Force Directed Graph Visualizations on Graphics Processing Units using Java and Nvidia CUDA

Daniel Elias, B.S. Computer Science Student
Tecnológico de Monterrey, Campus Querétaro, México
A01208905@itesm.mx daniel.eliasbecerra98@gmail.com

Abstract

This paper explores the use of CUDA to visualize Force Directed Graphs on Java applications in $Big\ O(|V|)$ time.

I. Introduction

According to the World Economic Forum, by 2025 463 exabytes of data will be created each day globally. This is equivalent to saying that 212,765,957 DVDs will be created per day. [1] With this amount of data, new methods to visualize it are emerging to have a better understanding of data. Data visualization means drawing a human-readable representation of data, in order to clean it, explore the data's structure, detecting outliers and unusual groups, identifying trends and clusters, spotting local patterns, evaluating modeling output, and presenting results. [2] Specifically, graph visualization is concerned with representing WLAN networks, social interconnections or any system that has objects connected between them. This can lead to obtaining insights such as uncovering frauds or outliers in data, finding important groups or clusters, or even visualizing the relevance or importance of a person or website. [3]

II. Graph Visualization Algorithms

There are two main groups of algorithms that can be used for drawing general purpose graphs. The first one is the Spectral layout, where the node coordinates are extracted from the eigenvectors of a Lapacian matrix derived from the adjacency matrix of the graph. The second approach is the Force Directed layout, which is a gradient descent minimization of an energy function based on physical phenomena such as the repulsion or attraction of charged particles. For this project, the Force Directed approach was used. [4]

III. Forced Directed Graphs

In Force Directed layouts, graphs are represented as a system of particles that exert forces one each other. These algorithms attempt to converge to an equilibrium of the positions of the graph's vertices. For this project the Eades' 1984 algorithm is used to represent this system of attractive and repulsive forces given by the following equations [5]:

$$f_a = c_1 * log(d/c_2)$$

Equation 1.1: Logarithmic strength - Force of attraction

$$f_r = c_3/d^2$$

Equation 1.2: Inverse square law - Force of repulsion

Equations 1.1 and 1.2 use Hooke's Law springs logarithmic forces where d represents the length of the spring, or the distance between two vertices in the case of a graph. c_1,$c_2$and $c_3$are constants that an extensive use of the algorithm has show that work better with the values: $c_1 = 2,\ c_2 = 1$and $c_3 = 1$. This is the algorithm:

Algorithm 1: SPRING (G:graph)

```
1  place vertices of G in random locations;
2  repeat M times
3      For each vertex of G:
4          For each vertex adjacent to the vertex
5              calculate f_a of the adjacent vertex
6              move the vertex c_4 * f_a
7          For each vertex non-adjacent to the vertex:
8              calculate f_r of the non-adjacent vertex
9              move the vertex c_4 * f_r
10 draw G on graphical user interface
```

Algorithm 1: Iterative Eades' Force Directed Graph Visualization

In the algorithm, the recommended value for $c_4 = 0.1$. Most graphs achieve the minimal energy state or equilibrium after $M = 100$ iterations.

IV. The problem

Big O Time complexity and Rendering

The time complexity for running this algorithm is $O(|V|^2 + |E|)$ where $|V|$ is the number of vertices and E the number of edges. [5] As the Big O complexity states, if the number of vertices increases, the time taken to complete the algorithm will increase following a quadratic time function. Thus, the new positions of each vertex will take more time to be calculated as more vertices are in the graph. This has a direct impact on the rendering time when the graph is visualized in a graphical user interface. The time it takes for the application to redraw the new positions of the graph's vertices increases as $|V|$ also becomes larger, resulting in a delay in rendering each frame, which is easily perceived by the human eye. Therefore, the main problem to be solved is: how could the time of the algorithm be improved in order to maintain a high performance rendering in the application's graphical user interface?

