Pontificia Universidad Javeriana Cali

Electronic and Computer Science Department

Floyd Warshall

Parallel Programming

Authors:
Elkin Jadier Narvaez Paz
David Alejandro Sanchez Arias
Nicolas Alexander Lagos Seguel
Juan Sebastian Reyes

June 4, 2021

1 Problem specification & algorithm description

1.1 Problem description

- **Input:** A graph $G = (V, E)$ represented as an adjacency matrix. Let $W : V \times V \to \mathbb{Z}$ be a weight function such that $\forall_{(u,v)\in E} W(u, v)$ represents the weight of the edge (u, v). Additionally, $\forall_{(u,v)}(u, v) \notin E \to G_{u,v} = \infty$.
- **Output:** A matrix M of $n \times n$ such that $\forall_{1\leq i,j\leq n}, M_{i,j}$ represents the minimum distance between the edges i, j.
- **Description:** The algorithm allows to calculate the minimum distance between all the pair of vertices of the graph G. Also, this is done using the potential of the dynamic programming. This paradigm allows to use a brute force search memorizing previous computations, then, the algorithm calculates for each pair of vertices u, v the best route considering every middle point K and memorizing the previous computations in the matrix M.

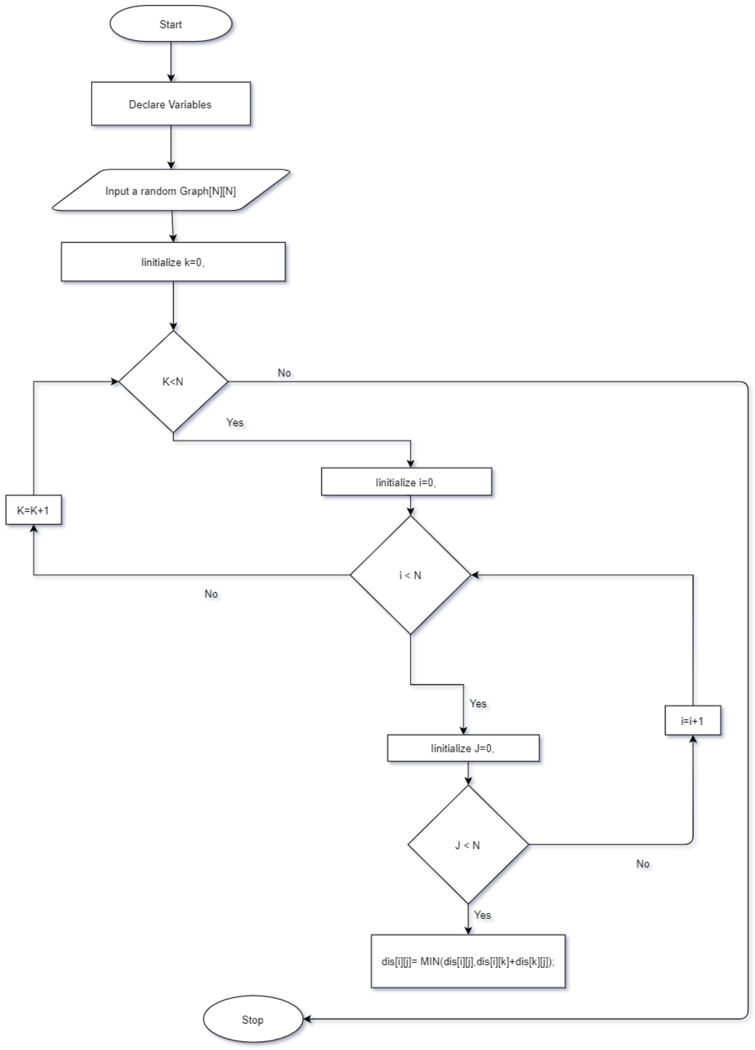
1.2 Computational complexity analysis

The dynamic programming implementation of the algorithm uses three for loops such that every one of them iterates over the integers from 1 to n. Then, the overall temporal complexity of the algorithm is $O(n^3)$. Also, as the algorithm uses a matrix for memorization and to generate the output, it has an overall spatial complexity of $O(n^2)$.

1.3 Data-flow

The image Figure 1 presents the flow chart of the algorithm.

