Also, with no change in the graph data structure, the Dijkstra's algorithm only needs to run once until it will visit all the reachable nodes [1]. Based on the advantages of Dijkstra's algorithm, we can see many applications have adopted Dijkstra's algorithm over the other algorithms, such as routing systems in computer network and GPS systems, etc.

Originally, Dijkstra's algorithm is running in sequential matter, which means that one process executes after the other in one processor. It seems to be inefficient since modern computers have already equipped with multiple processors. Therefore, our project goal is to parallelize the Dijkstra's algorithm based on researching existing proposed parallelized Dijkstra's algorithm and try to improve in some extent. Eventually, we will implement our proposed method using OpenMP and CUDA and compare the performance of parallelized Dijkstra's algorithm out of these two platforms.

1.1. Related Works

Yulian Yu and Wenhong Wei bring forward a method to resolve the single source shortest path problem [2]. They firstly construct a model of vector matrix multiplication, and then they replace multiplication operations with the addition operations which update the distance from the source to the other nodes, and substitute addition operations with minimum operations which select the minimum distance through comparing. With the help of those changes, they successfully parallel the computation of distance between nodes.

In the paper, *CUDA Solutions for the SSSP Problem*, it explains the GPU methods to parallelize Dijkstra's algorithm using CUDA [3]. It divides the sequential algorithm into three essential steps: first is to obtain estimates for *unresolved vertices* (*U*-vertices) that are relaxed using the last vertex added to *resolved vertices* (*R*-vertices), called *frontier vertex*; second is to find the minimum estimate of the *U*-vertices; last is to update the *U*-vertices which will compose the new set of *frontier vertex.* Then, the paper presents *Dijkstra's algorithm Adapted to Compound Frontier* (DA2CF) which consists of the same operations as above, which are called relax, minimum, and update. The paper proposes methods to parallelize these three steps that can potentially run on GPU based on these operations. The methods then are implemented using GPU and evaluated by considering adjacency lists and adjacency matrices of graphs. It also compares the performance with CPU. The results show that the performance on GPU is better than on CPU.

2. BACKGROUND

In this section, some background information about Dijkstra's algorithm, OpenMP, and CUDA will be given.

2.1. Dijkstra's Algorithm

Dijkstra's Algorithm is an algorithm aiming at finding the shortest path between the source and other nodes in a graph. In the Dijkstra's Algorithm, let S be a set of explored nodes. V is defined to be a set of all the nodes. For each node in S, we store the shortest path distance d(u) from the source s. Initially, S = {s}, d(s) = 0. For all nodes v not equal to s, d(v) are infinite. While S is not equal to V, select unexplored node v (in V but not in S) that minimizes

$$\pi(v) = \min(d(u) + l_e)$$

In the formula, le is the distance between node u and v. Then, add v to S and set d(v) to π(v). Update π(w) for all neighbors w of v.

For the step of getting π(v), we are required to firstly calculating distances of all the nodes, which is time-wasted for sequential computing. Thus, parallel computing is necessary to reduce the runtime. Our goal for the project is to implement parallel on Dijkstra's Algorithm using OpenMP and CUDA and then compare the runtime of improved algorithm with existing methods.

2.2. OpenMP

OpenMP (Open Multi-processing) is an application programming interface (API) which is composed of compiler directives, library routines and environmental variables. It is designed for shared memory and used for parallelism within a (multi-core) node. OpenMP realizes parallelism through the use of threads. The number of parallel threads can be controlled and defined before implementation. Most variables in OpenMP code are visible to all the threads by default. However, sometimes private variables are required to avoid race condition. Using fork-join model, master thread forks a specified number of threads for parallel work. Those threads execute the statements in the parallel region and then synchronize and terminate. The limitation of OpenMP is that it cannot be used on the distributed memory system.