V. The solution

Parallel programming in the GPU

In order to improve the performance of the SPRING algorithm the proposed solution makes use of the parallel paradigm. This is the resulting algorithm:

Algorithm 2: CUDA_SPRING (G:graph)

```
1  place vertices of G in random locations;
2  repeat M times
3      Assign G vertex to a thread in the GPU
4      For each of the other vertices of G:
5        If vertex adjacent to the thread's vertex:
6              calculate f_a of the adjacent vertex:
7              move the thread's vertex c_4 * f_a
8      If vertex non-adjacent to the thread's vertex:
9              calculate f_r of the non-adjacent vertex:
10             move the thread's vertex c_4 * f_r
11 draw G on graphical user interface
```

Algorithm 2: Parallel Eades' Force Directed Graph Visualization

Algorithm 2 parallelizes the calculation of *fa*, *fr* and the movement of each vertex with respect to every other vertex of the graph. This is done in parallel by a unique thread assigned to each vertex in a linearized adjacency matrix:

$$index = threadIdx.x + blockIdx.x * blockDim.x$$

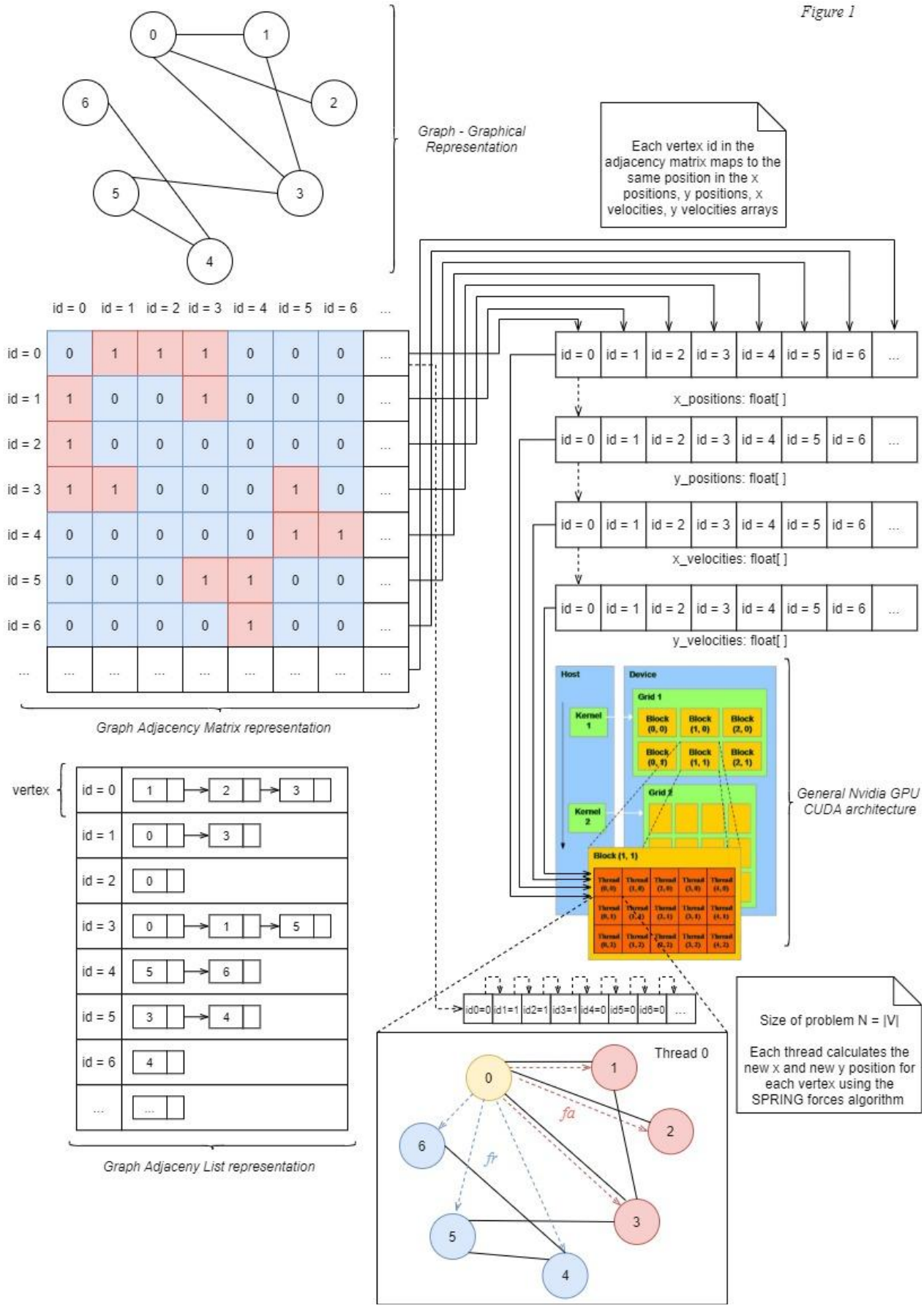
Calculation 1.1:index in the adjacency matrix for each vertex

