Parallelization of the Edmond Karp Algorithm for Maximum Flow on CUDA C Architecture

Sheheryar Amin
Computer Science
Habib University
Karachi, Pakistan

Oqba Jawed
Computer Science
Habib University
Karachi, Pakistan

Maha Usmani
Computer Science
Habib University
Karachi, Pakistan

***Abstract*—Edmond Karp is a widely used algorithm for finding the maximum flow in a flow network. However, as the size and complexity of flow networks increase, the demand for faster and more efficient computation methods also grows. This paper presents a parallel implementation of the Edmond Karp algorithm using CUDA C on NVIDIA GPUs.**

The aim of this research is to evaluate a parallel implementation of the Edmond Karp algorithm on GPU architecture. The proposed implementation utilizes the parallel processing power of GPUs to accelerate the algorithm, with a focus on parallelizing the major components of the algorithm, including the breadth-first search (BFS) algorithm, residual graph construction, and augmenting path search. The implementation also includes optimizations such as thread coarsening and shared memory usage to further enhance the performance.

We hope to provide an efficient and scalable solution for solving large-scale flow network problems. The proposed parallel implementation applied in various real-world applications, such as transportation networks, communication networks, and supply chain networks.

I. Introduction

A popular graph approach for determining the maximum flow in a flow network is the Edmonds-Karp algorithm. Jack Edmonds and Richard Karp initially presented this technique in 1972, and several scholars have subsequently examined and enhanced it. [3] The algorithm works by finding augmenting paths in the network continually and incrementing the flow sent, until there are no more such paths remaining.

The Edmonds-Karp method is utilised in a variety of applications, including computer networks, transportation networks, and resource allocation. However, when dealing with large-scale networks, the algorithm's speed might become a bottleneck. As a result, academics have been looking for ways to parallelize the algorithm in order to improve its performance.

As an illustration, consider Figure 1. It shows a graph of 7 nodes and 12 edges. A is the Source node and G is the Sink node. the values on the edges represent the flow (numerator) and the capacities (denominator). Initially, the flow is 0 for all edges.
Let's say we select the path $A \to D \to F \to G$ on the first iteration. The bottleneck capacity of this path is 3, since this is the minimum flow that can be sent through this path. After sending the flow, we increment the total flow by 3 and update the residual capacities of all edges. The updated capacities become:

1) (A,D) : 3/3
2) (D,F) : 3/6
3) (F,G) : 3/9

This step is repeated until no augmenting paths can be found.

The total flow sent through the graph is the sum of the flow sent through each augmenting path.

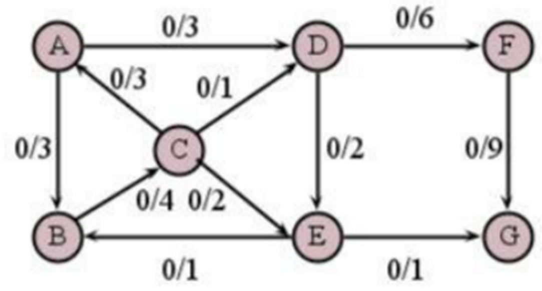


Fig. 1. Maximum Flow Problem

II. Previous Work (Literature Review)

Research on the Edmonds-Karp algorithm, has been widely conducted due to its practical applications in various fields, including computer networks, transportation networks, and resource allocation. Despite its efficiency, the algorithm's performance may become a bottleneck when applied to large-scale networks. Significant speed up can be achieved by implementing this algorithm on the GPU and fully utilizing the power of GPUs.

While significant work has been done on parallelizing the Ford Fulkerson method, which is the basis for the Edmonds-Karp algorithm, parallelization of Edmonds-Karp is an onging research topic. One area for improvement is its use of BFS to find the augmenting path with the minimum number of edges.

