Bachelor Informatica

Universiteit van Amsterdam

Accelerating Fruchterman-Reingold with OpenCL

Gerrit Krijnen

June 20, 2014

Supervisor(s): Robert Belleman (University of Amsterdam)

Signed:

Abstract

Twilight, an interactive multi-touch graph visualization research tool jointly developed by the University of Amsterdam and SURFsara, uses the Fruchterman-Reingold algorithm (amongst others) to create meaningful layouts for graphs. The required time for this algorithm to complete scales exponentially with the size of the graphs, therefore rendering large graphs quickly becomes impractical. To combat this an investigation was conducted into parallelizing this algorithm for use on Graphical Processing Units. This thesis describes the design and implementation of an OpenCL based implementation of the Fruchterman-Reingold algorithm. We show that this implementation speeds up the rendering of graphs by a factor of up to 15 times compared to the original implementation within Twilight.

Contents

CHAPTER 1

Introduction

Twilight is an interactive multi-touch graph visualization research tool that has been developed by the University of Amsterdam (UvA) and SURFsara. It is used by researchers who want to detect patterns and structures in large graphs. For instance, Twilight has played a central role in the research of HIV-1 transmission[13].

The Fruchterman-Reingold (FR) algorithm[7] is the most popular of several algorithms by which layouts for graphs are generated within Twilight. The right part of Figure 1.1 shows an example layout generated with this algorithm. In this case the dataset of the before mentioned HIV-1 transmission was used, which consists of 1471 nodes and 7954 edges. The resulting layout shows that the graph is clearly divided into three groups split by means of transmission.

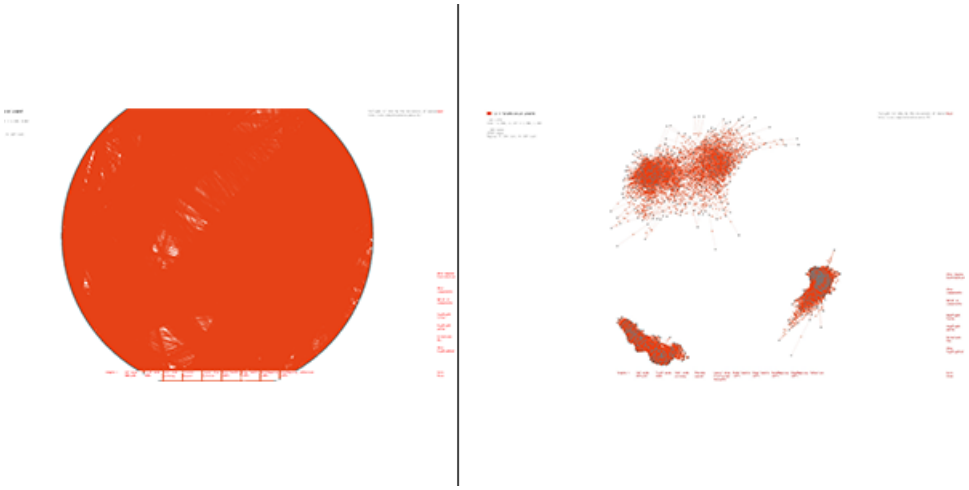

Figure 1.1: Comparison of two layouts of the same graph using Twilight. The left picture shows the default circular layout and the right picture shows the layout that was produced by the FR algorithm. While the circle layout shows little to no structure, the layout computed by FR clearly shows three distinct clusters.

The time to compute a graph layout with FR is $O(|V|^2)$ where V is the number of nodes/vertexes in a graph. Considering this complexity, the computation time for large graphs quickly becomes impractical. For example, the largest graph that was distributed with Twilight has 13993 vertices and 39869 edges and takes upwards of 20 minutes to generate a layout on a laptop with an Intel® Core™ i7-3537U processor running at 2GHz.

For this reason a quicker implementation of the algorithm has been requested by the developers and users of Twilight. The specific question that is raised is whether it is possible to

leverage the computing capabilities of the Graphical Processing Unit (GPU) that is present in virtually every modern computer.

1.1 Related Work

Previous research into parallelizing force-directed graph algorithms, such as FR, has been conducted by the creators of AllegroLayout[1]. They have created an extension for Cytoscape[10], which is an open-source, publicly available, free alternative to Twilight. However the extension itself is neither open-source nor free. Nevertheless, their website does boast about an impressive speedup of about 160 times compared to the original version on an overclocked GeForce GTX 560 Ti GPU, which is, at the time of writing, a 2 year old budget gaming GPU with 384 CUDA cores. This promises an even better improvement when using newer hardware like the latest flagship GPU from NVIDIA™, the GeForce GTX 780 Ti, which has 2880 CUDA cores. AllegroLayout's creations have however released no explanation on how their parallel implementation achieved this speedup, so it is possible that corners were cut to increase speedup at the cost of accuracy compared to the sequential algorithm.