This results in reducing the Big O time complexity to $O(|V|)$ as each thread only traverses its corresponding row of size $|V|$ in the linear adjacency matrix. Moreover, because of the 1024 threads per block limitation on Nvidia GPUs, thread reutilization is used:

$$while\ index < |V| and\ index * |V| + |V| <= |V|^2$$
$$CUDA\_SPRING()$$
$$index = index + blockDim.x * gridDim.x;$$

Calculation 1.2:thread reutilization with new index calculation

Figure 1


Graph - Graphical Representation
Each vertex id in the adjacency matrix maps to the same position in the x positions, y positions, x velocities, y velocities arrays
x_positions: float[ ]
y_positions: float[ ]
x_velocities: float[ ]
y_velocities: float[ ]
Graph Adjacency Matrix representation
vertex
Graph Adjaceny List representation
Host
Device
Grid 1
Grid 2
Kernel 1
Kernel 2
General Nvidia GPU CUDA architecture
Thread 0
fa
fr
Size of problem N = |V|
Each thread calculates the new x and new y position for each vertex using the SPRING forces algorithm

On the left side of Figure 1, three different representations for a graph are displayed: the graphical representation, the adjacency matrix and the adjacency list data structures. It is worth noticing that this paper only explores undirected graphs as shown on the adjacency matrix. Number 1s in this adjacency matrix represent edges that go from row index to column index, these indices are the same as the ids of each of the graph's vertices. In the right side of *Figure 1* there are 4 arrays: *x_positions* and *y_positions* arrays, each one of size $|V|$, have the data of each of the vertices positions in the x and y components; *x_velocities* and *y_velocities*, each one of size $|V|$, have the data of each of the vertices velocities in the x and y components. These velocities result from applying the *fa* and *fr* forces on the vertices and dictate the motion of the vertices in a two-dimensional space.

As shown in Figure 1, each of the rows of the adjacency matrix, which represent a vertex, are mapped by index to its corresponding index in the *x_positions, y_positions, x_velocities, y_velocities* arrays (e.g., vertex 0 has its x and y positions and velocities data on index 0 of the corresponding arrays).

A representation of the general architecture of an Nvidia GPU is portrayed below the arrays in *Figure 1*. As explained earlier, each thread of each block in the GPU is mapped to one vertex using *Calculation 1.1* (in case of thread reutilization, each thread may have more than one vertex assigned using *Calculation 1.2*). Each thread then calculates *CUDA_SPRING* for its corresponding vertex. (e.g., Thread (0,0) calculates *CUDA_SPRING* for vertex 0 using the calculated index 0 to access the vertex's row in the adjacency matrix and the x,y positions and velocities in the corresponding arrays.)

Finally, for each thread to calculate vertex's *fr*, *fa* and movement, the thread traverses the vertex's row in the linear adjacency matrix that starts at *position = index * vertices.* When it finds a 1 that represents an edge, the *fa* is calculated, if a 0 is found the *fr* is computed. These forces then cause the vertex to have a new velocity and position in x and y and move in the graphical user interface after every iteration of *M.*