2.3. CUDA

CUDA is a parallel computing platform and a programming model created by NVIDIA. It can implement parallel computation efficiently for GPU with programming language C and C++. CUDA is designed for a wide SIMD parallelism and it can provide a thread abstraction to deal with SIMD. Between small thread groups, synchronization and data sharing can be realized. CUDA supports thread parallelism, data parallelism and task parallelism, which means each thread, block and kernel are independent. CUDA possesses many advantages, such as parallel computing in a very large scale and faster download from GPU. The limitation of it is that it can only be utilized on NVIDIA.

3. PROPOSED METHODS

In this section, the explanation of two parallelization methods for Dijkstra's Algorithm will be conducted. First method is based on OpenMP platform. Second method is based on CUDA platform. The results will be shown in next section.

3.1. OpenMP

In the serial code, the most significant two parts are finding the closest node and updating the distance from source to each node. The implementation using OpenMP is based on the two functions [4]. Statements of OpenMP are utilized to set number of threads and to control the behavior of threads in order to ensure high performance and no data hazards**.**

The first paralleled function is findMin_p(), which is shown in Fig. 1. The purpose of this function is to find the node that is closest to the source among all the nodes. OpenMP implementation parallelizes it through splitting the whole work (in the form of for-loop) among all the threads and unrolling the for-loop. Each thread takes the responsibility to find the minimum distance and corresponding closest node in their own chunks. Each thread owns two private variables, min_dist_thread and min_node_thread which separately updates the minimum distance and the corresponding closest node in each for-loop. To get better performance, some iterations of the for-loop are unrolled. Having tested different amount of iterations being unrolled, we found the speed-up is most significant with four unrolled iterations. When jumping out of the loop, threads end up finding the closest node. Those threads then update the global closest node, which is marked with *#pragma omp critical* to ensure no race condition there. The global closest node will be marked as visited. Fig. 2 is the pseudocode of findMin_p() function.

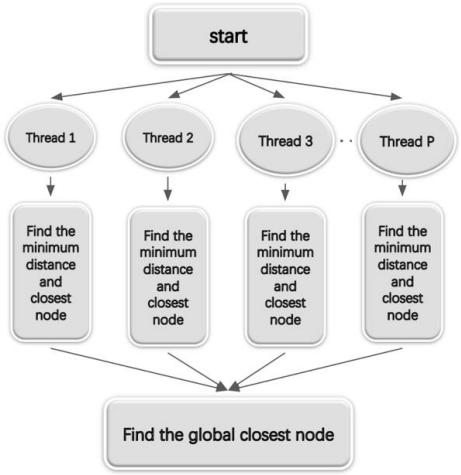

Fig. 1. Flow chart of findMin_p() function

```
int findMin_p(){
omp_set_num_threads(THREADS);
#pragma omp parallel private (min_dist_thread, min_node_thread)
shared(dist_p, visit_p)
{
    Initialize min_dist_thread and min_node_thread;
    #pragma omp barrier
    #pragma omp for nowait
    for vertex = 0 to N with increment 4 {
      if ((distance from source to vertex less than min_dist_thread)
      && (vertex has not been visited)) {
        Update min_dist_thread and min_node_thread;
      }
    }
    #pragma omp critical
    {
      if (min_dist_thread less than global min) {
        Update global closest node;
      }
    }
}
visit_p[minNode] = true;
return minNode;
}
```

Fig. 2. Pseudocode of findMin_p() function

The second paralleled part is updateDist_p() function, which aims at updating distance from source to each node after finding a global closest node. The parallel implementation in this function is similar to that in the findMin_p() function. All the work within the for-loop are divided into small chunks. Each thread holds each chunk and updates distance from source to the node in the subset if conditions are satisfied. Similarly, unrolling is applied again in this function to reduce number of instructions. Since there is no data dependency, it needs only a barrier at the end to ensure all the threads have completed their work before memory space being free. Fig. 3 is the pseudocode of findMin_p() function.