Figure 1. Flow-chart for Floyd Warshall algorithm



1.4 Algorithm (pseudo-code)

Data: A graph $G = (V, E)$ represented as an adjacency matrix. Let $W : V \times V \rightarrow \mathbb{Z}$ be a weight function such that $\forall_{(u,v)\in E} W(u, v)$ represents the weight of the edge (u, v). Additionally, $\forall_{(u,v)} (u, v) \notin E \rightarrow G_{u,v} = \infty$.

Result: A matrix M of $n \times n$ such that $\forall_{1\leq i,j\leq n} M_{i,j}$ represents the minimum distance between the edges i, j.

```
Function FLOYD_WARSHALL($A, p, q$):
    $k \longleftarrow 0$;
    while $k < n$ do
        $i \longleftarrow 0$;
        while $i < n$ do
            $j \longleftarrow 0$;
            while $j < n$ do
                $dis[i][j] \longleftarrow MIN(dis[i][j], dis[i][k] + dis[k][j])$;
                $j \longleftarrow j + 1$;
            end
            $i \longleftarrow i + 1$;
        end
        $k \longleftarrow k + 1$;
    end
```

Algorithm 1: Floyd Warshall Algorithm

2 Running time

All the simulations were performed in a computer with the following features:

- **Processor:** Intel ® Core™ i5-9300H CPU @ 2.40GHz × 8
- **RAM memory:** 16 GB
- **OS system:** Ubuntu 20.04.2 LTS
- **Number of cores:** 4
- **Number of logical processors:** 8
- **L1 cache:** 128 KiB
- **L2 cache:** 1 MiB
- **L3 cache:** 8 MiB
- **SSD speed:** 550MB/s
- **GPU Engine Specs:** Geforce GTX 1650 4 GB GDDR5 @ 1485 MHz. 1024 CUDA Cores

2.1 Running time using Python built-in time function

The running time as the number of nodes in the graph increases is described in the Table 1 and shown in the Figure 2. In addition, Figure 3 shows the standard deviation of the running time for 5 executions.

Table 1. Running time as the number of nodes in the graph increases

N	Time (seconds)	Standard deviation
50	0.112484	0.000616
100	0.933496	0.00195
150	3.112119	0.02724
200	7.330021	0.06123
250	14.187668	0.16072
300	26.495729	0.25649
350	41.187141	0.10342
400	60.056369	0.41489
450	84.997181	0.47919
500	119.963587	0.678837

Figure 2. Running time as the number of nodes in the graph increases

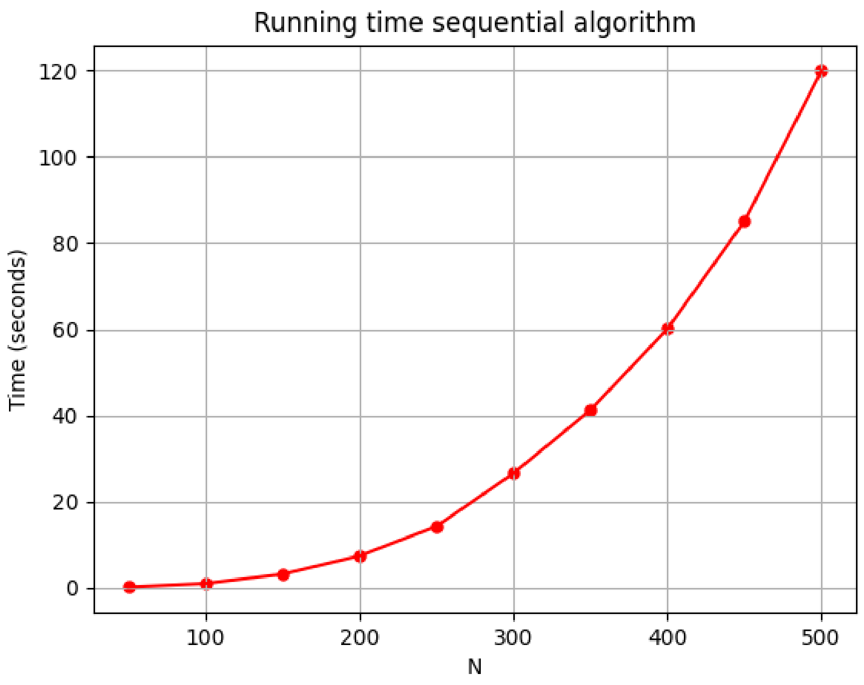
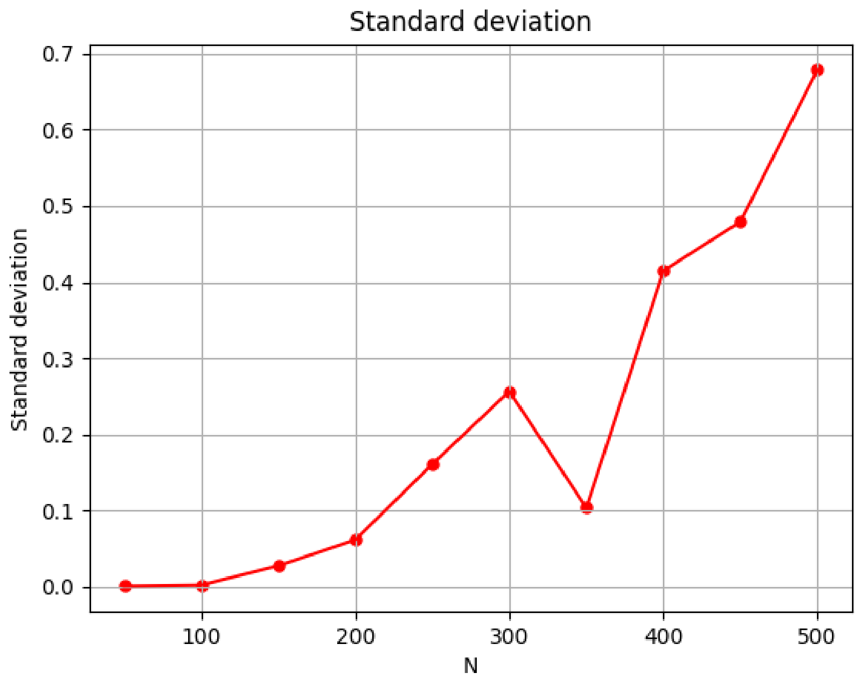