VI. Technical implementation

Java and JCuda

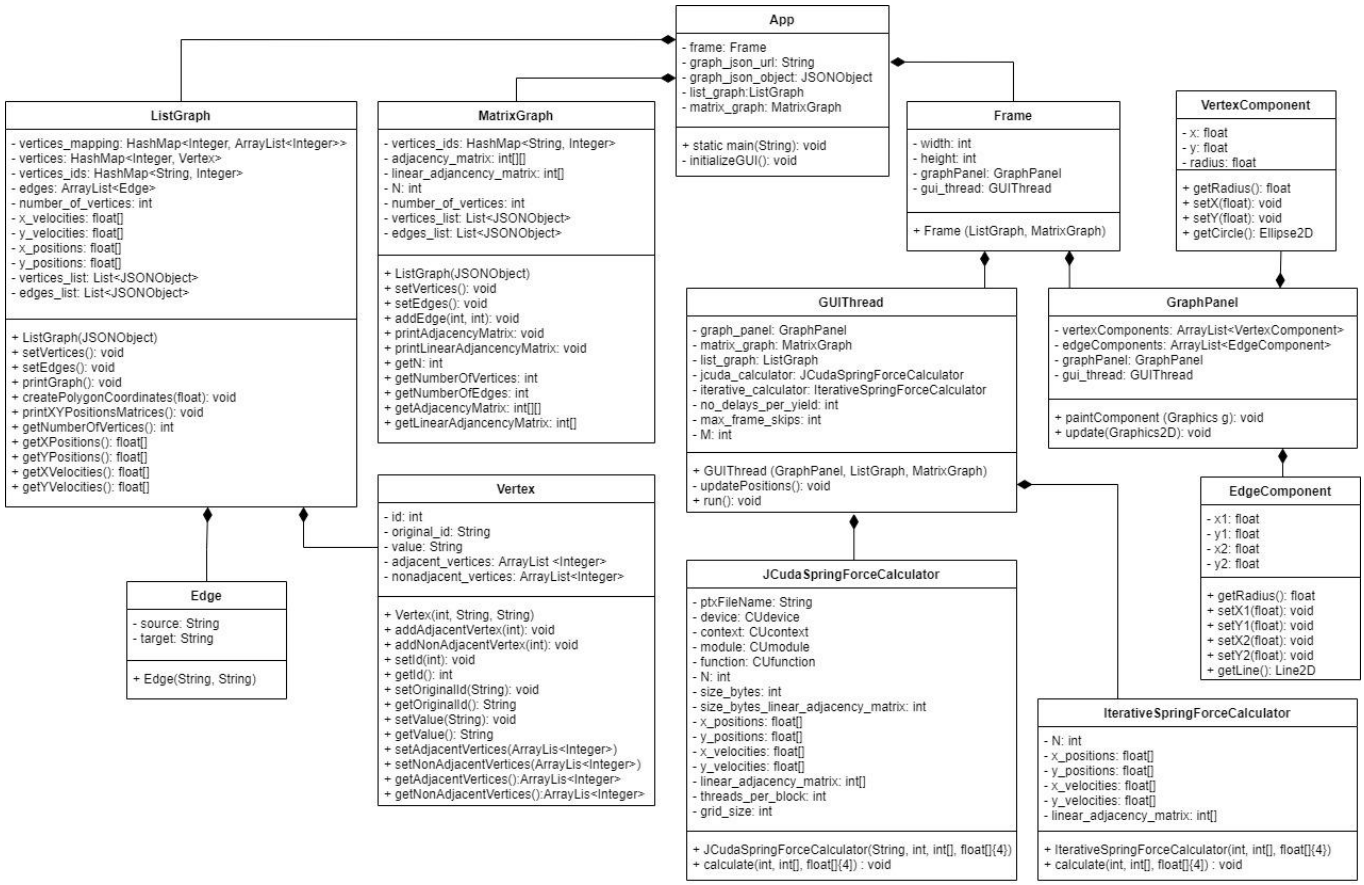
The main purpose of *CUDA_SPRING* is to reduce the Big O time from $O(|V|^2 + |E|)$ to $O(|V|)$. An application in Java was developed to test and measure the iterative and parallel approaches. The app was implemented using the *JCuda* dependency to easily incorporate CUDA, which kernels are commonly written in C or C++. Moreover, the *javax.swing* and *java.awt* libraries were used to create a Graphical User Interface in its own Thread inherited class. *Figure 2* shows the class diagram for the application. Class ListGraph is the Adjacency List and MatrixGraph is the Adjacency Matrix representation. App class runs the application and loads the graph data which is stored in a JSON file with the following format:

```
{
    "vertices": [{"id":0}, {"id": 1}, ...],
    "edges": [{"source": 0, "target": 1}, ...]
}
```

Graph in JSON Format

Frame and GraphPanel are in charge of displaying and drawing the VertexComponents and EdgeComponents 2D Graphics on a Graphical User Interface. Most importantly, GUIThread is the class that redraws the graph on the GraphPanel after every iteration of *M*. This GUIThread class has a reference to the adjacency matrix, x_positions, y_positions, x_velocities and y_velocities. GUIThread also creates the JCudaSpringForceCalculator and IterativeSpringForceCalculator objects which implement *CUDA_SPRING* and *SPRING* respectively.