After obtaining the latest distance for all nodes, findMin_p() function is called again to find the closest node. The entire process is to go through these two functions repeatedly until all the nodes have been visited.

```
void updateDist_p(){
    omp_set_num_threads(THREADS);
    #pragma omp parallel
    {
      #pragma omp for
      for vertex = 0 to N with increment 4 {
        if (the vertex has not been visited && the vertex connects
        to the minNode && distance from source to vertex more
        than sum of minNode's distance from source and distance
        from minNode to vertex){
          distance[vertex] = distance[minNode] + graph[minNode][vertex];
        }
      }
      #pragma omp barrier
    }
}
```

Fig. 3. Pseudocode of updateDist_p() function

3.2. CUDA

In serial Dijkstra's Algorithm, there are three main operations that account for majority of execution time. For this reason, it needs to construct possible solutions to parallelize these three steps in order to speed up the algorithm execution time. The first operation is an outer for loop to iterate through all the nodes and it does not require any computation, so there is not much to do about it in CUDA. The second operation, which is called find_min(), is to find the index of unvisited node with minimum distance value in the graph. In serial implementation, inside the outer for loop, it needs to iterate through all nodes to find the needed index while comparing distance values. The running time complexity for this operation is O(n), where n is the total number of nodes. Therefore, we can potentially use CUDA to parallelize this operation to gain much speed-up. The third operation, which is called update(), is to update the distance values for all nodes that connected to the node just found in the second operation. It is another for loop after the second operation but still inside the outer for loop. In serial implementation, this operation also requires O(n) running time because it needs to iterate through all nodes. Therefore, it is also possible to try to parallelize it to gain more speed-up.

For serial Dijkstra's Algorithm, the running time complexity is $O(n^2)$, which is from one outer for loop and two consecutive inner for loops. The proposed solution to parallelize the find_min() function is using reduction. Therefore, in theory, it can achieve O(log(n)) in running time. The proposed solution to parallelize update() function is to update every node in parallel. Because updating the nodes that are connected to one single node has no dependency between the nodes, it is possible to do the update operation independently. In theory, it can achieve constant running time for update operation. However, in practice, it is almost impossible to update all

the nodes simultaneously. The running time depends on the hardware parameters, such as the available number of SM (streaming multiprocessor).

CUDA Parallelized find_min(). For CUDA implementation, closestNodeCUDA() function is used to find the unvisited node index with minimum distance values. The main method used here is reduction. In order to make the explanation more clear, the closestNodeCUDA() function will be divided into four steps.

Step 1. To operate on the data, it needs to load all the data from global memory into shared memory for each block. Here, array node_dist[] is used to store the distance values for all nodes. The index of all nodes in the array represents the index in the graph. The shared memory are cache[] and cacheIndex[] which are to store distance value and corresponding index for each block. The size for cache[] and cacheIndex[] is assigned to be the number of threads per block. The index is calculated as:

$$index = blockIdx.x + blockDim.x + ThreadIdx.x$$

Since it can be predefined that the number of blocks times the number of threads per block equals the total number of nodes, each thread can operate on each node individually. The advantage is that no threads will become idle after being just created. Fig. 4 provides an example to illustrate how step 1 works.

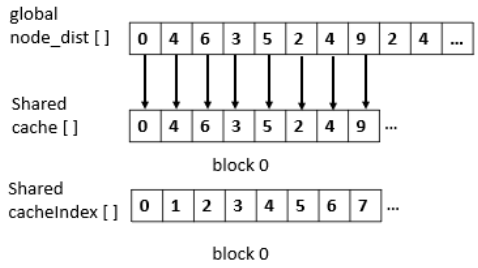

Fig. 4. Illustration of step 1

Step 2. After loading all the data to shared cache[] and cacheIndex[], it can use reduction operating on the cache[] array to find the minimum value. The main target is to find the corresponding index instead of the distance value, so it needs to keep track of the index all the time while finding the minimum distance value. For example, in Fig. 5, it is assumed there are total 8 elements in each block. It can compare every two elements with stride 4, stride 2, and stride 1 to find the minimum distance value. Again, the goal is to find the corresponding index. It can easily keep track of index while comparing the value using cacheIndex[]. After performing reduction for each block, the thread 0 at each block will hold the smallest value and the corresponding index. In the end of second step, thread 0 at each block will store the value and index to a global memory temp[] and tempIndex[] respectively. Therefore, after second step, we can obtain a temp[] that contains minimum distance value from each block and a tempIndex[] that contains the index.