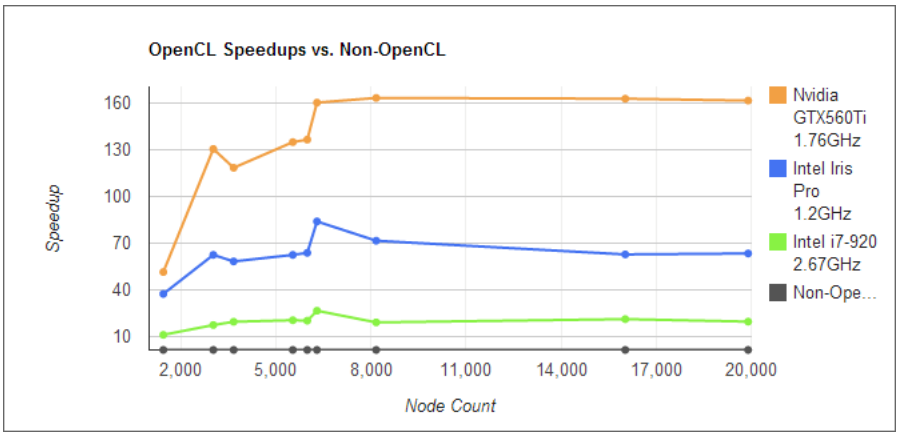

Figure 1.2: Comparison of speedup created by AllegroLayout's OpenCL implementation of a Force-Directed Graph layout creation algorithm when compared to their original sequential Force-Directed Graph layout creation algorithm. Source : `http://allegroviva.com/allegrolayout2/opencl-acceleration`

Philip Bjorge[2] has made a blog post about a personal attempt to implement a 3D variant of the FR algorithm in CUDA. While his attempt has failed to deliver a correctly working implementation of the algorithm, a good lesson on how to tackle this problem has been shared on his blog, namely that a sequential solution of the problem should be implemented before attempting to parallelize the algorithm.

BioLayout 3D[5] has completed an adaptation of FR in OpenCL and measurements show that it offers a speedup of up to 60 times[11] when compared to the old sequential FR algorithm within BioLayout 3D. The program is open-source software and freely available under the GNU GPL license.

[1] `http://allegroviva.com/allegrolayout2/`

[2] `http://philipbjorge.com/archived_wp_blog/www.philipbjorge.com/2011/11/11/3d-fruchterman-reingold/index.html`

1.2 Research Questions

The main question that is answered in this research is :

"Can the execution time of the Fruchterman-Reingold algorithm be accelerated by parallelizing it for use on Graphical Processing Units?"

A subquestion has been defined so that a possible indirect gain of parallelizing the FR algorithm, increasing the maximum size of graphs that can effectively be researched, can be identified.

"Does a parallel implementation of the Fruchterman-Reingold algorithm help to facilitate research on larger graphs?"

This thesis is organized in the following way: chapter 2 defines the design considerations for this project. Chapter 3 covers the details of the actual implementation and optimizations that were made to the algorithm. Subsequently, in chapter 4 the results of those optimizations will be benchmarked and compared to the original sequential CPU implementation of the FR algorithm. Finally the conclusions and recommendations for further research will be discussed in chapter 5.

CHAPTER 2

Design Considerations

This chapter describes the design considerations that were analyzed in the process of creating this version of the algorithm.

The research question that was formulated in the previous chapter is answered by first analyzing how the FR algorithm works in detail. After that possible options for parallelizing algorithms are discussed and compared.

2.1 Fruchterman-Reingold

The FR algorithm[7] is a force-directed graph drawing algorithm. Which means that a physics based model is used to control the position of the nodes. Figure 2.1 shows the original pseudocode from the paper in which this algorithm was first detailed. The pseudocode shows that there are three distinct steps within the main loop. The three steps are :

1. Calculate Repulsion Force between nodes
2. Calculate Attraction Force between nodes
3. Check distance of movement and save movement to layout

These steps are repeated until the layout converges to an equilibrium.

To calculate the repulsive force for a node, the distance of that node to all the other nodes in the system is calculated. Each node is modeled like a charged particle that exerts a repulsive force on all others relative to their distance. The smaller the distance, the bigger the force becomes, as illustrated in figure 2.2.

The second step, calculating the attractive forces, is iterated per edge in the graph. The two nodes are then moved a certain distance that corresponds with the length of the edge. This means that strongly connected nodes bunch together. Figure 2.3 shows the calculation of the attractive forces for node 1. The calculated force is then added to the aforementioned repulsive force.

The final step of the algorithm limits the euclidean length of the resulting vectors to a predefined maximum. This maximum reduces in every iteration step so that the changes in position decrease over time. All the position changes are now saved to the layout. Figure 2.4 shows a visual representation of this calculation.

```
area  := W * L; { W and L are the width and length of the frame }
G := (V, E); { the vertices are assigned random initial positions }
k  := sqrt(area/|V|);
function fa(z) := begin return x²/k end;
function fr(z) := begin return k²/z end;

for i := 1 to iterations do begin
    { calculate repulsive forces}
    for v in V do begin
        { each vertex has two vectors: .pos and .disp }
        v.disp := 0;
        for u in V do
            if (u # v) then begin
                { Δ is short hand for the difference}
                { vector between the positions of the two vertices )
                Δ := v.pos - u.pos;
                v.disp := v.disp + ( Δ /| Δ |) * fr(| Δ |)
            end
    end

    { calculate attractive forces }
    for e in E do begin
        { each edge is an ordered pair of vertices .v and .u }
        Δ := e.v.pos - e.u.pos
        e.v.disp := e.v.disp - ( Δ/| Δ |) * fa(| Δ |);
        e.u. disp := e.u.disp + ( Δ /| Δ |) * fa(| Δ |)
    end

    { limit the maximum displacement to the temperature t }
    { and then prevent from being displaced outside frame}
    for v in V do begin
        v.pos := v.pos + ( v. disp/ |v.disp|) * min ( v.disp, t );
        v.pos.x := min(W/2, max(-W/2, v.pos.x));
        v.pos.y := min(L/2, max(–L/2, v.pos.y))
    end
    { reduce the temperature as the layout approaches a better configuration }
    t := cool(t)
end
```

Figure 2.1: Original pseudocode of the FR algorithm[7].