The most recent study was conducted by Chaibou etal. [1] in 2019, where they developed a parallel version of the Edmonds-Karp algorithm and implemented it using OpenMP for the CPU and CUDA for the GPU. Their results showed a significant improvement in performance on the GPU compared to OpenMP, demonstrating performance boost that was achieved due to the compute power of the GPU and the potential for further optimization of the algorithm through parallelization.

The parallelization approach of the algorithm is as follows;

1) The determination of the augmenting path from source to sink has to be done sequentially, however, the search for unvisited nodes can be done in parallel during BFS.
2) The calculation of bottleneck capacity of the augmenting path is trivially parallizeable: each block calculates the capacity of the edges assigned to it and then the global minimum is calculated using reduction
3) Updating the capacities of each edge can also be done in parallel, since each step independently calculates its residual capacity.

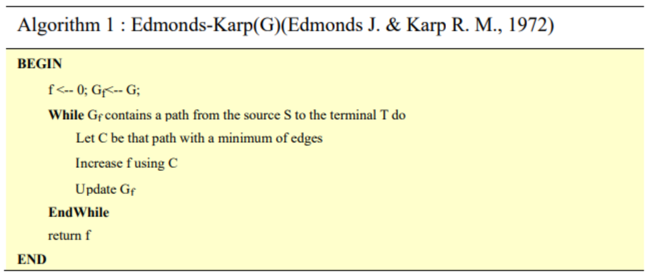
These three steps are repeated in a main loop until no augnenting path remains.

III. Methadology

This section describes the approaches taken for implementing the Edmonds-Karp algorithm on both CPU and GPU, including the implementation details

A. Edmonds Karp Algorithm

Edmaond Karp is an extension of Ford Fulkerson algorithm. The difference being that the latter finds the shortest augmenting path at each step. This is done using Breath First Search (BFS) which improves the search criteria by selecting the path containing minimum number of edges with a non-zero capacity. Edmond Karp Algorithm is shown in Figure 2. The sequential algorithm is implemented in C. We have used Breath First Search (BFS) to find the shortest path augmenting path at each iteration. The graph representation used is an adjacency matrix.



Algorithm 1 : Edmonds-Karp(G)(Edmonds J. & Karp R. M., 1972)

```
BEGIN
    f <-- 0; G_f <-- G;
    While G_f contains a path from the source S to the terminal T do
        Let C be that path with a minimum of edges
        Increase f using C
        Update G_f
    EndWhile
    return f
END
```

Fig. 2. Edmond Karp Algorithm

B. Working of the Algorithm

The working of the Algorithm contains 3 main steps:

Repeat until no more augmenting paths are remaining:

1) Search for an unexplored path from source S to sink T using BFS.
2) Calculate the bottleneck capacity of the path. This is the minimum flow that can be sent through this path
3) Update the capacities of all edges and increment the total flow.

The total flow through the network is the sum of flow through all augmenting paths.

C. GPU Implementation

We have taken the simple parallel approach algorithm from Sumit-Negi-github [6]. The approach involves three kernels `find_augmentingpath`,`kernel_update_residual_graph()`, `kernel_find_max_flow()`

The `parallel_edmonds_karp()` function initializes the graph, the residual graph, and memory allocations for frontier, visited, previous, path length, path, minimum flow, frontier empty, and result on CPU as well as GPU. It then copies the initial graph and residual graph to GPU. Then, it executes the iterative phase of the Edmond-Karp algorithm until there is no augmenting path remaining in the residual graph.

The first kernel, `kernel_find_augmenting_path()`, is executed in parallel to find an augmenting path from the source to the sink. The kernel takes as input the residual graph, frontier, visited, previous, and frontier empty, all stored on the GPU. The kernel checks the vertices in the frontier to identify their neighbors in the residual graph that have a positive residual capacity. It then checks whether those neighbors have been visited before, and if not, it updates the frontier, visited, previous, and frontier empty arrays on the GPU.