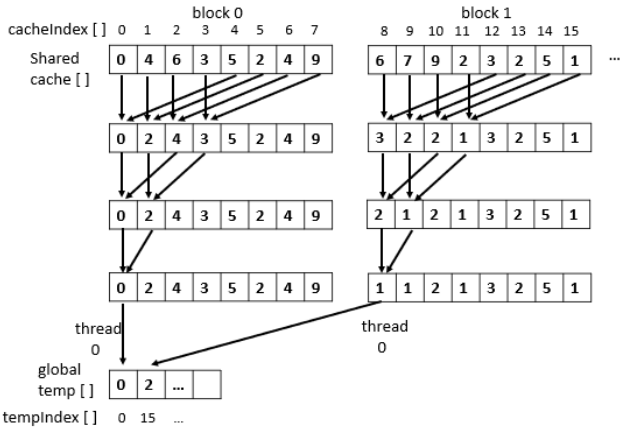

Fig. 5. Illustration of step 2

Step 3. Another reduction can be performed to find the minimum distance value and the index in temp[] and tempIndex[]. However, in this case, it does not need all threads to perform the reduction since it will be redundant. It only needs to use the threads in one block to perform the reduction. In our implementation, we used only thread 0 at block 0. After the third step, it can have minimum distance value positioned at 0 in temp[] and the corresponding index at 0 in tempIndex[]. Fig. 6 illustrates how step 3 works.

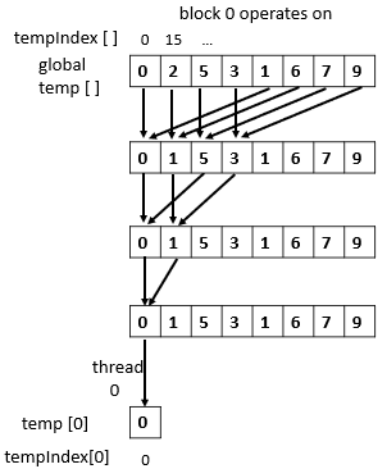

Fig. 6. Illustration of step 3

In the last step, it can store the index with minimum distance value at tempIndex[0] to a global pointer and this pointer will be used in next operation. Meanwhile, the status of this node can be updated to visited.

CUDA Parallelized update() – For CUDA implementation, the function is called cudaRelax() [5]. Since the node with minimum distance value is already found, it can update all the nodes that are connected to it. By iterating through all nodes, if a node satisfies the three conditions: 1) connected to the node with minimum distance 2) has not been visited, and 3) the distance value of the node just found plus the edge value connected to it is less than the current distance value, then it can update the distance value to minimum distance value plus the edge value. Because the nodes needed to update are independent to each other, all threads can be utilized to perform the update operation.

4. EXPERIMENTAL RESULTS

In this section, the performance of OpenMP and CUDA implementation of Dijkstra's Algorithm will be shown here. After showing the results separately, the comparison of two implementations will be provided.