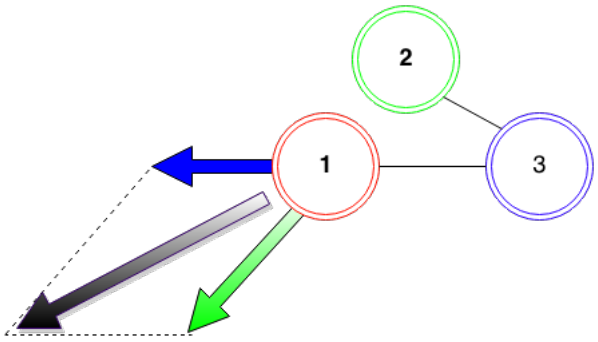

Figure 2.2: Calculation of the repulsive force exerted on node 1 is based on the sum of repulsive forces exerted on node 1 by node 2 and 3. Note that edges between nodes play no role in this part of the algorithm. The blue and green vector are added to each other and the resulting vector is the vector with the black gradient.

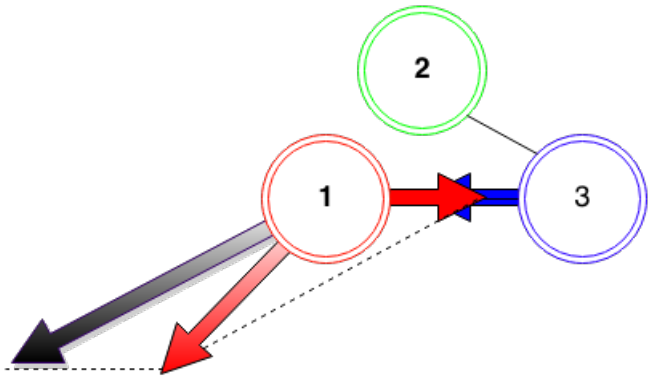

Figure 2.3: Calculation of the attractive force exerted on node 1 is based on the distance to nodes that are connected to node 1 through an edge. The solid red and blue vectors represent the attractive force that is exerted through the edge. The black gradient vector is the vector calculated in figure 2.2 and the red gradient vector represents the total displacement vector, i.e the summation of the black gradient vector and the solid red vector.

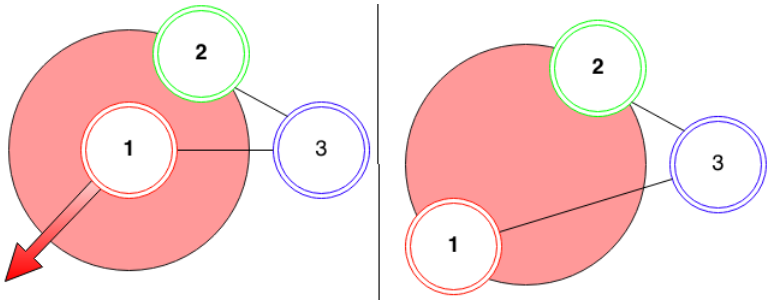

Figure 2.4: The displacement of node 1 is limited by a maximum, represented by the red circle. Left: situation before displacement of node 1. Right: situation after displacement of node 1.

The algorithm consists of $|V|^2 + |E| + |V|$ operations, which results in a time complexity of $O(|V|^2)$ where $|V|$ is the number of vertices and $|E|$ represents the number of edges. This quadratic complexity is caused by the repulsion step. This step needs to iterate over each node and for that node calculate the distance to all the other nodes.

2.1.1 Options for parallelization

There are quite a few options available to run parallelized code on GPU's. A few of the popular options include Nvidia's CUDA, OpenCL and OpenMP. OpenMP only recently added the possibility for GPU computation, so support and documentation for this feature is lacking compared to the others and was therefore dropped as an option. A comparison between the other two options was made based on four characteristics :

1. Portability: Twilight has to work on Linux, Windows and MacOS. The brand of GPU within that machine needs to have no influence on whether Twilight can run or not. OpenCL is a clear winner here, since OpenCL is supported by all major GPU vendors, while CUDA is NVIDIA exclusive.

2. Performance: The running time of the algorithm needs to be shorter than the current implementation. Both CUDA and OpenCL can provide a quicker implementation than the sequential implementation. Research conducted by the Delft University of Technology[6] suggests that CUDA and OpenCL are very close performance wise, so both implementations tie in this category.

3. Debugging support: The algorithm needs to be stable and thus be free bugs. The main threat to stability for this program is leakage of memory. Because the memory is not managed by the CPU, memory leaks are relatively hard to discover. CUDA has an excellent plugin[1] for IDE's that provides debugging and profiling capabilities of parallelized code. OpenCL has no such native debugging support, but external tools such as gDEBugger[2] can be used to provide similar functionality. CUDA wins here because of the native support for debugging.

4. Compilation procedure: We want the compilation procedure for the parallel version of FR within Twilight to change as little as possible when compared to the original implementation. CUDA requires an additional compiler to compile the parallel kernels whereas OpenCL kernels are compiled by the drivers at runtime. Therefore OpenCL is a clear winner in this category, because all kernel compilation is provided by the already present GPU drivers.

Table 2.1: Comparison of characteristics of CUDA and OpenCL.