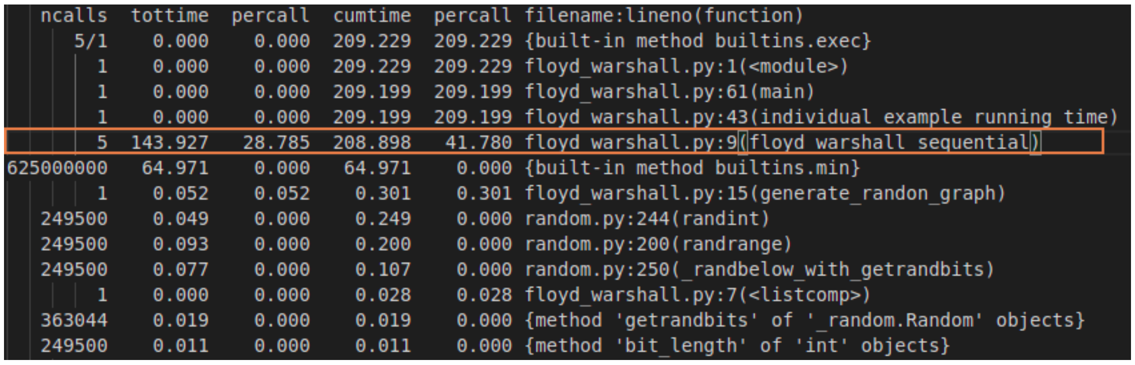
Figure 3. Standard deviation



2.2 Running time using cProfile

The complete running time analysis calculated with cProfile is shown at `https://raw.githubusercontent.com/elkinnarvaez/floyd-warshall/master/analysis.txt`. For this simulation, a set of 500 nodes was used, which is the maximum number of nodes that is supported during running time.

