University of St Andrews

SENIOR HONOURS PROJECT

Network Science on GPUs

Author:
Martynas Noreika

Supervisor:
Prof. Simon Dobson

April 9, 2018

Declaration of Authorship

I, Martynas Noreika, declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 10,166 words long, including project specification and plan.

Abstract

Network science has been used in many areas of research to design models of complex systems. By simulating processes on networks, researchers are able to get useful insight into the phenomena that govern such systems. With the improvements in GPU technology, successful attempts were made to parallelise network simulations. However, there seems to be a lack of research in the area of general network simulators that could efficiently simulate a wide range of models and networks in parallel. We investigated the possible implementation approaches of such solutions and developed a prototype that is capable of simulating SIR epidemic model on ER networks using NVIDIA GeForce GTX 1060 GPU and managed to achieve an average execution time speedup of 87.4 under certain simulation conditions, when compared to a single core sequential CPU implementation. Further analysis also showed that it produces stable performance with varying simulation parameters and can be extended to simulate arbitrary models and networks with some limitations.

Contents

Chapter 1

Introduction

Networks, as a concept, have been used in many fields of science and research, to provide a description model over interactions between various entities and phenomena. This abstraction allows us to reason about such complex systems as social networks, Word Wide Web, metabolic pathways and many others. By simplifying the systems to a network model we are able to come up with useful insights regarding the patterns of interactions between the entities in a system and even predict the behaviour or improve the efficiency of such interactions. For example, a model of a computer network might help us to determine the best path to take in order to transfer information faster or an epidemic model might help us to deduce how a decease will spread in a population.

Since the appearance of the first computers, simulations of all kinds have been an active research area in Computer Science. In particular, network simulations have been used to advance fields such as biology, physics, chemistry and many others. A model of a complex system can be reduced to a simple graph representation and with a simple set of rules the system can be simulated and analysed. This realisation led Computer Scientists to try and find the most optimal algorithms and ways of representing networks, in order to be able to simulate larger and more complicated systems. Furthermore, recently with the advancements in machine learning, there has been a huge interest in developing and finding the best hardware and algorithms to use for simulations of neural networks. Graphics Processing Units (GPUs) have been used as the hardware solution in speeding up the learning process. Initially designed for rendering three dimensional graphics, the GPUs are now used to power most of the machine learning happening today and it seems to find use in other, more general, areas of research as well. One of which is network science.

This project aims to investigate how GPUs could be used to speed up network simulations and develop a prototype that would be able to efficiently simulate custom network topologies with arbitrary simulation models. The rest of this report is structured into five sections: Chapter 2 provides the context survey, Chapter 3 and 4 discusses the design and implementation of the prototype, Chapter 5 performs the evaluation and Chapter 6 concludes the results.

Chapter 2

Context Survey

A network, also called a *graph* in mathematics, can be described as a collection of *nodes*, also known as *vertices*, and a collection of edges that connect the nodes. Each edge connects two nodes together. The structure of the network is called its topology and depending on the edges, networks can have various kinds of topologies. For example, some network may include directed edges, where edges are defined by specifying their preferred direction. We refer to such networks as directed networks. Others, contain parallel edges, where two edges are connecting the same pair of nodes, or self-loops, where both end points of an edge are connected to the same node. In general, these types of edges are rarely used [3]. A network that contains only undirected edges, has no parallel edges, and no self-loops is called a simple network.

Furthermore, it is possible to generate random networks that maintain some kind of overarching properties. For example, ER networks, introduced by Erdős and Rényi, can be defined by specifying a probability of an edge between any two arbitrary chosen nodes in the network. Such networks are easily defined and are complex enough to be used for describing sophisticated systems [3].

Systems can be simulated by applying processes on networks. A process changes the state of the system over time. The realisation of any process is a time series, where the elements in it are the state of the system at successive moments [3]. Time series can be deterministic, where the process state can be uniquely specified at any time from the initial conditions, or stochastic, having an element of randomness that leads to different states with the same initial conditions [3]. By performing simulations at the node to node basis we can reason about the global phenomena affecting the network [3]. A good example of this is the process of epidemic spreading [3].

2.1 SIR Epidemic Compartment Model

Epidemics can be described as a process running over a social network, where the edges represent social connections between individuals [3]. We can split the individuals in the network into groups or so called compartments and simulate the progress of a disease by moving these individuals between the groups. For example, the individuals can become infected and then recover. A simple compartment model of a disease, referred to as the SIR model, can be defined by splitting the individuals into three groups: *Susceptible*, *Infected* and *Recovered* or *Removed*. We can describe how the population of each compartment changes over time by defining the probabilities of each transition. In example, if we define probabilities P_{infect} and P_{recover}, then we can simulate, during each time step, how the susceptible nodes will transition to infected and the infected to recovered over time. Such simulations might give us insight into the spread of a particular disease in a population over time.

2.2 General Purpose GPU Computing

Graphic Processor Units (GPUs), traditionally developed for processing and rendering of realtime, high-definition 3D graphics, have evolved into a highly parallel, multithreaded, manycore processor with a significant computational power and high memory bandwidth. Figure 2.1 illustrates the differences between the CPU and the GPU architectures.

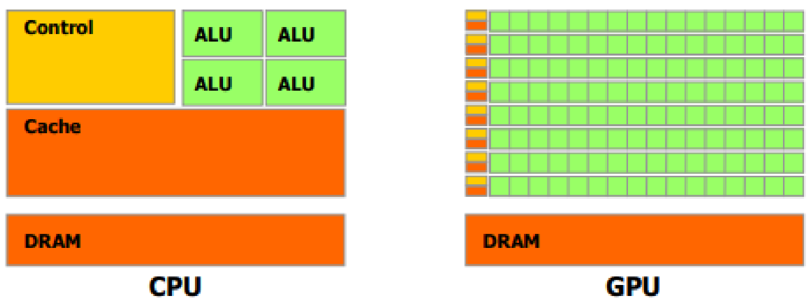


FIGURE 2.1: CPU and GPU architecture comparison. *Image from CUDA C Programming Guide [14]*

The GPU specialises in compute-intensive, highly parallel computations and therefore is designed in a way so that more transistors are devoted to data processing rather than data caching and flow control, as in the case of a CPU [14]. It is well-suited to problems that are possible to solve with data-parallel computations [14].

GPUs are build around an array of Streaming Multiprocessors (SMs) that execute the program in parallel [14]. The program is mapped onto a computation grid which is divided into blocks (Figure 2.2). Each block contains a fixed number of threads and is assigned to a Streaming Multiprocessor during execution. The threads in the block are executed in parallel. Moreover, there is a limit to the number of threads per block, as all threads in a block are expected to reside on the same processor core and have to share the limited resources available [14]. The number of thread blocks in a grid depends on the size of the data being processed and the number of processors in the system [14].

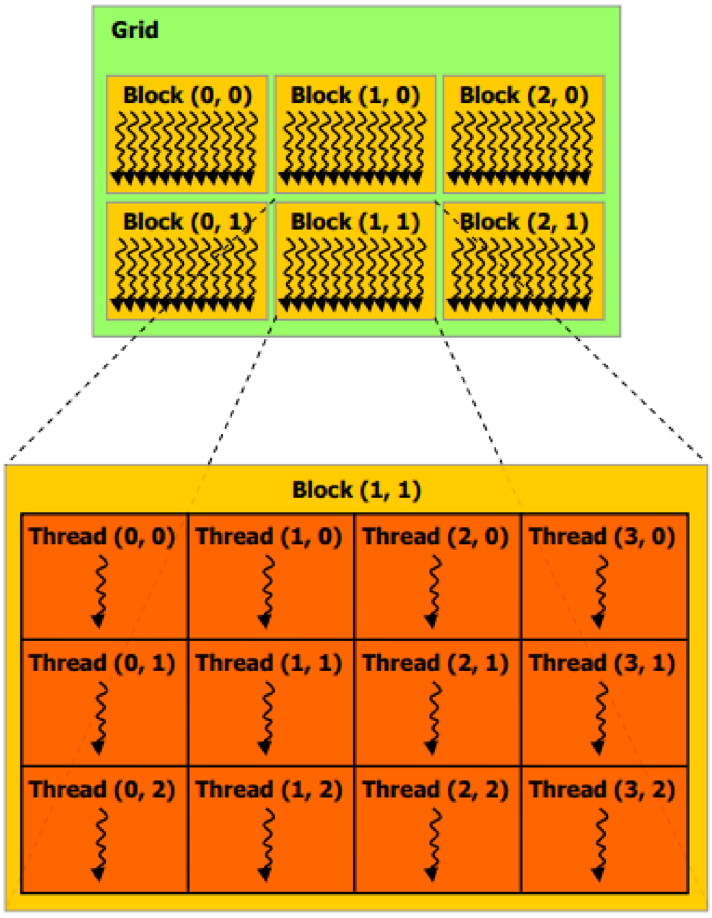


FIGURE 2.2: Computational grid. *Image from CUDA C Programming Guide [14]*

Each thread has access to multiple memory spaces during the execution [14]. Apart from the global memory, each thread has a private local memory and also a shared block wide memory to access.

In the CUDA programming model, developed by NVIDIA [13], the functions called kernels are used to perform the calculations on the GPU. When the data is mapped to the computational grid, threads execute a kernel function and process each element in parallel. Ideally, all threads follow the same execution path during the parallel computation, any divergence in the thread program's flow reduces the hardware utilisation and can significantly hinder the performance [14], as some threads might have to stall until different branches are being computed. Figure 2.3 shows how a typical heterogeneous CUDA program is executed. We can see that the program switches between executing serial code on the CPU and parallel on the GPU.