Figure 2: UML Class Diagram for the Force Directed Graph Visualization Application



JCudaSpringForceCalculator is the class which makes use of the JCuda dependency. JCuda works similar to the CUDA programs made in C and C++, the difference is that in Java you have a different syntax to allocate memory or obtain pointers to arrays or variables. Another difference is that the *__global__* and *__device__* functions of the CUDA Kernel .cu file should still be written in C in a separate .cu file. JCuda then compiles it using the Nvidia Cuda Compiler (NVCC) with the JCuda Driver API to create a .ptx file used by the Java program. [7]

The project's repository

The project's repository can be found at: https://github.com/DanElias/Kurve2D**.** For more information on how to run the project refer to the *Appendix Section 1*. The kernel used by the JCudaSpringForceCalculator can be found in the file JCudaSpringForceCalculatorKernel.cu found in: src/main/java/com/kurve/kurve2d/CudaKernels/

VII. Testing and experimentation

Hypothesis

H_1: The parallelization on the Eade's Force Directed Algorithm (SPRING) reduces the Big O complexity from from $O(|V|^2 + |E|)$ to $O(|V|)$.

Variables

Our independent variables would be $|V|$and M. The dependent variable would be the time t in milliseconds it takes for the CUDA_SPRING and SPRING algorithms to finish after M iterations.

Datasets

Four different graphs in JSON format were used. Refer to *Table 1*.

Graph in JSON file	$\|V\|$
miserables.json	77
random.json	581
blocks.json	1238
random2.json	2000

Table 1: Graphs used and their corresponding $|V|$

Experiment 1

The experiment consists of measuring for each of the datasets the time t_1 it takes for the iterative *SPRING* algorithm to finish after M iterations, and then measuring the time $t_2$it takes for the parallel *CUDA_SPRING* algorithm to finish after *M* iterations. This will be repeated for *M* = {100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200}

This time measurement is done in the run() method in the *GUIThread* class, which repaints the *GraphPanel* with the updated graph positions after every iteration of *M*.

Experiment 1 results per dataset

For miserables.json *Figure 3.1* shows both iterative and parallel approaches maintain a similar linear increase not displaying any advantage of CUDA_SPRING over SPRING.

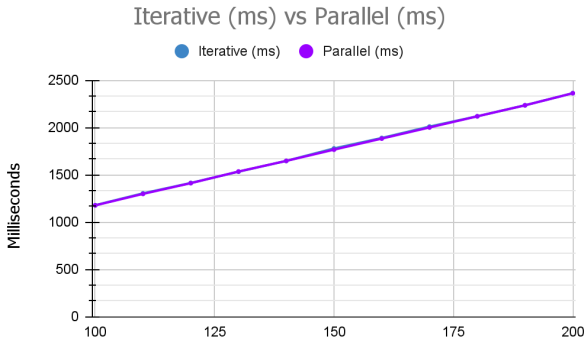


Figure 3.1: Iterative vs parallel miserables.json

For random.json *Figure 3.2* shows both iterative and parallel approaches maintain a similar linear increase not displaying any advantage of CUDA_SPRING over SPRING.

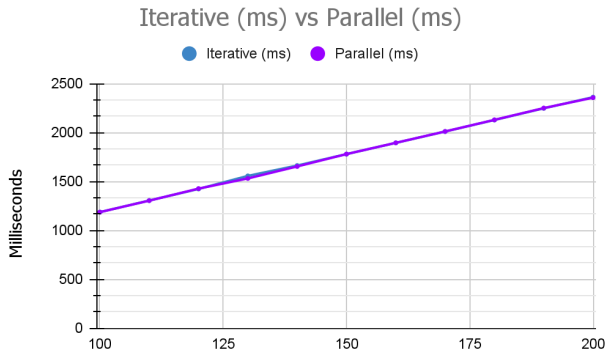


Figure 3.2: Iterative vs parallel random.json

For blocks.json *Figure 3.3* shows a visible advantage of the parallel approach over the iterative. The difference in time between both approaches was obtained, *Figure 3.4* shows a linear increase in it as *M* increases.