	CUDA	OpenCL
Portability	$--$	$++$
Performance	$++$	$++$
Debugging	$++$	$-$
Compilation Procedure	$-$	$++$

With all the above taken into account, the choice was made to develop the parallel version of FR on the OpenCL platform. The main reason for doing so is that OpenCL provides superior portability when compared to CUDA. This means that the algorithm can be used on a wider variety of computers, whereas with CUDA all non NVIDIA devices would have to fall back on the sequential algorithm.

[1] `http://www.nvidia.com/object/nsight.html`
[2] `http://www.gremedy.com/`

CHAPTER 3

Implementation

3.1 Parallelization using OpenCL

To fully understand the parallelization options for the FR algorithm, we need to understand how parallelization can be done using OpenCL. OpenCL depends on kernels for the execution of code. A kernel is a small piece of code that can be run in parallel. All kernels have two vital arguments in common:

1. The number of jobs to start.
2. A list of other kernels to wait for.

The number of jobs determines how much data is processed. And the list of other kernels to wait for can be used to create complex code flows on the GPU.

An OpenCL program is composed of one or more of these kernels. All these kernels are compiled by the OpenCL compiler at runtime. These compiled kernels are then uploaded to a specific device, which can be either a GPU or CPU. After this OpenCL has to reserve the memory that is used by the application up front. When this is done, initial data can be uploaded to the reserved memory after which the kernels can be started.

3.2 Parallelization

An algorithm can be parallelized using one or both of the following options :

1. Task parallelism.
2. Data parallelism.

Task parallelism means that several different tasks do not depends on each other for data at all and therefore can be executed in any order and at the same time. An example for the FR algorithm would be the repulsion and attraction steps of the algorithm. It does not matter in which order the repulsion and attraction are calculated and they do not rely on each other for their input, since they both take the original layout as input.

Data parallelism means that the same instruction has to be executed many times with different data. This generally means that loops are executed in parallel instead of sequentially. In the FR algorithm all steps use loops and therefore the main method to improve performance of the algorithm is by applying data parallelism.

A combination of these two different types of parallelism should be used to produce an algorithm that manages to keep GPU utilization high.

For the repulsion step each node could be seen as different data and a kernel for this calculation could sum the distances to all the other nodes in the graph. This implementation still has a loop in which all the distances are summed, and therefore still executes the same instruction over multiple different data points. Because of this a more efficient way to parallelize this algorithm is by seeing each distance calculation as a kernel. This kernel is then used to calculate all the distances between the nodes and another kernel is used to sum the distances applicable to a specific node.

In the attraction step each edge can be seen as a different data point. The resulting attraction vectors for each edge can be calculated in parallel.

In the distance limitation step of the algorithm every node can be seen as a different data point. A kernel calculates the displacement vector and compares it with the maximum allowed length ,resizes the displacement vector if needed and then saves the new location of the node to the layout.

All of these can be combined in order to provide a parallel implementation of the FR algorithm. Step 1 and step 2 are task parallel and are therefore executed at the same time. Step 3 depends on the data from step 1 and step 2 and therefore has to wait until both steps are finished. Within these steps data parallelization is applied to speed up the core of each step. In doing so the aforementioned GPU utilization is kept high and therefore the performance gain will be maximized.

3.2.1 Repulsion Step

The repulsion step is the most computationally complex step of the algorithm and therefore the one where parallelization has the most influence. As discussed in the previous paragraph the most obvious way of parallelizing this part of the algorithm is by parallelizing each distance calculation. The drawback of this method is that the result of every calculation has to be saved in memory. For 25,000 nodes there are $\frac{25000^2 - 25000}{2}$ calculations to be performed and stored. This is because the distance between two nodes only has to be calculated once and the distance of a node to itself does not have to be calculated at all. Assuming that each distance is stored in a 64 bit float (4 bytes) 1192.05 megabytes of memory would be required for 25,000 nodes. Due to the quadratic nature of the memory usage the maximum capacity of the currently available GPU's would quickly be exceeded as the graphs get larger in size. Currently a typical high-end GPU available on the market with a price tag of around 300 euro has 4 GB of memory. This means that a maximum of around 46,000 nodes could be processed on such a card (disregarding memory usage by other parts of the algorithm). Figure 3.1 shows the memory usage of this part of the algorithm versus the number of nodes in a graph.

Because this approach severely limits the maximum graph size that can be calculated with this implementation of FR a second approach has been devised. This approach breaks the problem up per node instead of per calculation. For each node all of the distances to the other nodes are calculated in parallel and stored in a shared array. Afterwards another kernel is run to sum all the distances in this array and save it to the general repulsion array. This process is then repeated for every node in the graph. This means that $|V| - 1$ values need to be stored, where V is the number of nodes in the graph. Which for 25,000 nodes would amount to 97KB of memory usage. While the memory usage might improve vastly with this approach, the number of calculations that have to be performed are doubled in this approach. The previous approach saved all the distances in an array, so if the distance from node x to node y was calculated this could be referenced by node y when trying to determine the distance to node x. Whereas in this approach only the final distances are saved and any intermediate results have to recalculated.