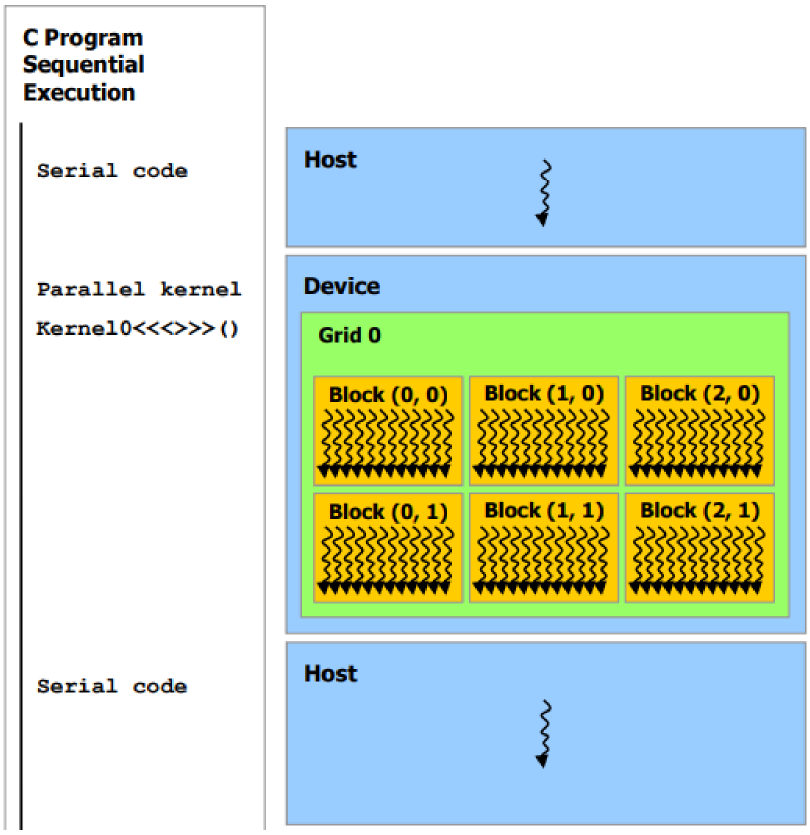


FIGURE 2.3: Heterogeneous execution of CPU and GPU code. *Image from CUDA C Programming Guide [14]*

2.3 GPU-based Network Simulations

Zhou et al. managed to accelerate biochemical network simulations using NVIDIA CUDA GPUs [17]. In their work they argued that mathematical modelling is central to systems and synthetic biology and, therefore, simulations are common means for analysing such models. They have noticed that such simulations are computationally expensive, however easily parallelisible. Therefore, they have designed a Python package called *cuda-sim* which provides highly parallelised algorithms for large scale simulations of biochemical network models [17]. Using the parallel implementation, the authors managed to achieve a 360-fold decrease in simulation runtime, when compared to a single CPU implementation.

Furthermore, another promising result was achieved by Bilel et al [2]. They have designed a distributed and parallel framework for simulating large scale wireless mobile networks. The framework called *Cunetsim* is able to combine the CPU and GPU in a co-simulation scenario where the GPU is the main simulation environment and the CPU is a controller [2]. This approach allowed them to distribute the computational work over several machines and achieve extra-large scale simulations. For example, in a benchmarking scenario which involved 1.5 millions nodes, the authors simulated an exchange of 4 billion messages during the period of 5602 simulated seconds.

2.4 SIR Epidemic Model Simulations

Attempts have also been made to parallelise the simulations of discrete SIR epidemic models. Šošić et al. [18] came up with a parallel solution that managed to achieve a 10x faster execution time when compared with a common CPU implementations. The implementation was tested on real social networks consisting of more than 5 million nodes [18]. The solution uses a graph, where edges are represented as an adjacency list in memory. This list is stored in a single large array making it convenient to store and use on a GPU. In this way, a graph is being held in two arrays, N and L [18]. Array N represents nodes of a graph where each node points to a the starting position of its own adjacency list in L. This way a network can be represented in an optimal manner. The simulation is executed in steps and during each step the CPU analyses information regarding the current state in the network and calls an appropriate CUDA function [18]. Each node is identified by a particular level. The infected node is given a level 0, while the adjacent neighbours of that node are in level 1, their neighbours in level 2 and so on [18]. Initially, only one node has to be evaluated and then depending if the infection spreads, other higher node levels are evaluated as well. This approach is similar to a BFS (*Breadth First Search*), where each node is traversed (in this case infected) with a probability of infection [18]. During the simulation, the implementation operates in two phases: Single Instruction Single Data (SISD) and Single Instruction Multiple Data (SIMD). In the SISD phase, the node array is split between groups of CUDA threads and each group sequentially loops through the assigned nodes. If the currently inspected node is infected, the execution diverges for each thread in a group and enters the SIMD phase. In this phase, each thread in a group tries to infect a different neighbour of the infected node [18]. After all the neighbours are processed, the thread group continues scanning the array N for other infected nodes. This process continues until the whole network is infected or there are no more susceptible nodes left. In the latter scenario, the algorithm uses a different CUDA function to inspect nodes. This time, instead of diverging and processing all the neighbours of the infected node, the solution just tries to *heal* the infected node with an appropriate probability [18]. Finally, after there are no more infected nodes in the network, the simulation is terminated. According to the authors, the spread of infection can be inferred from a data structure called *Levels*, where the moments of infection are being held for each node.

2.5 Limitations of Current Research

The discussed papers show that there is scope for introducing parallelism in various simulations of networks and that significant speedups can be achieved. Zhou et al. [17] used GPUs to speedup the simulations of synthetic biology models. Furthermore, Bibel et al. [2] used a cluster of GPUs controlled by the CPU to achieve an extra-large scale simulations and showed that it is possible to scale the simulations beyond one GPU. Finally, Šošić et al [18] introduced a parallel algorithm for simulating the SIR epidemic model. However, all of these approaches also have certain limitations.

They seem to focus on simulating a very particular type of network and simulation model and do not try to generalise their algorithms for a wider range of possible network and simulation scenarios. The algorithm proposed by Šošić et al [18] could be potentially extended to work with an arbitrary set of node types and simulation models. However, it would seem to have multiple limitations if such an attempt would be made. For example, in the suggested solution the nodes are updated by dividing the node array to a group of threads to process. Each thread group then traverses the assigned nodes in sync and diverges in case an infected node is found. This approach seem to work well with SIR networks, as only one node type could affect the adjacent nodes. However, in a more complicated simulation model such approach might slow down due to a potentially more frequent thread flow divergence scenarios. Furthermore, the proposed solution [18] also seem to lack the ability to end the simulations after an arbitrary number of timesteps and it is not clear if the full state of the network could be inferred at each step of the simulation process.

Therefore, there seems to be a lack of research in the area of a general network simulator that might be slower in terms of performance when compared with a specialised algorithms for a particular problem, however would be able to efficiently simulate a much wider range of models and networks in a scalable manner.

Chapter 3

Design

This project explored the path of building a general network simulator on a GPU, that would be able to simulate an arbitrary system model on a custom network topology. This problem seemed too complex to start tackling first, therefore we decided to initially build an efficient parallel algorithm for a specific simulation problem and then see if it would be possible to generalise it to work with a wider range of networks and simulation types.

The initial implementation focused on designing a parallel algorithm for SIR epidemic network simulations. The SIR epidemic model was chosen due to its simplicity. There are only three different nodes types and the transition rules, from one type to another, can be defined relatively easily. Furthermore, it has stochastic properties, which seemed a good feature to explore and have in a generalised network simulator. Finally, the SIR model seemed to be a useful example of how the designed simulator potentially could be used in the field of epidemiology and others. Moreover, ER network model was used to define the networks, as it provides an easy way to probabilistically vary the topology of the network.

To start with, we did a lot of research to understand how GPUs work and what approach should be taken to successfully parallelise the simulations. The two biggest source of information were the *NVIDIA Programming guide* [14] and the *NVIDIA Developer Forum* [15]. The reasons for choosing NVIDIAs programming infrastructure and graphic cards rather than other available alternatives will be explored in Chapter 4. Furthermore, to learn about the basic principles of network science and, in particular, the mechanics of SIR models, we investigated multiple sources and the three most useful ones were Simon Dobson's blog on *Complex networks, complex processes* [3], Newman's paper on *The spread of epidemic disease on networks* [11] and his book titled *Networks An Introduction* [10].

During the design process, it became clear that the simulator could be structured in three separate execution parts or phases. The responsibilities of each phase are summarised below:

- **Initialise**
  The initialisation phase is responsible for setting up the topology of the network and configuring the initial node and edge types.
- **Simulate**
  During the simulation phase, the network should be simulated the provided number of timesteps, where during each timestep, the affected nodes are updated according to the rules specified.

- **Evaluate**
  The final phase, evaluation, should determine the final state of the network after the simulation is finished.

We developed two algorithms that follow this design pattern and successfully simulate the SIR epidemic model on the ER networks. The first algorithm, called *Naive Parallel SIR* produced correct results, however was relatively inefficient and didn't scale well with bigger networks. Therefore, we made an attempt to speed up the computations and, as a result, the second algorithm, called *Improved Parallel SIR*, was devised. The rest of this chapter explores the design details of both algorithms.

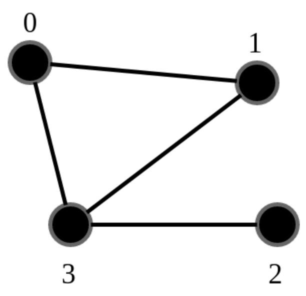
3.1 Naive Parallel SIR

The initial idea was to come with up a simple parallel solution that would perform the simulations correctly and then try to improve it and make it more efficient and scalable. This simple solution developed into the *Naive Parallel SIR* algorithm.