4.1. OpenMP

The graph we used as input is created with rand() function and stored in the form of 2-D array graph[N][N] (N is the node amount). If node i connects directly to node j, a nonzero value should be stored in graph[i][j]. Otherwise, graph[i][j] is set to be zero. OpenMP parallelization is implemented on the computer with 8 cores. The output consists of three parts: the minimum distance between the source and other nodes, two running time of serial code and parallel code, the result of error checking. Error checking is conducted through comparing whether the result from serial code is identical to that from OpenMP. Table 1 is the running time used to execute the program with same number of threads and variable input sizes for serial and OpenMP parallel codes. It directly demonstrates that the speed-up rises with the growing of number of nodes. When the input size up to 16384, the improvement is highest, about 3.2 times faster than serial version. From Fig. 7, the change of gap between serial and OpenMP can be observed more straightforward.

Num of Nodes	512	1024	2048	4096	8192	16384
Serial	**0.005327**	**0.021351**	0.084384	0.219032	0.867634	3.12067
OpenMP	0.010836	0.022873	**0.045601**	**0.111297**	**0.286892**	**0.745239**
Threads	8	8	8	8	8	8

Table 1. Serial vs OpenMP execution time comparison

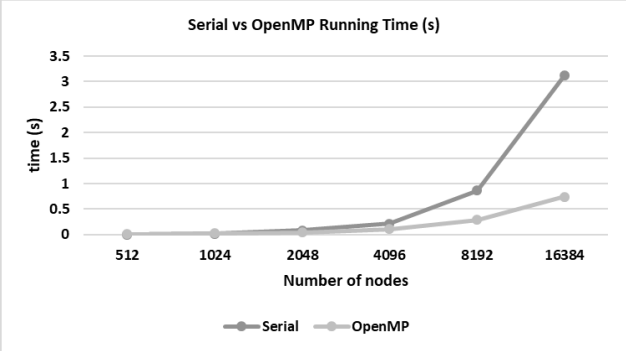

Fig. 7. Plot of Serial vs OpenMP execution time

In theory, the speed-up of OpenMP implementation should be the same as thread amount. Nevertheless, it does not come true in practice. For graph with fewer vertices (vertices <= 1024), the improvement is limited by the overhead, leading to longer execution time versus the one-thread implementation. Since the cost of creating threads in OpenMP is considerable, the speed-up cannot be identical to the number of threads in the experiment [6]. In addition, the input size to some extent limits the improvement of OpenMP Implementation. Better performance can be obtained if larger input size is applied to the experiment.

4.2. CUDA

The graph to test the performance is also created with matrix form as for OpenMP test. The matrix is stored at 2D like array. The values are randomly generated. The test input sizes are 512, 1024, 2048, 4096, 8192, and 16384. The GPU used for the testing is Tesla M2090 which has 512 cores and 6BG memory. To compile CUDA file, it uses CUDA 5.0, GCC 4.4.3, and Compute Capability 1.1. After compiling the test graphs with serial version and CUDA version of Dijkstra's Algorithm, the results are shown below in Table 2.

Num of Nodes	512	1024	2048	4096	8192	16384
Serial	0.01	0.03	0.09	0.4	1.58	5.81
CUDA	**0.005334**	**0.010972**	**0.023383**	**0.051648**	**0.119978**	**0.330931**
Blocks	2	2	4	8	16	32
Threads/block	256	512	512	512	512	512

Table 2. Serial vs CUDA execution time comparison

The performance of CUDA implementation became more promising as the input size increased. The highest speed-up for CUDA implementation was about 16.6 times faster than serial implementation. The plot in Fig. 8 generated from Table 2 makes the performance more obvious. It is more clear to see the speed-up gained from CUDA implementation.