Figure 4. Profiling

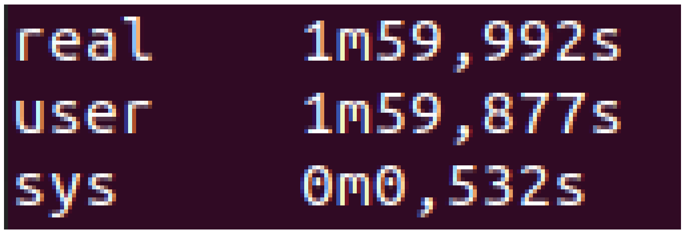


```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      5/1    0.000    0.000  209.229  209.229 {built-in method builtins.exec}
        1    0.000    0.000  209.229  209.229 floyd_warshall.py:1(<module>)
        1    0.000    0.000  209.199  209.199 floyd_warshall.py:61(main)
        1    0.000    0.000  209.199  209.199 floyd warshall.py:43(individual example running time)
        5  143.927   28.785  208.898   41.780 floyd warshall.py:9(floyd warshall sequential)
625000000   64.971    0.000   64.971    0.000 {built-in method builtins.min}
        1    0.052    0.052    0.301    0.301 floyd_warshall.py:15(generate_randon_graph)
   249500    0.049    0.000    0.249    0.000 random.py:244(randint)
   249500    0.093    0.000    0.200    0.000 random.py:200(randrange)
   249500    0.077    0.000    0.107    0.000 random.py:250(_randbelow_with_getrandbits)
        1    0.000    0.000    0.028    0.028 floyd_warshall.py:7(<listcomp>)
   363044    0.019    0.000    0.019    0.000 {method 'getrandbits' of '_random.Random' objects}
   249500    0.011    0.000    0.011    0.000 {method 'bit_length' of 'int' objects}
```

2.3 Running time using time

The running time that was measured using the time function is shown in the Figure 5. For this simulation, a set of 500 nodes was used.

Figure 5. Time function



```
real    1m59,992s
user    1m59,877s
sys     0m0,532s
```

3 Proposal for the parallel implementation

The parallel implementation will consist in the parallelization of the middle inner for. This one is in charge of computing the distance between each vertice of the graph. Notice that the middle inner for iterates over the rows and the most-inner one iterates over each one cell of that row, then we want to parallelize the computation of each one of these rows. For the parallel implementation we plan to take advantage of the GPU potential and the amount of cores it has by executing the same set of instructions in different execution threads.

4 Data dependency analysis

Let $u, v \in V$ two vertices of G with a path connecting them and k a third vertex such that a route $(u \to k) \wedge (k \to v)$ is considered. Notice that for a fixed K, the computation for each $i, j \in V$ is reproduced just once and each operation don't depends on annother for that fixed k, then, the estimation for values u, b is can be made in parallel.

The implementation of the algorithm can be found at `https://github.com/elkinnarvaez/floyd-warshall`.

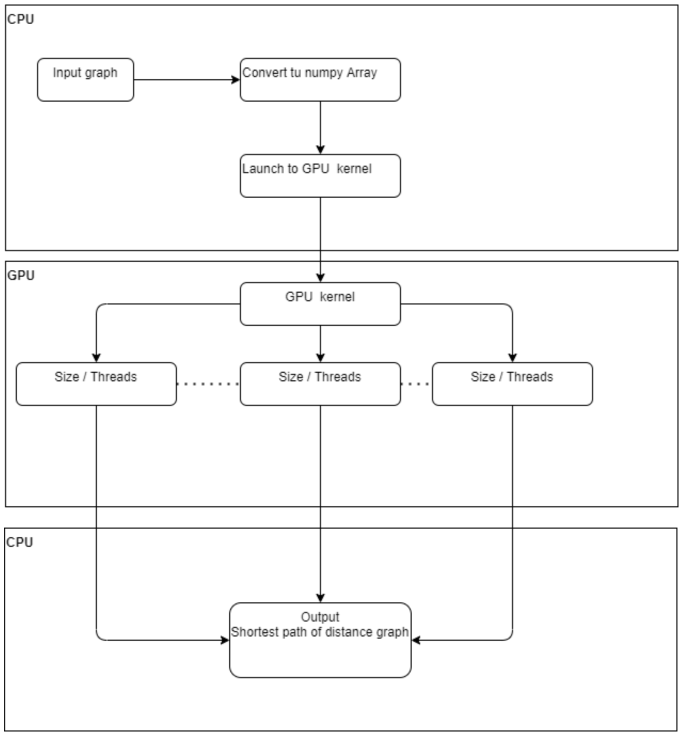
5 Parallel Algorithm Implementation

We implemented the parallel algorithm as follows:

First, looking at the data dependency analysis in the section 4, we noticed that the parallelization can be performed in a single block with n GPU threads, where each one of these will compute the task that is performed in each iteration of the middle-inner for.

The Figure 6 shows the diagram of the way the parallel implementation works.