3.1.1 Memory Representation

A decision had to be made regarding how the network will be stored in memory and if that representation would be efficient on GPU architecture. Šošić et al [18] used an adjacency list to store the connections between the nodes. Each node would contain a list of pointers that would indicate the connections this node has with other nodes. This approach seems to be dynamic in terms of memory usage, as in a loosely connected network, the nodes will have a smaller adjacency list and vice versa. However, in case when the network connectivity is high, the adjacency list would store a lot of duplicate data, as each edge would be defined twice by two pointers from each end nodes.

The alternative, called an adjacency matrix, uses a matrix to describe the edges between the nodes.



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 1 & 1 & 1 & 0 \end{array}$$

FIGURE 3.1: A simple network and its adjacency matrix

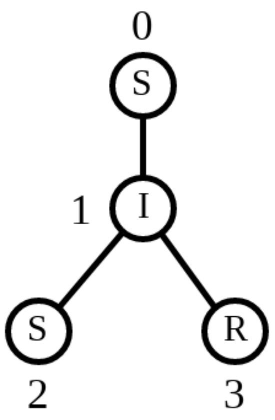
As we can see in Figure 3.1, the connections of a simple network with four nodes can be defined with an adjacency matrix with sixteen elements. The index of each row and column corresponds to an appropriate node in the network. While, the elements in the matrix indicate if an edge exists or not between the two corresponding nodes. One implies a connection and zero - a lack of one. The adjacency matrix seems to fit the GPU computational model well [14]. It provides a data structure that is easily divisible to smaller chunks that can be processed in parallel. On the other hand, in a scenario where the network is loosely connected, the matrix would be sparse and consume a lot of unnecessary memory. Although, it might be possible to compress the matrix in such scenarios.

Both approaches can be potentially utilised to provide an efficient network memory representation. Nonetheless, our solution focuses on utilising the adjacency matrices to represent the network in the GPU memory, as it seemed to more naturally fit the GPU computational model [14].

In the implementation, the adjacency matrix is flattened in row-major order and stored as a contiguously allocated array in the GPU memory. This simplifies the memory access operations and potentially allows some of the memory request to be coalesced into a single memory transaction [14].

3.1.2 Edge map

In the SIR epidemic model, the nodes are divided into three compartments or types: *Susceptible*, *Infected* and *Recovered*. During the simulations, the nodes have to be updated according to the model provided. Therefore, the implementation needs to keep track of the node state at each timestep. This could be achieved by using an additional data structure, however that would require more memory and might potentially complicate the update procedure. An approach that is used in the algorithm, utilises the adjacency matrices to store the node types. This way reducing the memory footprint and the number of memory accesses required. The typical adjacency matrix can only describe if an edge exists between two nodes in a network. However, we observed that it is possible to implicitly specify the node types by defining an edge type for each combination of nodes and storing it as an entry in the matrix.



$$\begin{array}{c c} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{pmatrix} 0 & 1 & 0 & 0 \\ 4 & 0 & 4 & 5 \\ 0 & 1 & 0 & 0 \\ 0 & 7 & 0 & 0 \end{pmatrix} \end{array}$$

FIGURE 3.2: SIR epidemic network and its edge matrix

Figure 3.2 shows an example of such an approach. Each entry in the edge matrix indicates a combination of nodes that comprise that edge. The corresponding arbitrary mappings between the edge entries and the node types can be seen in Figure 3.3. In this notation, the first node type indicates the type of the row index node, while the second one - column index node. Therefore, the entry *1* in the edge matrix implies that the node corresponding to row index is *Susceptible* and the node corresponding to the column index is *Infected*.

$$
\begin{aligned}
1 &\implies SI \\
2 &\implies SR \\
3 &\implies SS \\
4 &\implies IS \\
5 &\implies IR \\
6 &\implies II \\
7 &\implies RI \\
8 &\implies RS \\
9 &\implies RR
\end{aligned}
$$

FIGURE 3.3: Edge map

With this approach, we can, at any point, infer the types of the nodes by simply mapping the edge type to its appropriate group of nodes. Furthermore, during the simulation process, we can now easily identify the edges that link the *Infected* and *Susceptible* nodes and update them with a given probability accordingly.

3.1.3 Initialisation

In the initialisation phase, the *Naive SIR Algorithm* sets up the topology of the network and initialises the edge map. Two input parameters are used in the process: the probability of an edge and the probability of a node being infected at the start. Algorithm 1 defines the pseudocode for this step.

Algorithm 1 Naive SIR Initialisation

```
1: procedure INITIALISE($edges, N$)
2:     for every entry in $edges$ do
3:         $index \leftarrow$ entry index
4:         $rand \leftarrow$ random float
5:
6:         $rowIndex \leftarrow index$ / $N$
7:         $colIndex \leftarrow index \mod N$
8:         $simIndex \leftarrow i + (colIndex - rowIndex) * (N - 1)$
9:
10:        if $rowIndex < colIndex$ and $rand \leq P_{\text{edge}}$ then
11:            $edges[index] \leftarrow$ SS
12:            $edges[simIndex] \leftarrow$ SS
13:
14:    for every node do
15:        $nodeIndex \leftarrow$ node number
16:
17:        if $edges[nodeIndex] \neq 0$ then
18:            $rand \leftarrow$ random float
19:            if $rand \leq P_{\text{infected}}$ then $initEdges(nodeIndex)$
```

The procedure starts by traversing every edge in the edge matrix and establishing a connection between the nodes with a given probability of P_{edge}. This is achieved by calculating the row and column indexes in the matrix from the flattened matrix array index. Using those values we can then calculate where the symmetrical edge entry will be located in the matrix. Then, a random number is generated and if it is in the bounds of the probability, both edge matrix entries are updated to an edge type of SS. Updating the edge entry and its symmetrical equivalent at the same time prevents us from rolling the die twice for the same edge. After this step, the entries in the edge matrix contain either the edge type of *SS*, which implies a connection between two susceptible nodes, or *0*, which suggests a lack of one.

Moreover, after defining the topology of the network and the initial node types, the procedure continues to infect the nodes with a given probability of P_{infected}. For every connected node a random number is generated and if it is in the bounds of the probability, the node becomes infected. To update the edges associated with that node, the *initEdges* kernel function is called. It, in parallel, finds the entries that are related to the newly infected node and updates their values accordingly. Notice, that most of procedure is executed on the CPU and only the *initEdges* function performs computations on the GPU.

3.1.4 Simulation

During the simulation phase, the network is updated each timestep according to the rules defined in the simulation model. In the case of SIR epidemic model, two types of updates are possible: the susceptible nodes can become infected, if they are adjacent to an already infected node, and the infected nodes can recover. The pseudocode for this procedure is described in Algorithm 2.

Algorithm 2 Naive SIR Simulation

```
1: procedure SIMULATE STEP($edges$)
2:     $sEdgeTypes \leftarrow (SI)$
3:     $iEdgeTypes \leftarrow (IS, IR, II)$
4:
5:     $findAndUpdateEdges(edges, sEdgeTypes, P_{infect})$
6:     $findAndUpdateEdges(edges, iEdgeTypes, P_{recover})$
```

Each timestep, the *simulate step* procedure identifies the susceptible and infected nodes and updates them with the given probabilities of P_{infect} and P_{recover}. The algorithm achieves this by first defining a set of edge types that would contain susceptible and infected nodes and another set of edge types that contain infected nodes. The edges are selected so that the target node is the first node in an edge 3.3. This way the die is only rolled once for each target node. Furthermore, a *findAndUpdateEdges* procedure is called on each set (Algorithm 3).

Algorithm 3 Naive SIR Simulation

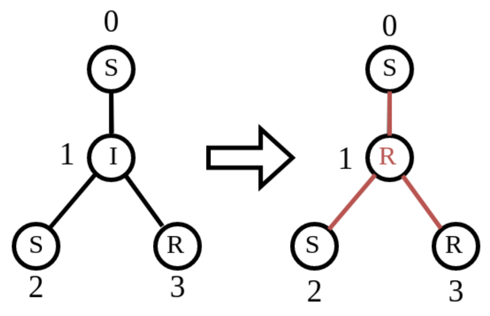
```
1: procedure FIND AND UPDATE EDGES($edges, edgeTypes, p$)
2:     $targetEdges \leftarrow findEdges(edges, edgeTypes)$
3:
4:     while edge in $targetEdges \neq -1$ do
5:         $rand \leftarrow$ random float
6:
7:         if $rand \leq p$ then
8:             $updateEdges(edges, edge)$
```

It finds the indexes of edges that have the specified edge types. Then, it traverses until a valid index is present and with a given probability of p updates the edges using the *updateEdges* kernel function. This functions runs on GPU and in parallel updates the edge type according to their current type. It determines the row$^{\text{th}}$ and column$^{\text{th}}$ nodes of the target edge and updates the edges that contain those nodes by upgrading their type. In SIR epidemic model, the node types always change in a unambiguous manner, therefore a one-to-one mapping for the edge types can be defined, as shown in Table 3.1. For example, if a susceptible node is being updated and it is the row$^{\text{th}}$ node of an *SI* edge, then the edge type will be updated to *II* and so on. This way the changes in the network can be simulated and propagated each timestep.