The second kernel, `kernel_update_residual_graph()`, is executed in parallel to update the residual graph based on the augmenting path. The kernel takes as input the residual graph, path length, path, and minimum flow, all stored on the GPU. The kernel checks the edges in the path and updates the residual capacity in the residual graph based on the minimum flow

The third kernel, `kernel_find_max_flow()`, is executed in parallel to find the maximum flow in the graph. The kernel takes as input the source, the initial graph, and the residual graph, all stored on the GPU. The kernel computes the maximum flow by adding up the difference between the flow and residual capacity along the edges connected to the source node. The `parallel_edmonds_karp()` function returns the final residual graph and the maximum flow computed by the `kernel_find_max_flow()` kernel.

D. Proposed Optimizations

1) The `kernel_find_augmenting_path()` can be optimized by using thread coarsening which will reduce the number of threads needed to execute the Kernel.
2) The `kernel_update_residual_graph()` can be optimized by incrementing indices using a stride value determined by the number of Blocks per grid. For each valid vertex the kernel uses atomics to update the residual graph.
3) The `kernel_find_max_flow()` can be optimized using shared memory to accumulate partial sums which are then added by a single thread, this will reduce the number of atomic operations needed to calculate the max flow.

IV. Implementation Details

The following Optimizations were applied to each of the kernels of the simple GPU implementation provided by Sumit-Negi-github [6]:

1) `kernel_find_augmenting_path`: The new implementation of the kernel using thread coarsening to improve performance by reducing the number of threads needed to execute the kernel. The previous kernel assigned one thread per vertex in the frontier set, which can result in a large number of threads for large graphs. This kernel, on the other hand assigns each thread to a group of vertices and iterate over them using a loop that skips over vertices not assigned to the thread. For each vertex in the group, the function checks if the vertex is in the frontier set. If not, it continues to the next vertex. If the vertex is in the frontier set, the function marks the vertex as processed, and then iterates over the outgoing edges of the vertex. For each outgoing edge with non-zero capacity, the function attempts to mark the destination vertex as visited using an atomic exchange operation. If the destination vertex was not previously visited, the function marks it as in the frontier, sets the previous array for that vertex, and sets the frontier empty flag to false.

```
__global__ void
    kernel_find_augmenting_path(int
    vertices, int *residual_graph, bool *
    frontier, int *visited, int *previous,
     bool *frontier_empty) {

    int id = blockIdx.x * blockDim.x +
    threadIdx.x;

    for (int i = id; i < vertices; i +=
    gridDim.x * blockDim.x) {


        if (!frontier[i]) continue;

        frontier[i] = false;

        for (int j = 0; j < vertices; j++)
     {


            if (residual_graph[i *
    vertices + j] == 0) continue;


            int visited_val = atomicExch(&
    visited[j], 1);


            if (visited_val == 0) {
                frontier[j] = true;
                previous[j] = i;
                *frontier_empty = false;
            }
        }
    }
}
```

Listing 1. Optimized `kernel_find_augmenting_path`

2) `kernel_update_residual_graph`: The new implementation of this kernel uses a for loop to iterate over the valid vertices in the augmenting path, starting at the current thread's index, tid, and incrementing by the stride value in each iteration. For each valid vertex in the augmenting path, the function uses atomicSub to subtract the minimum flow from the edge capacity in the residual graph from the destination vertex to the source vertex, given by residual-graph[u * vertices + v], and uses atomicAdd to add the minimum flow to the edge capacity in the residual graph from the source vertex to the destination vertex, given by residual-graph[v * vertices + u]. By incrementing the index by the stride value, the kernel distributes the work across multiple threads and ensures that each thread updates a unique set of edges in the residual graph, enabling efficient parallel execution.

```
__global__ void
    kernel_update_residual_graph(int
    vertices, int *residual_graph, int *
    path_length, int *path, int *min_flow)
    {

    int tid = blockIdx.x * blockDim.x +
    threadIdx.x;
    int stride = gridDim.x * blockDim.x;

    for(int id = tid; id < (*path_length)
    - 1; id += stride){

        int v = path[id];
        int u = path[id + 1];

        atomicSub(&residual_graph[u *
    vertices + v], *min_flow);
        atomicAdd(&residual_graph[v *
    vertices + u], *min_flow);

    }
}