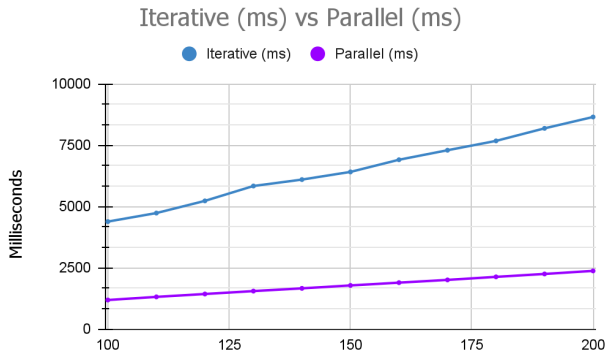


Figure 3.3: Iterative vs parallel blocks.json

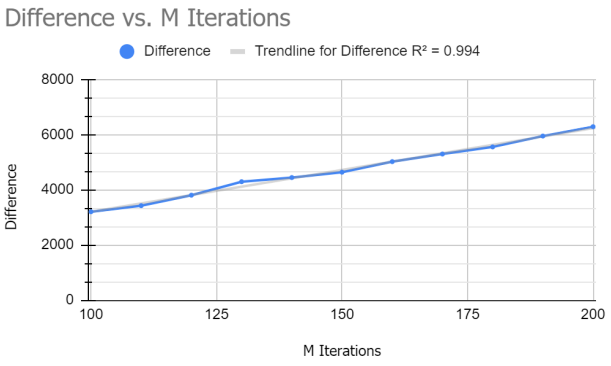


Figure 3.4: Difference (ms) blocks.json

For random2.json *Figure 3.5* shows a more visible advantage of the parallel approach over the iterative. The difference in time between both approaches was obtained, *Figure 3.6* shows a linear increase in it as *M* increases.

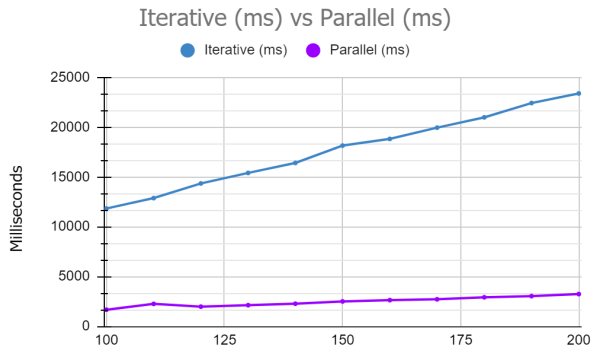


Figure 3.5: Difference (ms) random2.json

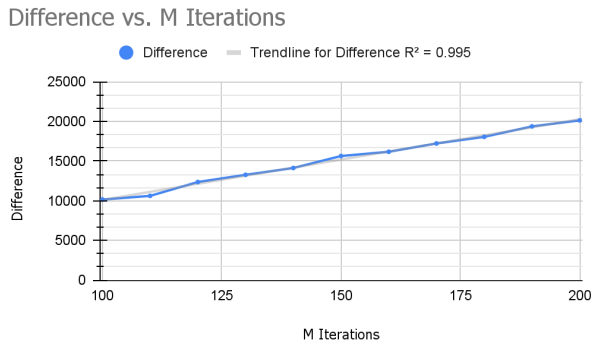


Figure 3.6: Difference (ms) random2.json

Experiment 2

The experiment consists of measuring 5 times for each of the datasets the time t_1 it takes for the iterative *SPRING* algorithm to finish after M iterations, and then measuring 5 times the time $t_2$it takes for the parallel *CUDA_SPRING* algorithm to finish after *M* iterations. This is now done only for *M* = 100. The 5 measurements made for each approach for each dataset are averaged to obtain a more consistent time measurement.

Again, the time measurement is done in the run() method in the *GUIThread* class, which repaints the *GraphPanel* with the updated graph positions after every iteration of *M.*

Experiment 2 results - Final results

Table 2 shows the final results and *Figure 3.7* both the iterative and parallel approaches graphs.

$\|V\|$	**Iterative M=100 (ms)**	**Parallel M=100 (ms)**
77	1117.77	1177.19
581	1223.79	1186.48
1238	4787.85	1187.08
2000	12039.27	1655.73

Table 2: Time taken by each dataset to complete M =100 iterations in the iterative and parallel approaches

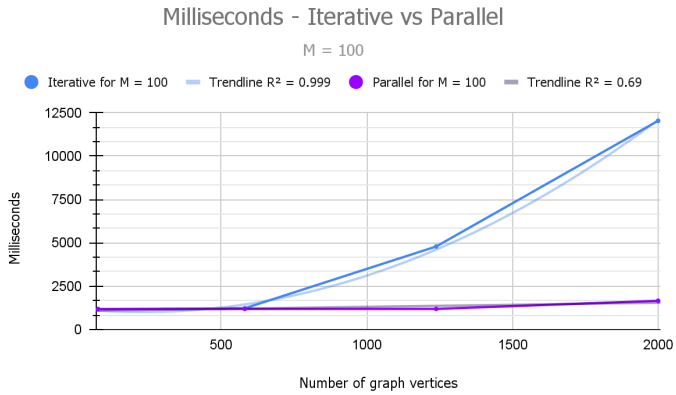


Figure 3.7: Time taken by each dataset to complete M =100 iterations in the iterative and parallel approaches