Edge Update Map			
Susceptible to Infected		Infected to Recovered	
Row node	Column node	Row node	Column node
$SI \implies II$	$SS \implies SI$	$IS \implies RS$	$SI \implies SR$
$SR \implies IR$	$IS \implies II$	$IR \implies RR$	$II \implies IR$
$SS \implies IS$	$RS \implies RI$	$II \implies RI$	$RI \implies RR$

TABLE 3.1: Edge Update Map

Another example is illustrated in Figure 3.4. In this case, node one is updated from being Infected to Recovered. The diagram illustrates which edges are be affected by the change and will have to be updated. We can see that if node one changes, then all non zero entries in the matrix that are in the first row and column have to be updated to account for the changed edge types.



$$\begin{array}{c|cccc} & 0 & \mathbf{1} & 2 & 3 \\ \hline 0 & 0 & 1 \Rightarrow 2 & 0 & 0 \\ \mathbf{1} & 4 \Rightarrow 8 & 0 & 4 \Rightarrow 8 & 5 \Rightarrow 9 \\ 2 & 0 & 1 \Rightarrow 2 & 0 & 0 \\ 3 & 0 & 7 \Rightarrow 9 & 0 & 0 \end{array}$$

FIGURE 3.4: Updating a node and propagating changes in the edge matrix

Finally, to find the edges of certain types, a *findEdges* procedure is called (Algorithm 4). It operates by first marking the edges that have one of the target types, calculating the offset for each marked edge in the result array and then storing the indexes of the edges, using the offsets, to the result array.

Algorithm 4 Naive SIR Simulation

```
1:  procedure FIND EDGES($edges, edgeTypes$)
2:      $markedEdges \leftarrow markEdges(edges, edgeTypes)$
3:      $offsets \leftarrow ()$
4:      $position \leftarrow 0$
5:
6:      for edge in $markedEdges$ do
7:          if edge is marked then
8:              $offsets[edge] \leftarrow position$
9:              $position \mathrel{+}= 1$
10:     return $filterEdges(edges, edgeTypes, offsets)$
```

This approach is required due to the parallel nature of the computations performed on the GPU. In comparison, storing the indexes of target edges would be simple on a CPU, as the computations are done sequentially and, therefore, each time we find the edge that matches the target type we can store its index in the result array and increment the counter that indicates the next free slot in the array. However, on a GPU, the computations happen in parallel and the order of executions is not guaranteed. Therefore, the execution threads that process each edge matrix entry have to synchronise in order to not overwrite each other's results. To resolve this issue, the procedure use an approach called *Stream Compaction* [1]. In this scenario, it is implemented by first calculating the indexes in the result array for each thread and then using those indexes to store the results in parallel.

3.1.5 Evaluation

The evaluation phase, determines the state of the network after the simulation. As discussed before, we have chosen to represent our node types implicitly in the edges. Therefore, the *evaluate* procedure counts the different node types given the edge matrix, as described in Algorithm 5.

Algorithm 5 Naive SIR Evaluation

```
1: procedure EVALUATE($edges$)
2:     $sNodeTypes \leftarrow (SI, SR, SS)$
3:     $iNodeTypes \leftarrow (IS, IR, II)$
4:     $rNodeTypes \leftarrow (RI, RS, RR)$
5:
6:     $sNumber \leftarrow countNodes(edges, sNodeTypes)$
7:     $iNumber \leftarrow countNodes(edges, iNodeTypes)$
8:     $rNumber \leftarrow countNodes(edges, rNodeTypes)$
```

For each node type, the procedure defines a list of edges that contain that type. Then, it calls a *countNodes* procedure, described in Algorithm 6, that counts the number of nodes for a particular set of edge types. In the SIR epidemic model, three node types are available: *Susceptible*, *Infected* and *Recovered*. Therefore, the state of the network can be determined by calculating the occurrence number of each.

Algorithm 6 Naive SIR Evaluation

```
1: procedure COUNT NODES($edges, nodeTypes$)
2:     $labelledNodes \leftarrow labelNodes(edges, nodeTypes)$
3:     $sum \leftarrow 0$
4:
5:     for node in $labelledNodes$ do
6:         if node is labelled then
7:             $sum += 1$
8:     return $sum$
```

To calculate the nodes of a particular type, a kernel function called *labelNodes* is used to in parallel analyse each edge and determine if it has the target node type. The function works by marking the nodes that have a particular type and then counting all the marks.

3.1.6 Analysis

After some initial experiments and tests, the *Naive Parallel SIR* algorithm seemed to be simulating the SIR epidemic model correctly, however, performance wise, it did not seem to scale well at all and performed poorly when compared to a sequential single core CPU implementation.

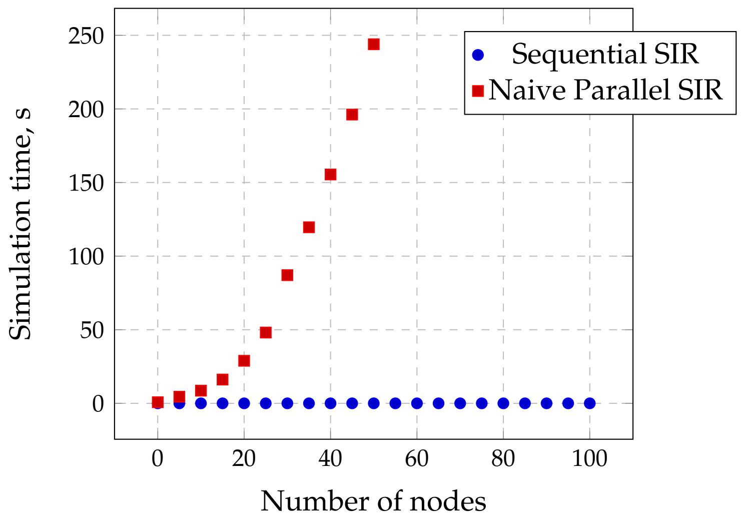


FIGURE 3.5: Naive Parallel SIR and Sequential SIR comparison.
Simulation parameters: $P_{infected}$=0.01, P_{infect}=0.2, $P_{recover}$=0.1, *timesteps*=20000

Figure 3.5 shows a preliminary comparison between a sequential SIR, implemented by Dobson [3], and the *Naive Parallel SIR* algorithms. We can see that the *Naive Parallel SIR* algorithm takes a significant amount of time longer to simulate relatively small networks in comparison to the sequential implementation. It provides a basic approach that performs the simulations correctly, however does not manage to scale well. After initial analysis, we noticed that it requires a lot of device switching during the simulation process. Only a small part of computations are actually executed on the GPU, the rest is performed by the CPU. During each timestep, the CPU probabilistically decides which nodes should be updated and the GPU propagates those changes over the network. However, while CPU is working, GPU has to stall and its maximum utilisation is significantly reduced. Therefore, it is possible that this constant switching between the CPU and the GPU hinders the performance. Furthermore, there seems to be much more scope for parallelism than it is currently achieved. For example, during the simulation, the nodes are updated sequentially and then the changes to the edges are propagated in parallel, as portrayed in Figure 3.4. In this case, it might be possible to execute both steps in parallel providing the updates are synchronised in some way.

3.2 Improved Parallel SIR

Based on the insights gained from the analysis of the *Naive SIR* algorithm, we developed a new approach that aims to mitigate the issues found and progress further. The improved algorithm builds on top of the same design pattern as the *Naive SIR* one and it introduces new design and implementation level ideas to improve the performance.

3.2.1 Memory Representation

The *Naive SIR* network representation uses an adjacency matrix that requires N^2 number of entries for N number of nodes. However, we can easily notice that the matrix is symmetrical and, thus, we only need to store less than half of the data to be able to describe the network unambiguously. The *Improved SIR* algorithm tries to exploit this fact and reduce the memory footprint.

Figure 3.6 shows how the edge matrix can be divided into two symmetrical parts. The dashed blue diagonal line splits the matrix in the upper and lower triangles. We can notice that both triangles describe equivalent edges, just from a different, however symmetrical, point of view. Furthermore, the symmetry observed here is not classical. The edge type values stored in the two triangles are not numerically equivalent, however store the same information, a connection between two nodes. The symmetrical values are descriptions of the same edge using a different node's perspective. Therefore, we can use only one of the triangles to fully describe the edges in the network.

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 0 & 0 \\ 1 & 4 & 0 & 4 & 5 \\ 2 & 0 & 1 & 0 & 0 \\ 3 & 0 & 7 & 0 & 0 \end{array}$$

FIGURE 3.6: Symmetry of the edge matrix

The implementation stores the upper triangle highlighted red in the diagram. It is flattened in row major order, and is stored as a continuous block in memory, to simplify memory accesses. However, this representation led to an unexpected problem. When we flatten the whole matrix, having an index of an element in the flattened array allows to simply determine its row and column indexes, as shown in Figure 3.7.

$$Row_{index} = i/N$$

$$Column_{index} = i \bmod N$$

FIGURE 3.7: Here i is an array entry index and N represents the node number in the network

However, the task of determining the row and column indexes from an entry of a flattened upper triangle, does not appear to be trivial. Initially, it seemed that there might be no analytic solution to perform this calculation. Therefore, an approach was taken where the decreasing length of each triangle row ($N-1, N-2, ...$) was subtracted from the index while the result remained positive and then the indexes could be determined from the number of subtractions required and the remainder of the last one. This solution had a linear time complexity and, thus, more optimal approaches, such as a binary search, were considered. In the end, a constant time solution was discovered.

$$Row_{index} = N - 2 - floor(\frac{\sqrt{-8 * i + 4 * N * (N-1) - 7}}{2} - 0.5)$$

$$Column_{index} = i + Row_{index} - \frac{N * (N-1)}{2} + \frac{(N - Row_{index}) * ((N - Row_{index}) - 1)}{2} + 1$$

FIGURE 3.8: Here i is an array entry index and N represents the node number in the network. Equations derived by McGibbon [8]

It seems that the sequence of row sizes in the upper triangle correspond to the triangular numbers [7], therefore we can use a triangular root to determine the row and column indexes analytically, as shown in Figure 3.8.

By storing an upper triangle of the edge matrix, we only need to keep track of $\frac{N^2-N}{2}$ memory entries, where N is the node number, in comparison to N^2 entries, when storing the whole matrix.