Figure 6. Parallel implementation flow



In addition, the parallel implementation is shown below.

```
ker = SourceModule("""
    #include <stdio.h>
    #define MIN(x, y) (((x) < (y)) ? (x) : (y))
    __global__ void calculate_kernel(int *dis, int n, int k){
        int i = threadIdx.x;
        for(int j = 0; j < n; j++){
            dis[i * n + j] = MIN(dis[i * n + j], dis[i * n +
              k] + dis[k * n + j]);
        }
    }
    """)

def floyd_warshall_parallel(dis, n):
    for k in range(n):
        calculate_kernel(drv.InOut(dis), np.int32(n),
          np.int32(k), block = (n, 1, 1), grid = (1, 1, 1))
```

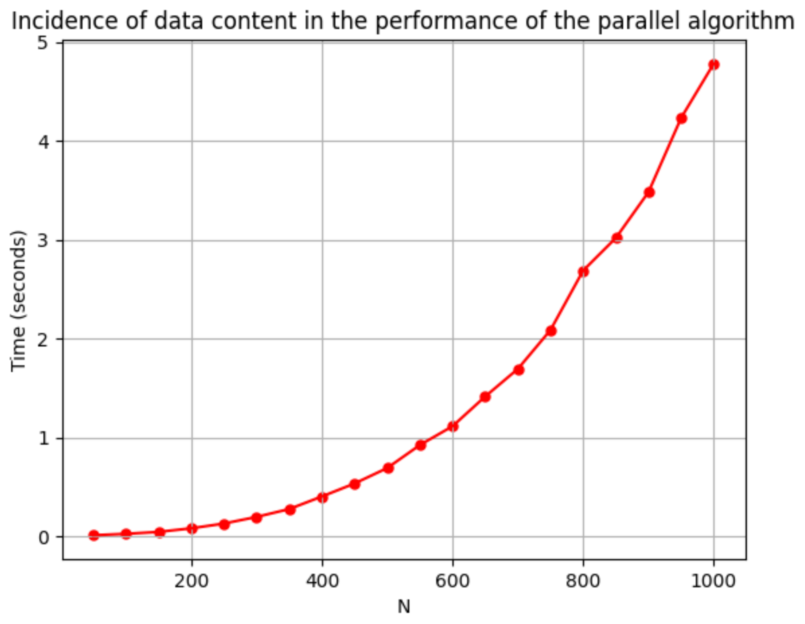
6 Incidence of data content in the performance

The Figure 2 shows the incidence of the data content in the performance of the sequential algorithm. On the other hand, the Figure 7 shows the incidence of the data content in the parallel implementation.

Table 2. Running time parallel implementation

N	Time (seconds)	Standard deviation
50	0.011032	0.000605
100	0.025692	0.000536
150	0.046174	0.000597
200	0.076588	0.000595
250	0.120385	0.002000
300	0.184551	0.002109
350	0.277795	0.001764
400	0.404381	0.002029
450	0.536774	0.005051
500	0.687838	0.002225

Figure 7. Running time parallel algorithm as the number of nodes in the graph increases



7 Performance analysis

7.1 Strong scaling analysis

The strong scaling analysis is shown in the Figure 8.

7.2 Isoefficiency

The isoefficiency analysis is shown in the Figure 8, since in this case the number of processors is equal to the number of nodes of the graph.

7.3 Running time vs data size (with different number of threads)

The Figure 7 shows the running time of the parallel implementation as the number of nodes in the graph increases.

7.4 Speed up vs data size (with different number of threads)

The speed up achieved with the parallel implementation is shown in the Figure 9.