The initial choice was made to calculate the memory usage of the algorithm when the algo-

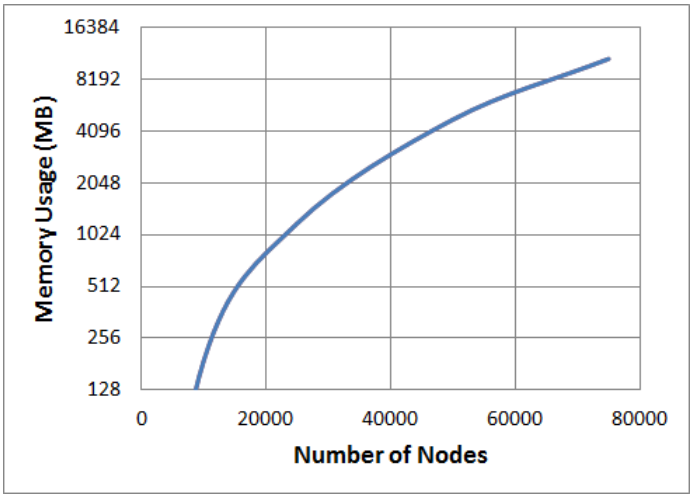

Figure 3.1: Memory usage versus number of nodes for the first variant of the parallel repulsion calculation.

rithm is first run and use the first variant where possible and only if the algorithm would not fit in memory to resort to the second variant. This variant can also run in parallel so multiple instances could be started for different nodes. This trick has been created to keep GPU utilization high and lessen the impact of the extra calculations. However early tests showed that the extra calculations and overhead caused a large drop in performance and as many as 50 of these instances running in parallel were needed to get the performance back to acceptable levels. Having this many instances running next to each other caused the algorithm to crash with OpenCL error code -9999 , which unfortunately is an undocumented error. Therefore this part of the algorithm was scrapped and only the first variant of the repulsive step is used in the actual implementation.

3.2.2 Attraction Step

The second part of the algorithm, the attraction step, is calculated by starting a work item for every edge there is in the graph. Each edge has the index of the two nodes it connects, those indexes are then used to retrieve the current positions for both nodes. These positions are converted into a distance, and the distance is then used to calculate an attractive force. The attractive force increases as the distance between the two nodes gets larger, this causes all the nodes that are connected to bunch up. The repulsive force calculated in step 1, which gets larger as the nodes get closer, also makes sure that this step does not pull all connected nodes into a single "black hole" but instead makes sure that there is always some distance between nodes. The calculated force then gets added to the index of the nodes in a global array.

Atomic float addition is used to prevent race conditions from accidentally discarding values. Atomic float addition is not a standard function in OpenCL but instead relies on a trick that was first documented by Igor Suhorukov[1]. This trick uses the available atomic integer compare and swap function which swaps the two values only if the original value did not change before the swap completed and takes the integer value that represents the said float in order to trick OpenCL into thinking that it is actually dealing with integers.

3.2.3 Movement limitation step

The final part of the algorithm, the normalization of the results, is calculated by starting a work item for every node in the graph. This kernel is only started after all other work has completed, because it depends on both the attraction and repulsion kernel results. This kernel takes the

[1] http://suhorukov.blogspot.com/2011/12/opencl-11-atomic-operations-on-floating.html

results for the node that it is working on from the repulsion and attraction result arrays, adds the two vectors and resizes them if needed. The data is then saved to the layout, after which the process can continue for another iteration or the data can be retrieved by the CPU if this was the last iteration.

All these parts of the algorithm combine to create the flow of data illustrated in Figure 3.2.

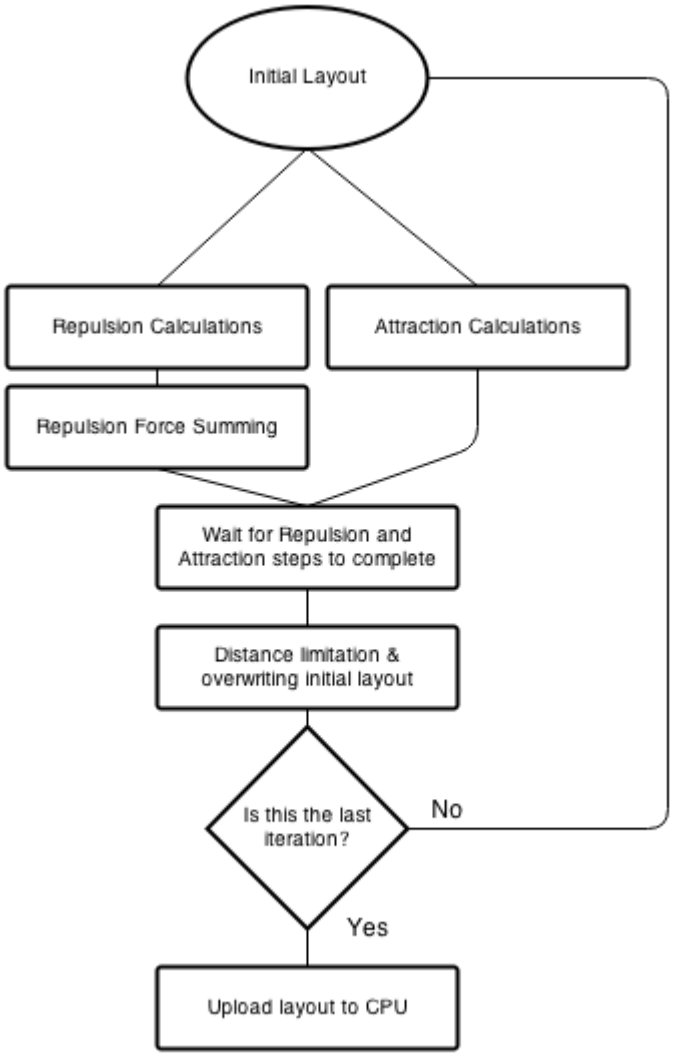

Figure 3.2: Data flow of parallelized FR algorithm.