3.2.2 Initialisation

To speed up the simulation process, we tried to improve upon the pitfalls of *Naive SIR* and introduced new approaches. First of all, all computations have been moved to the GPU for a parallel execution. Furthermore, the node update procedure is performed completely in parallel, meaning that all edges are updated in one go, rather than in groups after each node change. To achieve this, a way of synchronising between the different node updates had to be introduced. We made an observation that no matter the order in which the nodes are updated, the edges between the nodes will have the same type in the end. For example, if two nodes *Susceptible* and *Infected* are connected by an edge, then in a scenario where they are both updated, no matter which node will be updated first, the final edge will still have the edge type *IR* or *RI*, depending on the perspective. Therefore, it is possible to update all the edges at the same time, if we know which nodes will be changing beforehand.

Following this approach, the pseudocode for the improved *initialise* procedure is described by Algorithm 7.

Algorithm 7 Improved SIR Initialisation

```
1: procedure INITIALISE($edges, node_{ui}, node_{uc}, N$)
2:     $initEdges(edges, N)$
3:     $initNodes(node_{ui}, N)$
4:     $updateNetwork(edges, node_{ui}, node_{uc}, N)$
```

We can see that two new variables have been introduced: the $node_{ui}$ and $node_{uc}$. The first one, *node update indicator* is used to keep track which nodes will be updated in the current timestep, while the second one, *node update conditions*, stores the probabilities with which the particular nodes are updated. Using these two additional constructs, we can determine which nodes will be updated beforehand and use that information to update the edges in one go.

The *initialise* procedure starts by setting up the connections between the nodes in the network. In contrast to the *Naive SIR*, this step is performed in parallel on a GPU. The procedure described by Algorithm 8 is called on each edge matrix entry.

Algorithm 8 Improved SIR Initialisation

```
1: procedure INIT EDGES($edges, N$)
2:     $index$ <- thread index
3:     $rand$ <- random float
4:
5:     if $rand \leq P_{\text{edge}}$ then $edges[index] = \text{SS}$
```

The *initEdges* kernel function initialises an edge by generating a random float and determining if it is lower or equal the probability of an edge. The *initNodes* procedure determines which nodes in the network will be initially infected in the same manner, however a probability of initially infected is used instead. Finally, an *updateNetwork* kernel procedure is called which updates the edges of the initially infected nodes in the network.

3.2.3 Simulation

The simulation phase, uses the same approach as described in the *initialisation* step. Every timestep, we first compute which nodes will be updated and then propagate the changes in the edge matrix (Algorithm 9). Compared to the *Naive SIR* approach, this solution increases the number of tasks that can be executed in parallel and improves the utilisation of the GPU hardware. The only synchronisation required is between the node and edge update procedures, the procedures themselves are executed concurrently.

Algorithm 9 Improved SIR Simulation

```
1: procedure SIMULATE STEP($edges, node_{ui}, node_{uc}, N$)
2:     $updateNodes(node_{ui}, node_{uc}, N)$
3:     $updateNetwork(edges, node_{ui}, node_{uc}, N)$
```

The most important part of the *Improved SIR* algorithm is the *updateNetwork* kernel function. It propagates the changes in the network and, at the same time, sets the update conditions of each node. We can see a pseudocode version of this procedure in Algorithm 10.

Algorithm 10 Improved SIR Simulation

```
1: procedure UPDATE NETWORK($edges, node_{ui}, node_{uc}, N$)
2:     $index \leftarrow threadindex$
3:     $matrixPosition \leftarrow calcMatrixPosition(index, N)$
4:     $edgeMap \leftarrow$ { ... Figure 3.1 ... }
5:     $probMap \leftarrow$ { ... Figure 3.9 ... }
6:
7:     if $edges[index]$ is an edge then
8:         $edgeType \leftarrow edges[index]$
9:         $rowNode_{ui} \leftarrow node_{ui}[matrixPosition.row]$
10:        $colNode_{ui} \leftarrow node_{ui}[matrixPosition.column]$
11:
12:        $edges[index] \leftarrow edgeMap[edgeType, rowNode_{ui}, colNode_{ui}]$
13:
14:        $newProb \leftarrow probMap[edgeType, rowNode_{ui}, colNode_{ui}]$
15:        if row node is updated then
16:            $node_{uc}[matrixPosition.row] = newProb.rowNode$
17:        if column node is updated then
18:            $node_{uc}[matrixPosition.column] = newProb.colNode$
```

During the network update, the *updateNetwork* function is mapped onto the edge array and each thread processes a single entry in the array. Firstly, the current edge type is determined. Then, depending if the row and column nodes are being updated in this simulation step, a new edge type is retrieved from the *edgeMap*. The map is indexed by the current edge type and two boolean values that indicate if row and column nodes are updated this simulation step. Furthermore, by updating the edge type, we are changing the node types as well. Therefore, the probabilities, or the update conditions, of each node have to be changed too. This is achieved using a probability map defined in Figure 3.9.

$$P_{\text{null}} \Rightarrow P_{\text{Infect}}$$

$$P_{\text{Infect}} \Rightarrow P_{\text{Recover}}$$

$$P_{\text{Recover}} \Rightarrow P_{\text{null}}$$

FIGURE 3.9: Probability Map

The *updateNodes* procedure indicates which nodes will be updated in the current simulation step by rolling a die for each node with regards to the probabilities defined in the $node_{uc}$ array. The nodes who cannot be updated are given a probability of P_{null}. While, for the others, the update probabilities have to be changed according to the probability map 3.9, as soon as they are updated. Therefore, the *updateNetwork* procedure changes the update conditions of the nodes that define the updated edges each timestep.

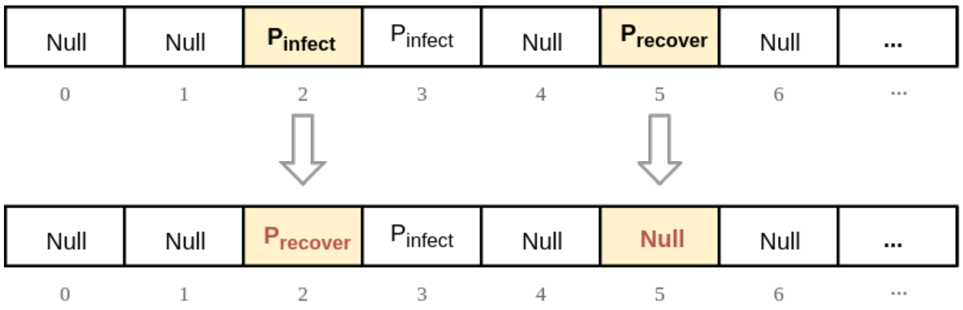


FIGURE 3.10: Upgrading node update probabilities

The simulation process is best illustrated with an example. Let us consider a scenario where two nodes *Susceptible* and *Infected* are connected by an *SI* edge and are both being updated during an arbitrary simulation step. In this case, the edge type will be changed from *SI* to *IR*. Furthermore, the node update conditions will be changed, for the *Susceptible* node, from P_{infect} to P_{recover} and, for *Infected* node, from P_{recover} to P_{null}, as portrayed in Figure 3.10.

3.2.4 Evaluation

In the evaluation phase, the *Improved SIR* uses a similar approach as the *Naive SIR* algorithm. However, instead of evaluating the network for each different node type, the solution evaluates the network once and then counts the occurrence of a particular node type from the result. The Algorithm 11 describes the pseudocode of such procedure.

Algorithm 11 Improved SIR Evaluation

```
procedure EVALUATE($edges, N$)
    $nodesTypes \leftarrow evaluateNodes(edges, N)$

    $sNumber \leftarrow 0$
    $iNumber \leftarrow 0$
    $rNumber \leftarrow 0$
    $undefNumber \leftarrow 0$

    for node type in $nodeTypes$ do
        switch node type do
            case S node
                $sNumber$ += 1
            case I node
                $iNumber$ += 1
            case R node
                $rNumber$ += 1
            case Undefined node
                $undefNumber$ += 1
```

The *evaluateNodes* kernel function produces an array of node types which are then counter to determine the state of the network. An extra node type, *undefined*, was introduced to account for cases where the node is not connected to any other node and, thus, its type cannot be inferred from the edge matrix. Moreover, notice, that the node evaluation is performed on the GPU in parallel, while the nodes are counted sequentially on the CPU. We took this approach considering that counting nodes on the GPU would require some sort of synchronisation step and having in mind that the node array is relatively small in size when compared to the edge matrix, the CPU seemed to be a better choice for this part of the computation.

3.2.5 Analysis

The *Improved SIR* algorithm developed upon the *Naive SIR* approach and introduced new solutions that increased the concurrency of the algorithm and potentially paved a way for more parallelism and better utilisation of the GPU hardware. Some preliminary measurements were made of the performance of the two algorithms and the results can be seen in Figure 3.11.

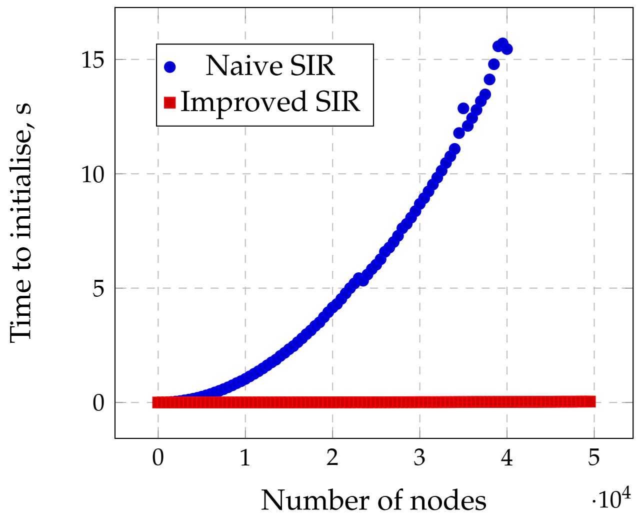


FIGURE 3.11: Naive Parallel SIR and Improved Parallel SIR initialisation phase comparison.
Simulation parameters: $P_{infected}$=0.01, P_{infect}=0.2, $P_{recover}$=0.1

The *initialisation* phase was chosen for the comparison, as it executes most of the procedures used during the simulation process. From the results, we can see that the *Improved SIR* algorithm performs significantly better than the *Naive SIR* equivalent and scales surprisingly well when the network size is increased. A more comprehensive performance analysis of the *Improved SIR* algorithm, in terms of how it compares the sequential SIR implementation, is explored in Chapter 5.