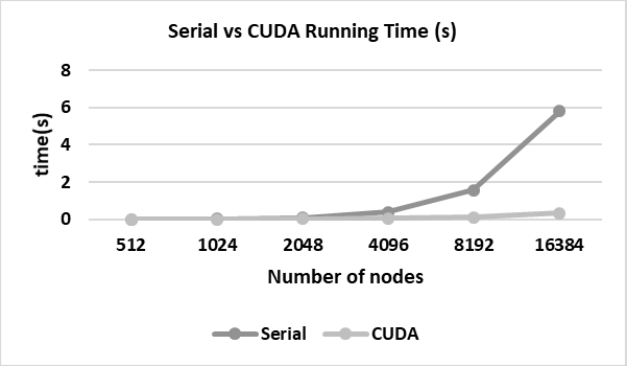

Fig. 8. Plot of Serial vs CUDA execution time

4.3. OpenMP vs CUDA

In theory, the running time complexity of Dijkstra's Algorithm with serial, OpenMP and CUDA implementation are $O(n^2)$, $O(n(n/p))$, and $O(n\log(n))$ respectively. With the graph put against each other for comparison, the performance of OpenMP and CUDA implementation behaved quite similarly as analysis. The highest improvement for OpenMP was 3.2 times faster than serial and CUDA was 16.6 times faster than serial. Therefore, it can predict that CUDA performs better than OpenMP for running Dijkstra's Algorithm.

Fig. 9 demonstrates the result of comparison between OpenMP and CUDA. The y-axis represents the speed-up of the parallel method compared with serial version. The negative value means that the paralleled implementation has worse performance than serial implementation.

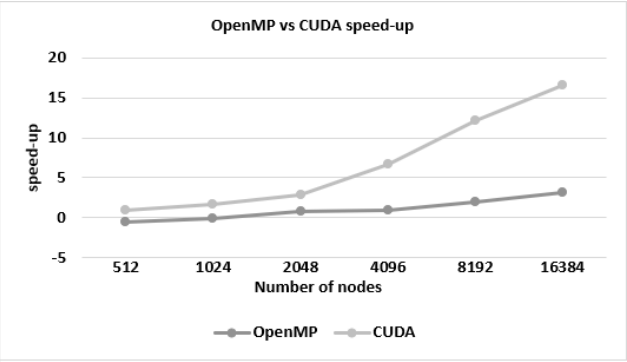

Fig. 9. Plot of OpenMP vs CUDA speed-up

5. CONCLUSTIONS

In our project, we successfully parallelized Dijkstra's Algorithm with OpenMP and CUDA. Both implementations have gained expected speed-up for the performance. For OpenMP, it achieved about 3.2 times faster than serial implementation. For CUDA, it achieved about 16.6 times faster than serial implementation. The results are quite consistent with our analysis.

Parallel implementation of Dijkstra's algorithm requires more work in the future to increase its efficiency. We plan to look for more optimization methods to address the Dijkstra's algorithm and obtain better performance. Many other platforms, like MPI and Python, are expected to support our parallel implementation. Some other algorithms to find the closest node and to update the distance are expected to be explored. Additionally, input is required to be improved in both size and source. We plan to test on much larger graphs and to seek for more real-life graphs as input.

6. REFERENCES

[1]https://www.myrouteonline.com/blog/what-is-the-best-shortest-path-algorithm

[2] Yulian Yu, Wenhong Wei, "The Shortest Path Parallel Algorithm on Single Source Weighted Multi-level Graph", *Second International Workshop on Computer Science and Engineering*, 2009.

[3] Pedro J. Martín, Roberto Torres, and Antonio Gavilanes, "CUDA Solutions for the SSSP Problem", *Computational Science*, pp 904-913, 2009

[4] N. Jasika, N. Alispahic, A. Elma, K. Ilvana, L. Elma and N. Nosovic, "Dijkstra's shortest path algorithm serial and parallel execution performance analysis," *2012 Proceedings of the 35th International Convention MIPRO*, Opatija, 2012, pp. 1811-1815.

[5] Alex Wong, "Parallelizing Dijkstra's Algorithm: OpenMP and CUDA", *Department of Electrical and Computing Engineering, Boston University of Engineering*, 2015.

[6] H. Cao, F. Wang, X. Fang, H. Tu and J. Shi, "OpenMP Parallel Optimal Path Algorithm and its Performance Analysis," *2009 WRI World Congress on Software Engineering*, Xiamen, 2009, pp. 61-66.