3.3 Twilight Integration

Since both Twilight and the parallel implementation of FR use igraph the most logical way to integrate the solution within Twilight would be to make a custom igraph library with OpenCL support. This would however reduce the ease of compilation of Twilight because a custom build of igraph would have to be distributed and compiled, which might influence other programs that use igraph.

Therefore a choice was made to create an almost identical copy of the node_layout.cpp file within Twilight, which handles all the layout logic within Twilight. This edited file contains all the logic needed to run the FR algorithm in parallel using OpenCL. Having the OpenCL implementation in a different file is a better coding practice, but doing it this way minimizes the

number of changes that have to be made to the compilation process.

Because Twilight already uses CMake for compiling, CMake is used to differentiate between the OpenCL and regular versions of the algorithm. To do this FindOpenCL[2] is used, which has been released as public domain software.

[2] https://gitorious.org/findopencl

CHAPTER 4

Experiments & Results

4.1 Performance Testing Software

In order to test the performance of the parallelized algorithm a custom testing application had to be developed. This solution provides more freedom than testing directly in Twilight and therefore helped to create an iterative development process of the algorithm.

The testing software is written in the C programming language because the igraph[4] library, which drives all the graph based logic in Twilight, depends on C as well. The main use of igraph is to provide a sequential FR algorithm to compare results with. It is furthermore used to read graphs from a GraphML[2] formatted file and to create an initial circular layout, which is needed because the outcome of FR depends on the initial layout. The circular layout is used to provide both implementations with exactly the same starting layout. This ensures us that any differences in layout between the two algorithms are due to differences in the implementations only.

The suite has five different executables :

1. *speedup_commandline* : Measures the time to compute the layout for the graph given as argument.
2. *difference_commandline* : Compute the difference between the layout produced by the parallel implementation and the sequential implementation for the graph given as argument.
3. *visualization* : Visualizes the graph given as argument after 500 iterations of the parallel FR implementation. Run with -S flag to use sequential implementation instead.
4. *configure* : Generates a few graphs that can be used for testing purposes, the benchmark executable depends on these.
5. *benchmark* : Runs the tests that are detailed in the next chapter on the graphs that are generated by the configure executable.

All of the executables above except visualization are console based tools designed for testing and benchmarking the results of the parallel implementation. The visualization tool was built to function as an extremely lightweight Twilight clone. This tool is build with the OpenGL[9] and GLUT[8] frameworks.

4.2 Experiments

The performance testing software has been used to run some experiments that compare the results of the parallelized FR algorithm to the original, sequential, implementation.

The first experiment compares the execution time of both adaptations of the FR algorithm. The experiment consists of running both implementations for 250 iterations. This number of iterations is chosen to mimic a moderately simple layout situation. If the number of iterations would be larger, as would be the case if a layout had to be generated for more complicated graphs, the parallel implementation would be faster compared to the current situation. However increasing the number of iterations has a large influence on the running time of the algorithm and as such the experiment would take an impractical amount of time to execute on the sequential version of the algorithm. This experiment is conducted for 5 different graphs which only differ in the number of the vertexes. The average number of edges that originates from a vertex is kept constant as the repulsive section of the algorithm is the main influence in the running time of the algorithm. And as detailed in chapter 2, edges are not used in this part of the algorithm. The graphs that are used have the following parameters :

Number of Vertices	Number of Edges
100	297
1000	2997
5000	14997
7500	22497
10000	29997

All graphs are generated using the Barabasi[1] game graph generator included in the igraph library. Which is an algorithm that generates scale-free graphs. Scale-free graphs have the characteristic of containing nodes that have an above average number of connections and thus act like hubs of various sizes. This usually means that a few hubs can be removed from the graph without the graph losing connectivity of its nodes. This is a property that is thought to be present in many real-world networks[3] and is therefore chosen to represent real-life Twilight use cases.

The second experiment compares the locations of the nodes in the final layouts. Since CPU and GPU implementations of floating point arithmetic are different from each other, the layouts that are generated might be different too.

4.3 Results

All experiments have been performed on a laptop running Ubuntu 14.04 LTS with an Intel i7-3537U running at 2GHz and a Geforce GT 740M with 384 CUDA cores. All times measured are CPU-time.

4.3.1 Speed benchmarks

The speed benchmarks discussed in the previous section have been performed fifteen times for each graph giving the following results :

Table 4.1: Results of the execution time measurements of the parallel implementation and the sequential implementation of FR. All timing measurements are represented in milliseconds. Average speedup is obtained by dividing the average sequential time by the average parallel time.