Figure 3.7 finally displays the time difference between the iterative and parallel approaches. The trendline for the iterative approach is a degree 2 polynomial with $R^2 = 0.999$ which is with accordance to the $O(|V|^2 + |E|)$ Big O time, while the parallel approach has a linear trendline with $R^2 = 0.69$, which is with accordance with $O(|V|$) Big O time. We can conclude H_1 to be true. This last linear trendline might need more datasets to be able to fit a correct trendline with R^2nearer to 1 to be even more sure of $H_1$to be true.

VIII. Conclusion

This project demonstrates the advantage of using Parallelized Force Directed Graph Algorithms with tools such as CUDA to improve their time, which in turn leads to better rendering performance in the GUI that benefits and aids the user in visualizing large amounts of interconnected data in graphs in fast and usable apps. The usage of Java and JCuda also demonstrates the possibilities of parallelization in other languages apart from C and C++ and how easy it is to incorporate the parallel paradigm in applications developed in Java.

IX. Appendix

Section 1: Project Setup

Requirements

CUDA version:

https://developer.nvidia.com/cuda-10.1-download-archive-update2

Windows 10 x86_64 10.1.2

Netbeans 12.0 LTS Windows x64

https://netbeans.apache.org/download/nb120/nb120.html

To see the project source code and run it you will need to use Netbeans 12.0 LTS for Windows x64 with Java 15, Java SE, Java EE, HTML, JavaScript and the Maven dependency manager.

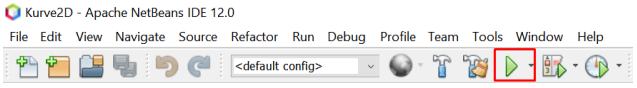
JCuda 10.1.0 dependencies:

Already included in the pom.xml file

You will need to have Maven enabled

Running the Project

Only click on the play icon on the top bar



Changing the data source

You can change the graph data that is visualized by changing the path shown in line 47 of the App.java file:

```
public static void main(String args[]) throws IOException{
    String url = "src/main/java/com/kurve/kurve2d/data_examples/blocks.json";
    App app = new App(url);
}
```

Section 2: JCuda Resources

Installation

You can follow this tutorial to install CUDA on Windows 10 and assure it was properly installed:

https://www.youtube.com/watch?v=cL05xtTocmY&ab_channel=NVIDIADeveloper

JCuda Tutorial:

http://www.jcuda.org/tutorial/TutorialIndex.html

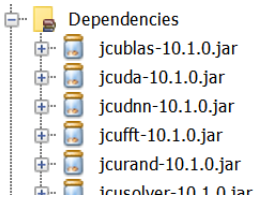
http://www.jcuda.org/

Setting up JCuda in a new project:

After setting up the project with Maven, add to the pom.xml file the JCuda dependencies that can be found at:

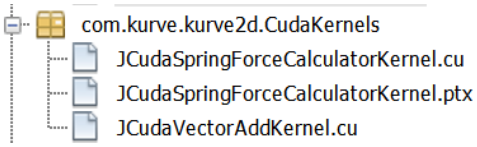
http://www.jcuda.org/downloads/downloads.html

Then build the project and the dependencies should be installed. They appear as .jar files in the Dependencies folder:

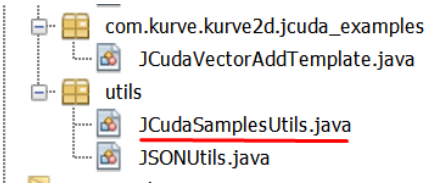


To create your own Kernels:

com.kurve.kurve2d.CudaKernels folder will store the .cu files for the device's code. .ptx files are just the compiled version of the .cu files after building the project.