Figure 8. Strong scaling

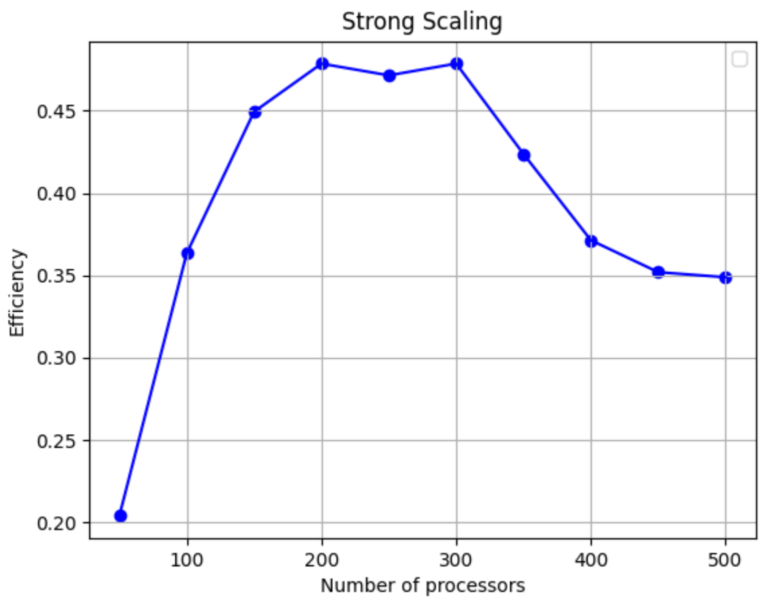
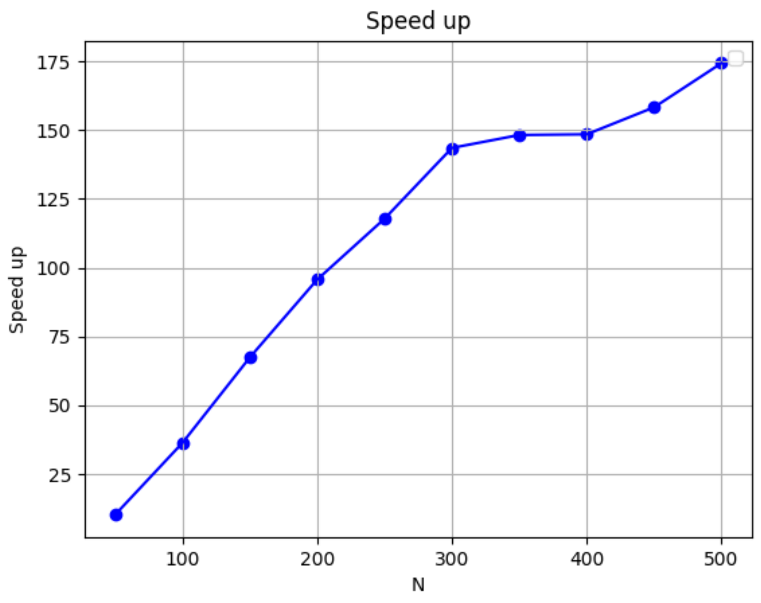


Figure 9. Speed up



7.5 VRAM usage vs data size (with different number of threads)

The VRAM usage was monitored with the following command: nvidia-smi –query-gpu=utilization.gpu –format=csv –loop=1

The Figure 10 shows the VRAM usage while the parallel program is running. In addition, the Figure 11 shows a screenshot of the process while the program is still running.