No. of Nodes	**Parallel**		**Sequential**		**Avg. Speedup**
	min	**avg**	**min**	**avg**	
100	112	117	84	84	0.71
1000	826	834	7741	7939	9.52
5000	11220	11257	112352	169606	15.07
7500	24208	24327	251521	348940	14.34
10000	42972	43175	447845	684526	15.85

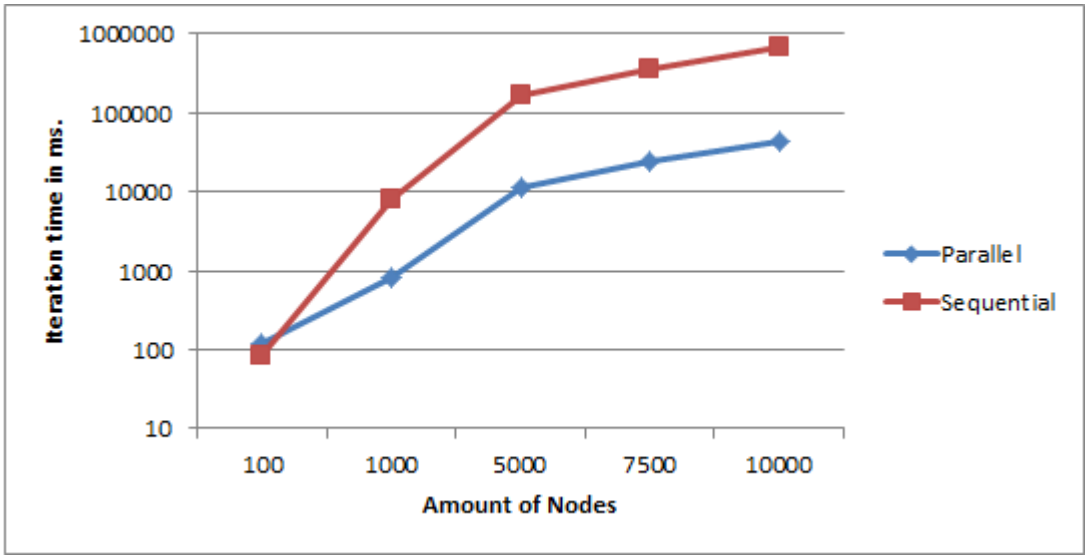

Figure 4.1: Average iteration times for the parallel and sequential version of the FR algorithm.

We can see that the parallel implementation is slower for very small graphs, which is to be expected since there is a significant overhead caused by the run-time compilation of the OpenCL kernels and the memory transfers needed to start the algorithm and retrieve the data when finished. The average speedup of the parallel algorithm quickly rises to 15 times and stabilizes there.

Table 4.2 and figure 4.2 show us that the parallel implementation has, for graphs that exceed 100 nodes, a relatively small deviation in measurements. Which means that the parallel implementation has a more stable execution time for those graphs. The explanation for this behavior is that long tasks have a high chance of having their memory swapped out on the CPU, contrary to the GPU where it remains allocated until freed up.

Table 4.2: Comparison of standard deviations for speed benchmarks. All measurements are represented in milliseconds.

No. of Nodes	**Parallel**		**Sequential**	
	std. dev.	**std. dev. in % of mean**	**std. dev.**	**std. dev. in % of mean**
100	4.79	4.08%	0.26	0.31 %
1000	5.75	0.69%	62.67	0.79%
5000	21.27	0.19%	37244.70	21.96 %
7500	368.10	1.51%	94644.27	27.12 %
10000	122.99	0.28%	135255.72	19.76%

4.3.2 Layout Comparison

The layout tests show some interesting characteristics from both implementations. The first test consists of comparing two layouts generated by the sequential algorithm for the same graph. Which, as was to be expected, turned out to be exactly identical every time. However, when this test is done with the parallel version of FR the algorithm has a small deviation in the results. This is caused by the attraction step. This step uses atomic adding in order to save memory on the GPU. The order in which the kernels complete their calculations, and thus the order in which they request access to the atomic part of the algorithm, is not exactly the same each time the algorithm is run. Therefore the atomic adding can still cause a race condition with regards to floating point rounding. Calculating the addition step on a single GPU core would remove this problem, but tests have shown that this causes the algorithm to lose up to 5 times speedup.

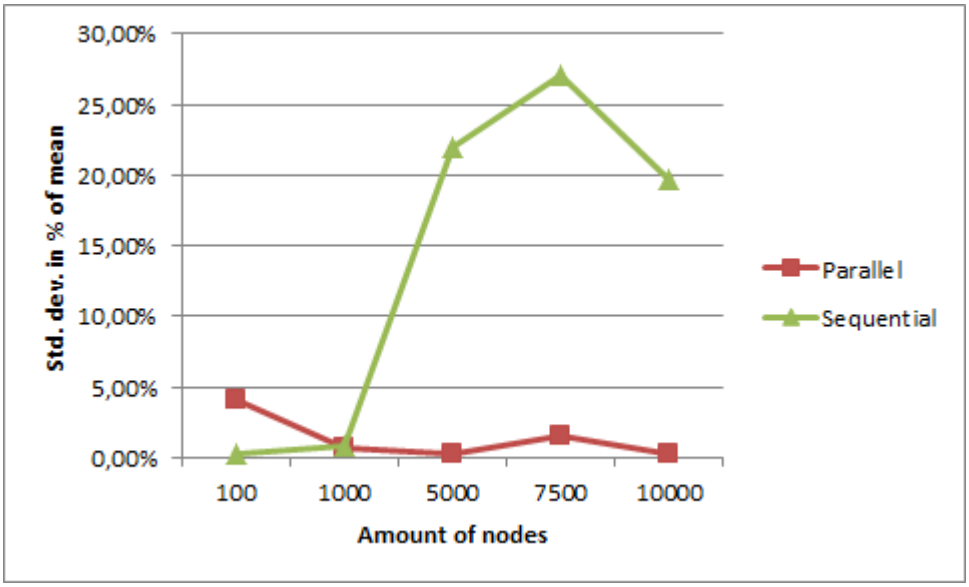

Figure 4.2: Standard deviation in % of mean for parallel and sequential FR implementations.