Chapter 4

Implementation

We implemented the solutions using CUDA C programming language [14]. It extends the standard C language by introducing new types of functions, called kernels, that can be executed in parallel by CUDA threads [14]. Furthermore, a subset of C++ language features is supported as well and was used in the implementation to provide a more maintainable structure to the code. The CUDA programming model was designed by *NVIDIA* and the code produced with it can only be executed on *NVIDIA's* graphic cards. Other alternatives such as *OpenCL* were explored as well and a brief summary of why CUDA was chosen is discussed in section 4.1. Moreover, the development process was performed using the *Nsight Eclipse Edition* IDE, which provides an easy to use CUDA source editor, a graphical user interface for debugging heterogeneous applications and a visual profiler with source code correlation for optimising the GPU code performance [16]. The implementation was version controlled using *Git* [4]. A user guide is provided in Appendix A.

4.1 CUDA vs OpenCL

We have identified two development frameworks for use in developing a GPU-based solution. The parallel computing and programming model developed by NVIDIA, called CUDA [13] and the open source alternative, an open standard for parallel programming of heterogeneous systems, OpenCL [6]. We investigated both platforms and some of the pros and cons of each were considered:

OpenCL

Supported by many vendors and works on multiple devices with various architectures, however is not optimised to work efficiently on all of them. Furthermore, due to extensive set of features available, is hard to use and there seems to be a lack of comprehensive documentation and well maintained development tools.

CUDA

Has mature development tools, including a debugger and a profiler. Furthermore, provides a comprehensive documentation and has an active development community. Moreover, seems to be easier to build and integrate and usually can produce a better performance when compare to OpenCL. However, is only compatible with NVIDIA's hardware.

In the end, we decided to use the CUDA development framework. It seems to have a more developed set of tools that can be used out of the box and provides a support infrastructure for new developers in the field of General Purpose GPU computing. Furthermore, it is designed to achieve an optimal performance with NVIDIA's hardware and, given that this project tries to implement an efficient general network simulator, it seemed to be a good platform for trying to push the speed of arbitrary network simulations.

4.2 Code Structure

The code is structured into C++ classes, where each class abstracts a different entity of the solution. An abstract class called *Network* defines a list of virtual methods that each subclass have to implement. These methods are *initialise*, *simulateStep*, *evaluate*, *print*, *printConfig*, *printState*. The first three methods correspond to the simulation stages discussed in the design section, while the other three are used for debugging reasons. By enforcing this, the simulator can call these procedures on any network implementation and perform the simulations without having to worry about the underlying network topology and simulation model. The simulator is represented by the *Simulator* class. It is responsible for simulating the given amount of timesteps and defining the parameters that are used to map the simulations onto the GPU hardware. Finally, the *SIR* class extends the *Network* class and implements the SIR epidemic simulation model.

4.3 Computational Model

The solution uses a heterogeneous computation model where two devices: CPU and GPU, are used. The CUDA development framework provides a way to control both devices during execution. The CPU (refereed to as the host) issues the commands to the GPU (refereed to as the device). The functions that are executed on the device are called kernels. The host can call a kernel using the $<<<$ *blocks*, *threads* $>>>$ notation, where the *blocks* and *threads* variables indicate how the particular computation will be mapped onto the GPU hardware. As explored in Chapter 2, each block of threads is mapped to a Streaming Multiprocessor in the GPU, where all the threads in that block are executed in parallel. The number of threads in a block can be tweaked and is a parameter that is passed to the simulator before the execution. The implementation calculates the number of blocks required for each kernel call depending on the size of the network that is being processed, as portrayed in Figure 4.1.

$$Block_{number} = \frac{Node_{number} + Thread_{number} - 1}{Thread_{number}}$$

FIGURE 4.1: Determining the number of blocks required

The block number is calculated in a way to ensure that a thread is assigned to each element required processing. For example, when the *updateNetwork* kernel function is called in the *Improved SIR* algorithm, the block number is calculated in a way, so that every entry in the edge matrix has an associated thread that will process it. The elements to process are not directly passed to the threads, but rather each thread calculates its index and uses it to retrieve the appropriate element from the global memory. In our case, when the *updateNetwork* procedure is called, every edge calculates its index, as shown in Figure 4.2, and then retrieves a particular edge entry from the edge matrix array in the global memory using that index.

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

FIGURE 4.2: Calculating thread array index

The *blockIdx.x*, *blockDim.x* and *threadIdx.x* variables are initialised by the device for each thread during the execution. The *blockIdx.x* indicates which block the thread is in, *blockDim.x* states the dimension of the block, in our case we only use one dimension, and *threadIdx.x* defines the id of the thread in the block.

Moreover, sometimes the amount of elements that need to be processed does not evenly map to the number of blocks and threads available. In those cases, an extra block is allocated. As a consequence, there might be threads who have an out of bounds index, as there might be more threads than there are elements to process. Therefore, to ensure that no illegal memory accesses occur, each thread, before accessing the array in the global memory, checks if its index is lower than the total number of elements that is currently being processed.

4.4 Naive SIR algorithm

The *Naive SIR* algorithm performs most of the computations on the CPU and uses the GPU for propagation of changes in the edge matrix, as described in Chapter 3. The network is represented by a contiguous array in memory where each entry is defined by an integer value. During the update procedure, to potentially speed up the computations, the edge values were hardcoded and are updated using a switch statement.

Figure 4.3 shows a code example from the *updateSEdges* procedure. As discussed in Chapter 3, the *Naive SIR* algorithm only propagates a single node change in the network at a time. Therefore, each thread has to determine if their edge is in the row or the column of the updated node. In the code snippet, the thread checks if its row position in the edge matrix matches the index of the node that is being updated and, if so, updates the edge appropriately. As noted in Chapter 3, the edge values here have been chosen arbitrarily.

```
if (index / node_number == t_edge / node_number) {
    switch(network[index]) {
        case SI_EDGE:
            network[index] = II_EDGE;
            break;

        case SR_EDGE:
            network[index] = IR_EDGE;
            break;

        case SS_EDGE:
            network[index] = IS_EDGE;
            break;
    }
}
```

FIGURE 4.3: Code snippet from the *updateSEdges* kernel

Looking back, we can reason why such edge update procedure, might not have been the most optimal to execute with the GPU computing model. As discussed in Chapter 2, any thread divergence from the normal execution path will significantly reduce the performance and hardware utilisation [14]. In this case, each time the edges are updated, the threads will diverge multiple times. The threads whose edges are not part of the node that is being updated, will stall throughout the whole update procedure. While, the rest will be updated by diverging and executing a different case of the switch statement. In a worst case scenario, all thread executions will be serialised and executed sequentially, giving a $1/\textit{Number of edges}$ of the peak performance. This might explain why the *Naive SIR* performed poorly and did not scale at all when some preliminary measurements were made (Chapter 3).

4.5 Improved SIR algorithm

The *Improved SIR* algorithm introduced many implementation level improvements that were designed to increase the utilisation of the GPU and leave more space for the potential parallelism to take place. First of all, to reduce the memory footprint, the edge is defined using a char, instead of an integer. This reduces the amount of memory required by at least a half. Furthermore, during the simulation phase, the algorithm pre-computes which nodes will be updated and, in one go, propagates the changes in the network for all edges. An example code snippet for this procedure can be seen in Figure 4.4.

```
matrix_position position = calc_matrix_position(index, node_number);
int edge_type = int(edges[index]);
int map_index = edge_type + d_node_uind[position.row]
                + 2 * d_node_uind[position.column];

if (edge_type != NO_EDGE) {
        edges[index] = edge_map[map_index];

        prob_set new_prob = d_prob_map[map_index];

        if (new_prob.row_node != NO_UPDATE) {
                d_node_ucond[position.row] = new_prob.row_node;
        }

        if (new_prob.col_node != NO_UPDATE) {
                d_node_ucond[position.column] = new_prob.col_node;
        }
}
```

FIGURE 4.4: Code snippet from the *updateNetwork* kernel

As discussed in Chapter 3, the *Improved SIR* algorithm updates the edges and the probabilities of the nodes. During the edge update step, it retrieves the new edge value without producing any divergence in the thread execution flow. Instead of using a switch statement, as the *Naive SIR* does, the *Improved SIR* algorithm uses a static edge map array (Figure 4.5).

```
  char edge_map[24];
  edge_map[0] = 0;
  edge_map[1] = SS_EDGE;
  edge_map[2] = IS_EDGE;
  edge_map[3] = SI_EDGE;
  edge_map[4] = II_EDGE;
  edge_map[5] = RI_EDGE;
  edge_map[6] = IR_EDGE;
  edge_map[7] = RR_EDGE;
  edge_map[8] = SI_EDGE;
  edge_map[9] = II_EDGE;

        /* ... */
```

FIGURE 4.5: Edge map defined as a static array in the *updateNetwork* kernel

The index for the appropriate array entry is calculated by taking the current edge type as the initial index and shifting it: by one, if the row node is being updated; by two, if the column node is updated and by three, in case both nodes are updated. This way, we can ensure that no divergence of the flow will be present, as all threads will be performing the same steps: reading from memory, performing simple arithmetic calculations, looking up a value in a local array and storing a value back into memory. Furthermore, the edge map is implemented as an array, in order to speed up the edge mapping process. Moreover, it seems that a local array might potentially be stored by the GPU's runtime system in the registers of each Streaming Multiprocessors core during execution [9].