Figure 10. VRAM usage

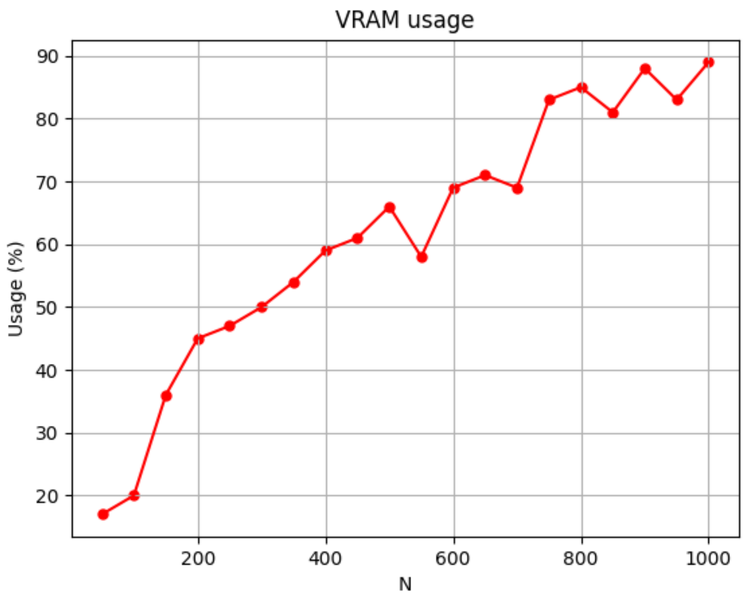
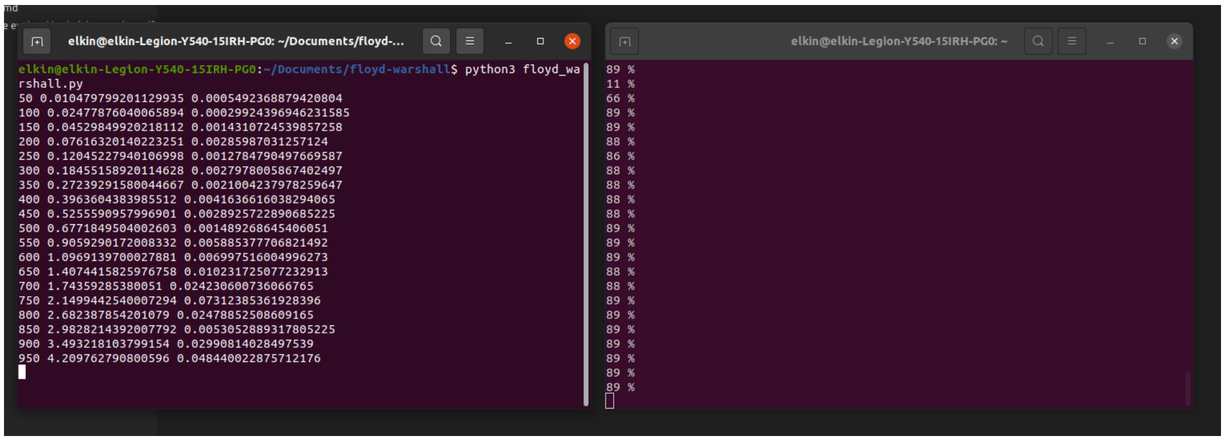


Figure 11. VRAM usage


elkin@elkin-Legion-Y540-15IRH-PG0: ~/Documents/floyd-...
elkin@elkin-Legion-Y540-15IRH-PG0: ~

```
elkin@elkin-Legion-Y540-15IRH-PG0:~/Documents/floyd-warshall$ python3 floyd_wa
rshall.py
50 0.010479799201129935 0.0005492368879420804
100 0.02477876040065894 0.00029924396946231585
150 0.04529849920218112 0.0014310724539857258
200 0.07616320140223251 0.00285987031257124
250 0.12045227940106998 0.0012784790497669587
300 0.18455158920114628 0.0027978005867402497
350 0.27239291580044667 0.0021004237978259647
400 0.3963604383985512 0.0041636616038294065
450 0.5255590957996901 0.0028925722890685225
500 0.6771849504002603 0.001489268645406051
550 0.9059290172008332 0.005885377706821492
600 1.0969139700027881 0.006997516004996273
650 1.4074415825976758 0.010231725077232913
700 1.74359285380051 0.024230600736066765
750 2.1499442540007294 0.07312385361928396
800 2.682387854201079 0.02478852508609165
850 2.9828214392007792 0.0053052889317805225
900 3.493218103799154 0.02990814028497539
950 4.209762790800596 0.048440022875712176
```

```
89 %
11 %
66 %
89 %
89 %
88 %
86 %
88 %
88 %
88 %
88 %
89 %
89 %
89 %
88 %
88 %
89 %
89 %
89 %
89 %
89 %
89 %
89 %
```

7.6 Running time vs number of threads

The Figure 7 shows the running time as the number of GPU threads (or nodes in the graph) increases.

7.7 Speed up vs number of threads

The Figure 9 shows the speed up of the parallel implementation as the number of GPU threads (or nodes in the graph) increases.

8 Problems found on the parallelization

During the parallel implementation of the algorithm using CUDA, we found the following problems and challenges:

- We had some issues during the installation of the cuda and pycuda tools. We also had some problems with the installation of the drivers that these tools require, and we finally realized that the error could be solved by installing a more recent Nvidia driver.
- The communication between the CPU and the GPU was a bit difficult to implement since it cost us to get the values of the graph to be maintained and not change.

9 Closest pair problem applications

The base problem Floyd Warshall algorithm solves is to find the shortest paths for each pair of nodes in a directed weighted graph with positive or negative edge weights. But there are several applications of this problem, some of them:

- Finding a regular expression denoting the regular language accepted by a finite automaton.
- Inversion of real matrices.
- Find the path of maximum flow in networking
- Fast computation of Pathfinder networks.