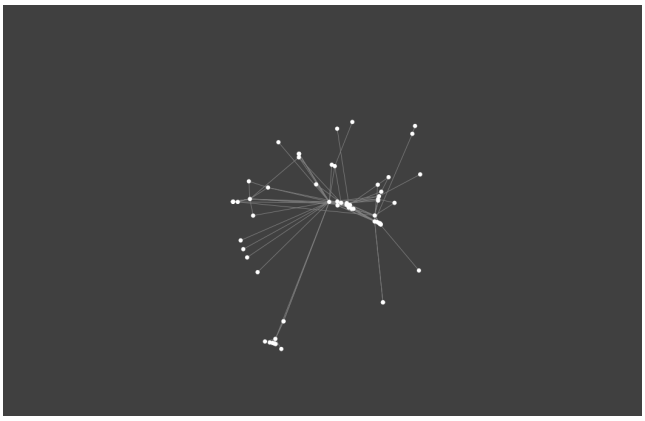


JCudaSamplesUtils.java has the tools to use .cu files. It is located in the utils package:



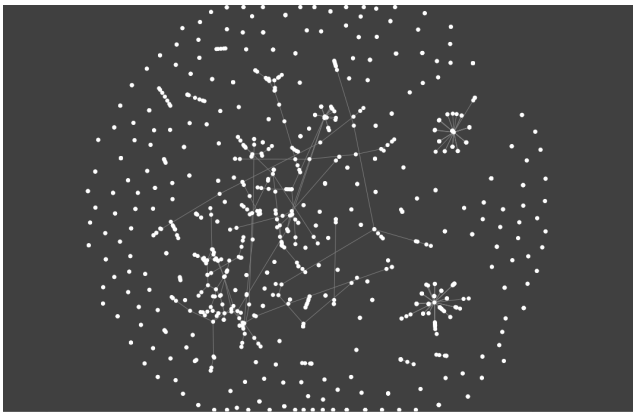
Section 3
Generated visualizations

miserables.json



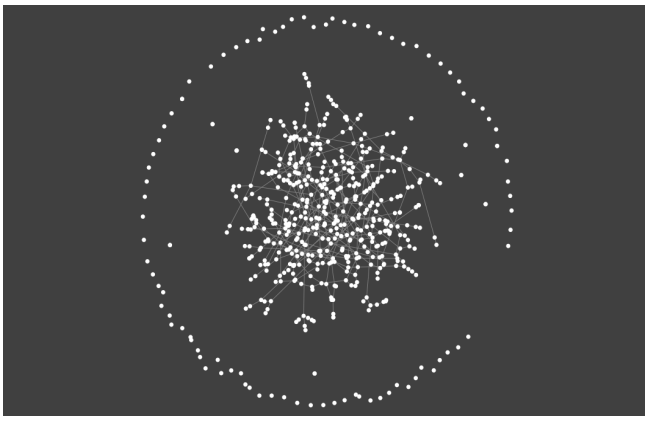
miserables.json represents the connections between the characters in the famous musical.

blocks.json

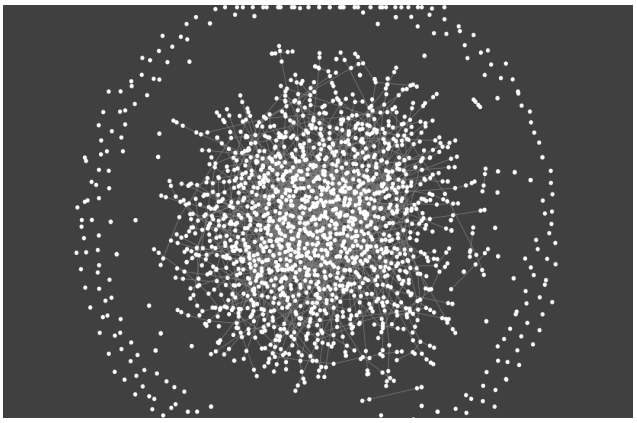


blocks.json is a real dataset of addresses, in the visualization 2 clusters can be seen on the left side of the image.

random.json



random2.json



X. References

[1] https://www.weforum.org/agenda/2019/04/how-much-data-is-generated-each-day-cf4bddf29f/

[2] https://hdsr.mitpress.mit.edu/pub/zok97i7p/release/3

[3] https://neo4j.com/blog/understanding-graphs-and-graph-data-science/

[4] https://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/2009/PHD/PHD-2009-02.pdf

[5] http://cs.brown.edu/people/rtamassi/gdhandbook/chapters/force-directed.pdf

[6] GPU Architecture Image: https://www.researchgate.net/figure/CUDA-Architecture_fig1_221142943

[7] http://www.jcuda.org