4.6 Testing

Both the *Naive SIR* and the *Improved SIR* implementations were tested in a variety of scenarios during the development process. Due to the stochastic nature of the SIR epidemic model, it was hard to determine the correctness of the implementation. Nonetheless, for the *Naive SIR* solution we wrote multiple tests to check if the simulation is working correctly. The tests made sure that the network converges over time to a particular node type or stays the same depending on the input probabilities.

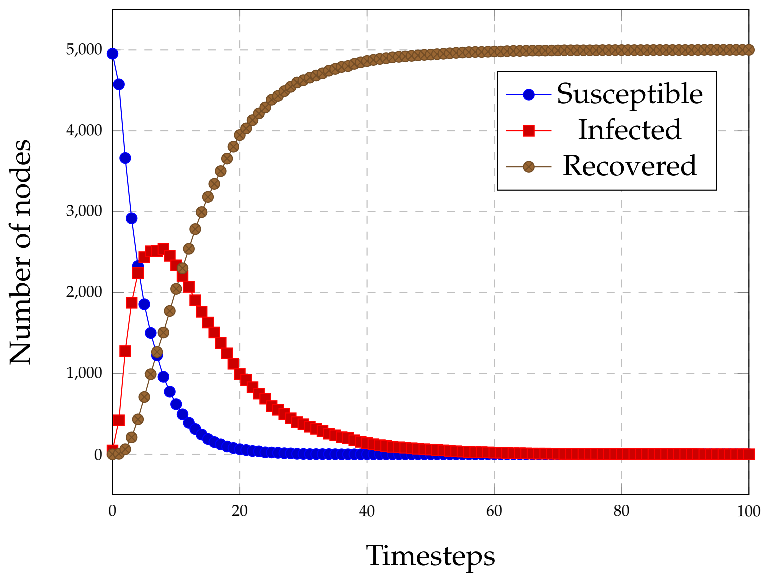


FIGURE 4.6: Improved SIR network changes over time.
Simulation parameters: P_{edge}=0.01, $P_{infected}$=0.01, P_{infect}=0.2, $P_{recover}$=0.1, 5000 *nodes*

We took an extra step to ensure that the *Improved SIR* algorithm is behaving according to the model. The node type changes over time were measured and are plotted in Figure 4.6. We can see that network changes in manner which one would expect from an SIR epidemic simulation. The susceptible nodes decrease over time, as more and more of them become infected. Once the infected nodes start to recover, the infection rate drops and the nodes in the network converge to either being recovered or susceptible. Furthermore, the epidemic dynamics observed here seem to resemble the results obtained by Dobson [3] in his blog on *Complex networks, complex processes*.

Chapter 5

Evaluation

To gain insight into the possible performance and scalability of our implementation and determine how it performs when compared to other similar solutions, we devised and performed a set of experiments. During preliminary measurements, the *Naive SIR* approach performed extremely poorly and did not scale at all. Therefore, no further analysis of it was explored and the *Improved SIR* algorithm was used as the final solution and is referred here as *Parallel SIR*. Furthermore, a sequential SIR simulator implemented in Python by Simon Dobson [3] was adapted and is used in the experiments for comparison.

5.1 Experimental Methodology

The experiments were performed on a Linux machine equipped with NVIDIA GeForce GTX 1060 graphics card (6Gb of VRAM), Intel Core i5-6500 (3.20GHz) CPU and 8Gb of RAM. Some experiments required more RAM and, thus, another Linux machine, equipped with Intel Xeon E5-2640 (2.50GHz) CPU and 132Gb of RAM, was used as well. To perform the measurements, various C++ benchmarking libraries such as: *Nonius* [12] and Google's *benchmark* [5], were considered. In the end, it seemed that most of the functionality provided by such frameworks would be unnecessary in our case. Therefore, we implemented a benchmarking script that was tweaked during the experimental process to measure the execution time of various simulation scenarios using the *Parallel SIR* solution. Furthermore, a script in Python was also written to perform equivalent kind of measurements using the sequential SIR implementation. Each measurement was made 5 times and an average was calculated and taken as the final result.

5.2 Results

The initial experiments tried to determine how the block size, used during the simulations, affects the performance and what would be the value to use in order to achieve peak performance. Four different block sizes were considered: 128, 256, 512, 1024 and the results can be seen in Figure 5.1.

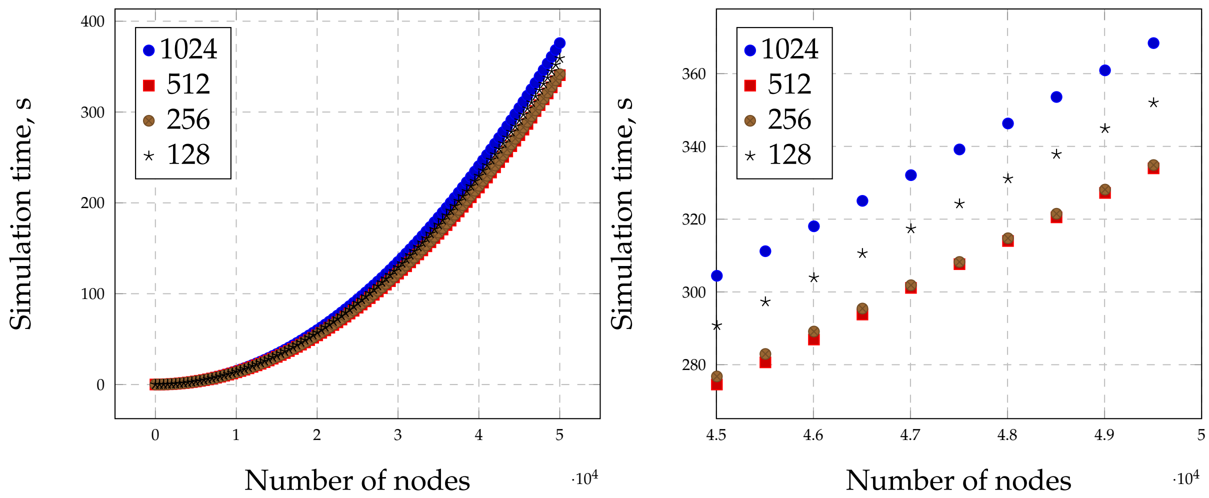


FIGURE 5.1: Parallel SIR simulator performance with varied block sizes
Simulation parameters: $P_{infected}$=0.01, P_{infect}=0.2, $P_{recover}$=0.1, 1000 *timesteps*

From the graph on the left, we can see that the overall performance, as the block size varied, did not change dramatically. However, it was not exactly equivalent either. The graph on the right zooms into the end of the plots and we can see that block size of 512 seems to produce the best performance, as the number of nodes in the network increases.

Using the optimal block size, we compared the performance of *Parallel SIR* with a sequential single core CPU implementation for varying degrees of connectivity in the network. Figure 5.2 shows a performance comparison for a fully connected network.

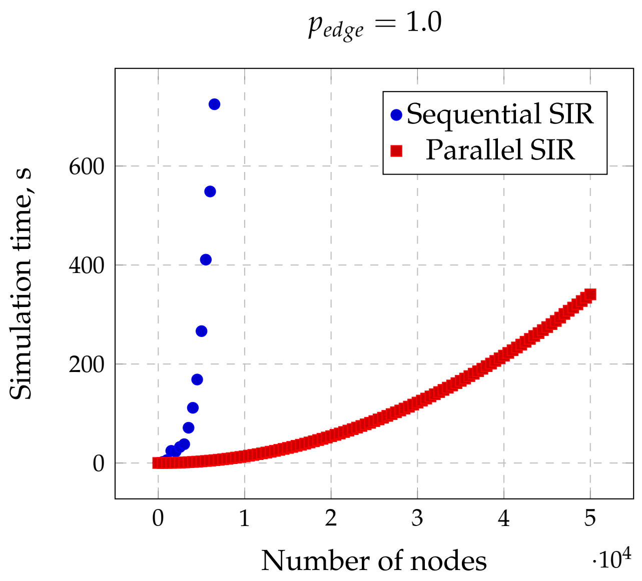


FIGURE 5.2: Sequential and Parallel SIR simulator performance comparison with a fully connected network
Simulation parameters: $P_{infected}$=0.01, P_{infect}=0.2, $P_{recover}$=0.1, 1000 *timesteps*, 512 *threads per block*

From the results, we can see that *Parallel SIR* approach performs and scales significantly better. After calculating the average execution time of each plot, we estimated the *Parallel SIR* solution to be about 87.4 times faster on average than the sequential equivalent in this simulation scenario. Furthermore, we investigated how the performance of both solutions changes with varying network degrees of connectivity. The results can be seen in Figure 5.3.