```

Listing 2. Optimized `kernel_update_residual_graph`

3) `kernel_find_max_flow`: The new implementation of this kernel uses shared memory to accumulate partial results computed by each thread within a block, which are then added together by a single thread (thread with tid = 0) and added to the final result using an atomic operation. This approach reduces the number of atomic operations needed to compute the final result unlike the previous implementation that performed an atomic operation for each vertex that satisfied the condition in the if statement.

```
__global__ void kernel_find_max_flow(int vertices, int source, int *graph, int *residual_graph, int *result) {
    __shared__ int s_partial_results[1024];

    int tid = threadIdx.x;
    int gid = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    int partial_result = 0;
    for (int i = gid; i < vertices; i += stride) {
        if (graph[source * vertices + i] > 0) {
            partial_result += graph[source * vertices + i] - residual_graph[source * vertices + i];
        }
    }

    s_partial_results[tid] = partial_result;

    __syncthreads();

    if (tid == 0) {
        int block_result = 0;
        for (int i = 0; i < blockDim.x; i++) {
            block_result += s_partial_results[i];
        }
        atomicAdd(result, block_result);
    }
}

```

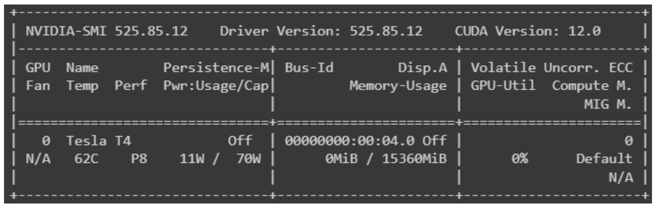
Listing 3. Optimized `kernel_find_max_flow`

V. Dataset

Implementations on the CPU and GPU were timed using synthetic data generated by SumitPadhiyar [5].This data is freely available on the respective repository on Github. For this data set the source vertex is the always 1 and the sink is always the number of vertices in the graph, for example, if the file has 50 vertices then the source is 1 and the sink is 50. Graphs with vertices 10, 35, 50, 100, 350, 500, 750, 1000, 2000, 3000, 4000, 5000 were used. Note that we had to limit to only 5000 vertices due to the transfer and stack limits on Github and Google Colab.

VI. Experimental Results

A. Characteristics of the GPU used



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.85.12    Driver Version: 525.85.12    CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   62C    P8    11W /  70W |      0MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

Fig. 3. Characteristics of the GPU

B. Results

In this section, we present the comparative performance analysis of Edmond Karp algorithm implemented on both the CPU and GPU, including the simple GPU implementation and the optimized GPU implementation presented in this paper. The results are shown in Table 1 and Figure 3 for graphs of up to 1000 vertices.

Number of Vertices	CPU Time	GPU TIme	GPU Optimized
10	104.8484	306.2789	248.1091
35	104.0432	223.1262	244.5267
50	103.8198	304.6937	244.6415
100	103.7803	306.2863	244.3310
350	105.5415	506.7089	404.8815
500	306.5567	707.9473	565.0724
750	906.6422	1115.8693	889.2796
1000	2012.2411	2020.5023	1609.9804

TABLE I
Comparision of CPU, GPU(optimized) and GPU(unoptimized) implementations

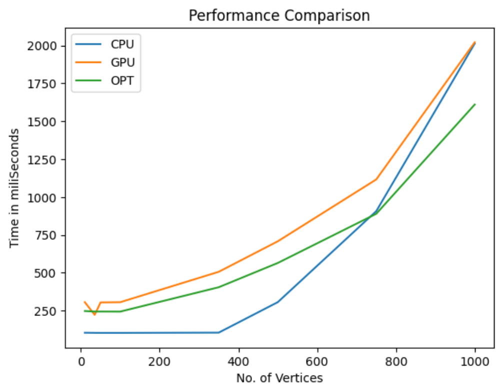


Fig. 4. Comparision of CPU, GPU (Optimized) and GPU (Un-optimized) Implementations