When this test is used to compare the parallel and sequential implementations of FR results still show differing layouts, even if the atomic calculations are replaced by calculations on a single GPU core only. Furthermore, comparison between the two variants of the addition steps show that the effect of the race condition is negligible on the total difference between the layouts.

Since both versions perform exactly the same calculations, the difference is likely caused by a combination of two factors. The first being that the parallel implementation uses floats to save memory space, which is the primary bottleneck of the current parallel implementation. Whereas the sequential algorithm uses double precision floats. The second being that both the CPU and GPU have different implementations of floating point arithmetic. Research shows[12] that this may cause differences in how numbers are rounded. Both these factors could cause small deviations in precision, but when those deviations are compounded for hundreds of iterations this difference can become quite large.

The effect of these deviations on how the graphs are interpreted by researchers is currently unknown. A visual inspection of the two different layouts show that the main differences in the two graphs are in the location of the outliers. So the main clusters and the locations of nodes within those clusters seem to be preserved between the two different implementations. Which hopefully limits the amount of influence that the different layouts have on the observations made by researchers using Twilight.

CHAPTER 5

Conclusion

5.1 Conclusion

Following the results of this research we can conclude that it is certainly possible to accelerate the execution time of the Fruchterman-Reingold algorithm by computing a parallel version of the algorithm on Graphical Processing Units. Because of the highly portable nature of OpenCL it was the platform of choice for this implementation. This does not specifically mean that OpenCL allows for the best performance of the algorithm. Instead it caters more to the wishes of the University of Amsterdam, whom requested that the application would remain highly portable.

There are however some non-trivial problems that arise from using such an implementation. Those problems include :

- Different layouts generated due to differences between floating point arithmetic on GPU and CPU.
- Quadratically scaling memory requirements, which imposes an upper limit on the size of the graphs that can be computed by this algorithm.

For situations where the above two points are not an issue this research provides a parallel implementation of the Fruchterman-Reingold algorithm with an average speedup of 15.85 times.

5.2 Future Work

Future research could concentrate on the memory usage aspects of the algorithm. Whereas this research failed to create an implementation that used constant memory space, it is still theoretically possible. The biggest setback in creating the memory constant variant of the algorithm was that the implementation generated an unknown OpenCL error, which could not be debugged. Perhaps this unknown OpenCL error will be fixed, or at least identified in a future update to OpenCL.

Building on a more memory efficient version of the algorithm another improvement that could be made is to upgrade the algorithm to use double (or higher) precision floating numbers. Since the sequential algorithm already does this, it is expected that this improvement would bring the final layouts of the two algorithms closer together.

A final recommendation is to attempt to generalize the execution of the algorithm in order to make it more broadly usable. This research focused purely on the usage within Twilight and as such has dependencies on the igraph library. Which is not strictly needed for the execution of this algorithm, but for this specific goal helped move development of the algorithm along.

Please do note that the author has chosen to keep the source code of the OpenCL implementation closed and therefore unavailable for the general public since it is his view that without the aforementioned memory optimizations the algorithm is not ready for general usage. Anyone who is interested in continuing this research is encouraged to contact the author through the University of Amsterdam.

Bibliography

[1] Réka Albert and Albert-László Barabási. "Statistical mechanics of complex networks". In: *Rev. Mod. Phys.* 74 (1 2002), pp. 47–97. DOI: 10.1103/RevModPhys.74.47. URL: http://link.aps.org/doi/10.1103/RevModPhys.74.47.

[2] Ulrik Brandes et al. *Graph markup language (GraphML)*. Bibliothek der Universität Konstanz, 2010.

[3] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. "Power-law distributions in empirical data". In: *SIAM review* 51.4 (2009), pp. 661–703.

[4] Gabor Csardi and Tamas Nepusz. "The igraph software package for complex network research". In: *InterJournal, Complex Systems* 1695.5 (2006).

[5] Anton J Enright and Christos A Ouzounis. "BioLayoutâĂŤan automatic graph layout algorithm for similarity visualization". In: *Bioinformatics* 17.9 (2001), pp. 853–854.

[6] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. "A comprehensive performance comparison of CUDA and OpenCL". In: *Parallel Processing (ICPP), 2011 International Conference on*. IEEE. 2011, pp. 216–225.

[7] Thomas MJ Fruchterman and Edward M Reingold. "Graph drawing by force-directed placement". In: *Software: Practice and experience* 21.11 (1991), pp. 1129–1164.

[8] Mark J Kilgard. *The OpenGL utility toolkit (GLUT) programming interface API version 3*. 1996.

[9] Randi J Rost. *OpenGL shading language*. Addison-Wesley Professional, 2004.

[10] Paul Shannon et al. "Cytoscape: a software environment for integrated models of biomolecular interaction networks". In: *Genome research* 13.11 (2003), pp. 2498–2504.

[11] Athanasios Theocharidis et al. *A Comparison of CPU and OpenCL Parallelization Methods for Correlation and Graph Layout Algorithms used in the Network*. http://www.biolayout.org/wp-content/uploads/2013/01/Manuscript.pdf. Jan. 2013.

[12] Nathan Whitehead and Alex Fit-Florea. "Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs". In: *rn (A+ B)* 21 (2011), pp. 1–1874919424.

[13] Narges Zarrabi et al. "Combining epidemiological and genetic networks signifies the importance of early treatment in hiv-1 transmission". In: *PloS one* 7.9 (2012), e46156.