From Figure 3 it is evident that For smaller graphs, the CPU implementation outperforms both GPU implementations, primarily due to the overhead involved in copying data from GPU to CPU memory. However, for larger graphs with up to 5000 vertices, the optimized implementation provided in this paper performs the best. We were not able to run simulations on the CPU for graphs with more than 1000 vertices due to the stack limit on Google Colab but it is expected that both GPU implementations perform better than the CPU implementations as shown by Sumit-Negi-github [6] and evident from Figure 3 where the optimized implementation line out performs the the CPU implementation line.

To further investigate the performance of our optimized implementation, we compared with the simple GPU implementation with larger graphs. As seen from Table 2 and Figure 4, the optimized version yields a significant improvement over the unoptimized version.

Number of Vertices	**GPU TIme**	**GPU Optimized**
50	307.1249	280.2607
100	303.7362	254.3577
500	705.7769	645.2677
750	1114.8319	886.7344
1000	2209.8784	1611.0394
2000	11145.2038	8918.4008
3000	28329.6342	22150.3996
4000	64875.1276	50959.4429
5000	109449.0705	84565.1384

TABLE II
COMPARISON OF GPU AND GPU OPTIMIZED RUNTIMES FOR DIFFERENT NUMBER OF VERTICES

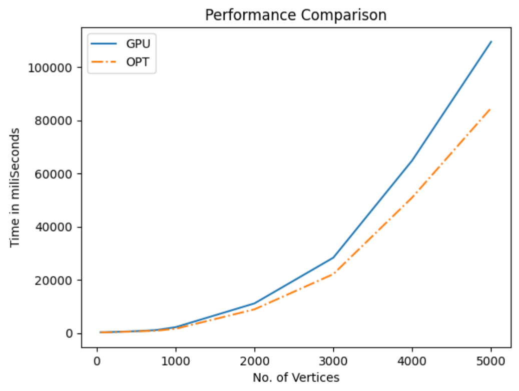


Fig. 5. Comparision of GPU Optimized vs Unoptimized Implementations

VII. CONCLUSION

In this paper, we studied the possibilities of Optimizing parallelization of the Edmonds-Karp algorithm and modeled the computation times of its sequential and parallel versions. The tests we have done on the parallel versions show that the optimizations suggested and implemented improve the calculation time of the algorithm compared to its sequential version. Furthermore, Overall, the results suggest that for larger graphs, the optimized GPU implementation is the most efficient implementation of the Edmond Karp algorithm. However, for smaller graphs, the CPU implementation may be a more suitable choice due to certain overheads.

REFERENCES

[1] Chaibou, A., Tessa, O.M., Sié, O. (2019). Modeling the Parallelization of the Edmonds-Karp Algorithm and Application. Computer and Information Science, 12(3), 81. https://doi.org/10.5539/cis.v12n3p81

[2] Jiang, Z., Hu, X., Gao, S. (2017). A Parallel Ford-Fulkerson Algorithm For Maximum Flow Problem. Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing, China. School of Mathematical Sciences, University of Chinese Academy of Sciences, Beijing, China.

[3] Edmonds, J., Karp, R. M. (1972). Theoretical improvements in algorithmic efficiency for network flow problems. Journal of the ACM (Association for Computing Machinery), 19(2), 248-264. https://doi.org/10.1145/321694.321699

[4] Ford, L, R., Fulkerson, D. R. (1962). Flows in Networks. Princeton University Press. https://doi.org/10.1063/1.3051024

[5] Padhiyar, S(2021). parallel_ford_fulkerson_gpu GitHub. https://github.com/SumitPadhiyar/parallel_ford_fulkerson_gpu

[6] Sumit-Negi-github. cs6023. GitHub, https://github.com/Sumit-Negi-github/cs6023/tree/92fda8173fbb7b41d8ca7cf15f73f255dc15b6ea