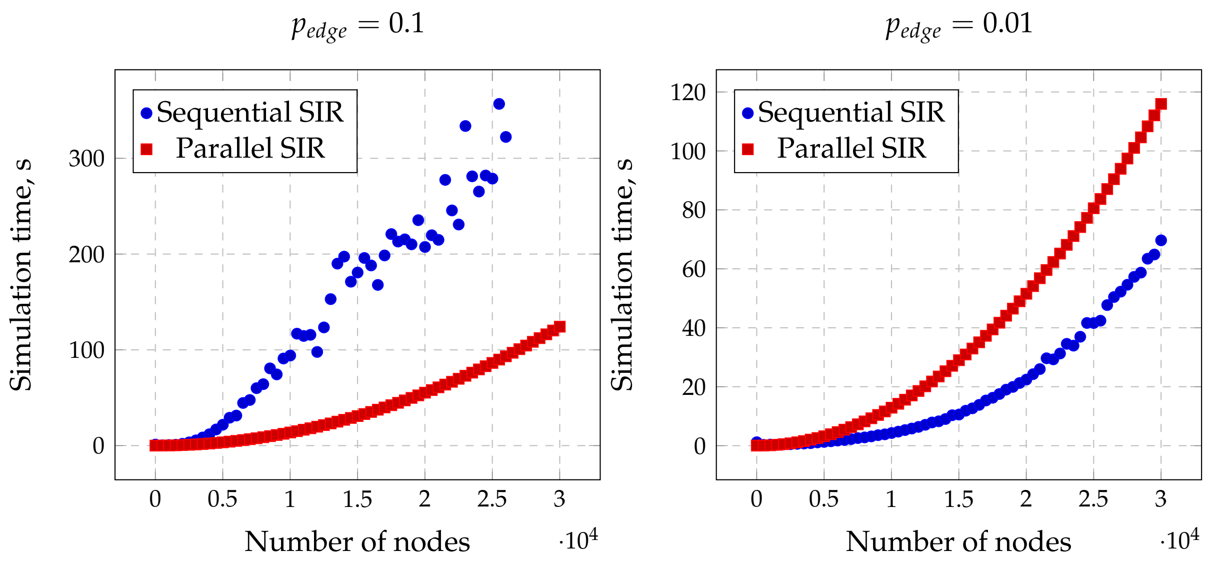


FIGURE 5.3: Sequential and Parallel SIR simulator performance comparison with varied degrees of connectivity
Simulation parameters: $P_{infected}$=0.01, P_{infect}=0.2, $P_{recover}$=0.1, 1000 *timesteps*, 512 *threads per block*

It seems that as we decrease the probability of an edge in the network, the sequential SIR implementation improves its performance relative to the Parallel SIR solution and even achieves faster simulation times then only 1% of the network is connected. Moreover, we can take a closer look at how the performance of the *Improved SIR* solution changes with varied network topologies (Figure 5.4).

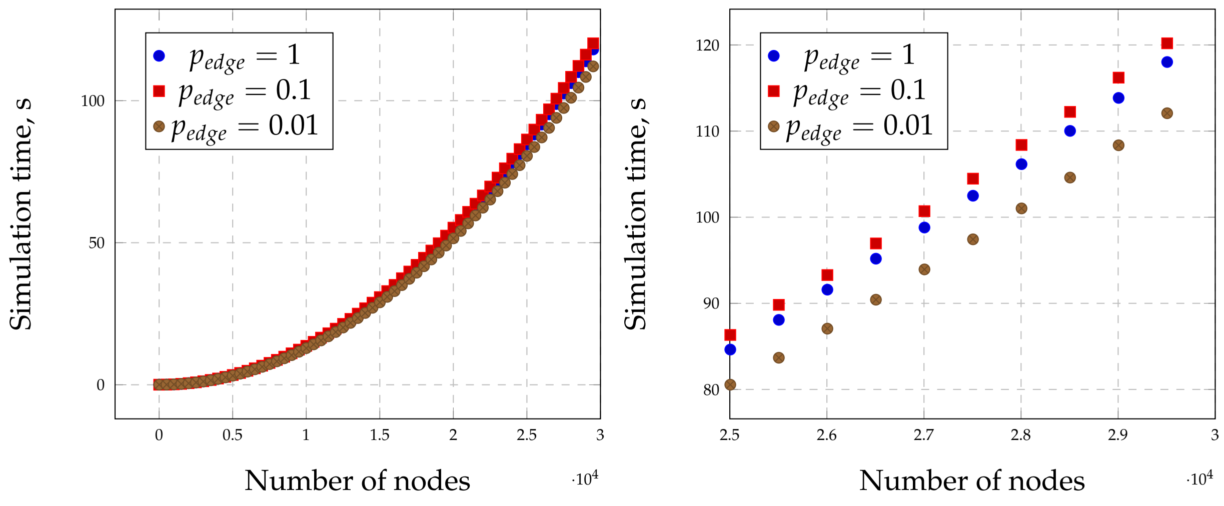


FIGURE 5.4: Parallel SIR simulator performance with varied network topologies
Simulation parameters: $P_{infected}$=0.01, P_{infect}=0.2, $P_{recover}$=0.1, 1000 *timesteps*

The *Parallel SIR* solution seems to be more stable as the network topology changes when compared to its sequential equivalent. The simulation time required as the number of nodes in the network increases is almost identical for all three degrees of connectivity. Only if we zoom in, as portrayed in the graph on the right (Figure 5.4), we can see that there is a slight improvement in the simulation time required as we decrease the connectivity of the network.

Furthermore, we designed some experiments to determine how the number of timesteps affects the performance of both the sequential and parallel SIR simulators. The results for a fully connected network are portrayed in Figure 5.5.

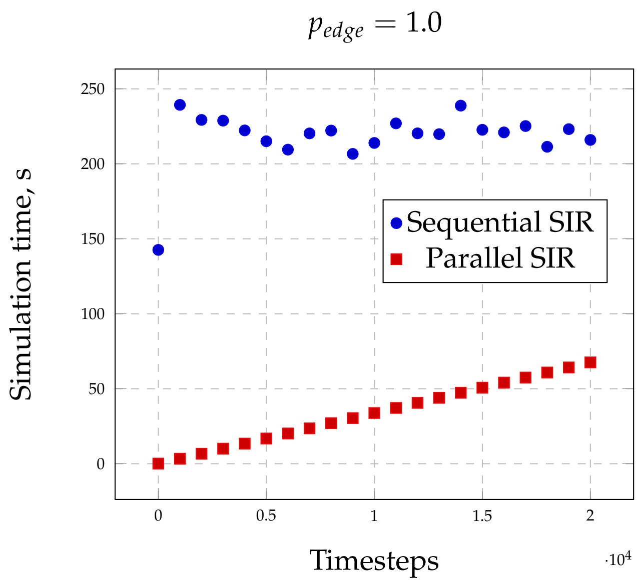


FIGURE 5.5: Sequential and Parallel SIR simulator performance comparison with varying number of timesteps and a fully connected network
Simulation parameters: $P_{infected}$=0.01, P_{infect}=0.2, $P_{recover}$=0.1, 5000 *nodes*

The *Parallel SIR* solution seems to scale in a linear manner as the number of timesteps increases. In contrast, the *Sequential SIR* simulator make a jump initially and then it seems to be fluctuating around a certain constant. To investigate this further, we performed measurements with varied network topologies (Figure 5.6).

Throughout different network topologies, the *Parallel SIR* solution seems to be stably scaling in a linear manner. While, the sequential implementation performs in a fluctuating constant time, although the constant decreases as the network becomes less connected.

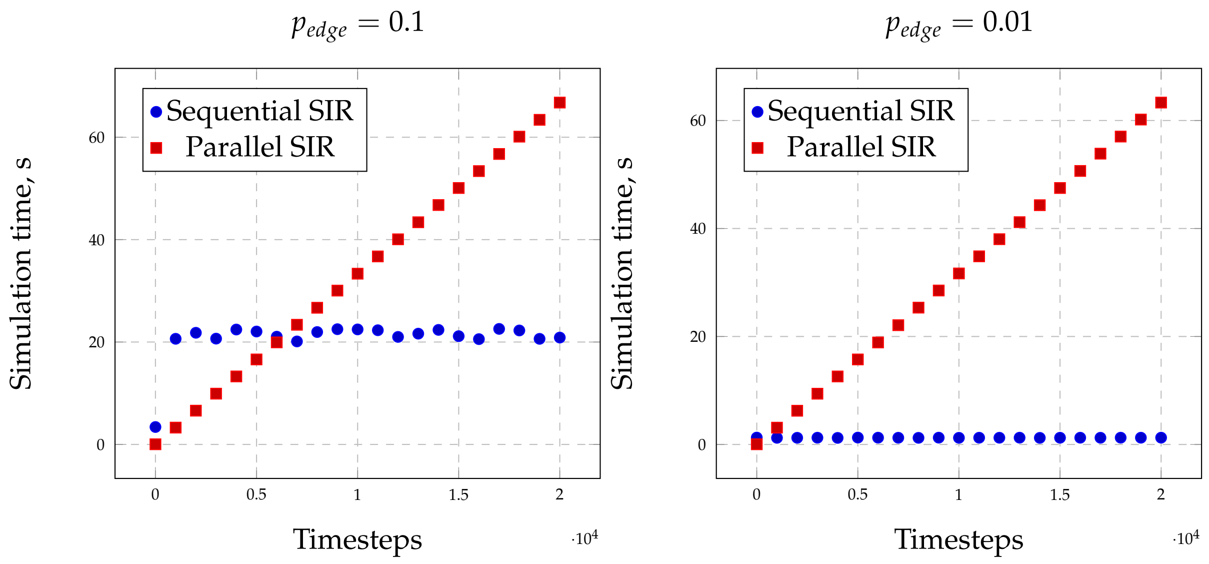


FIGURE 5.6: Sequential and Parallel SIR simulator performance comparison with varying number of timesteps and varying network topologies
Simulation parameters: $P_{infected}$=0.01, P_{infect}=0.2, $P_{recover}$=0.1, 5000 *nodes*

5.3 Discussion

The parallel SIR solution seems to be faster in simulating the SIR epidemic model when compared to a sequential equivalent with highly connected network topologies. Although, it should be noted that the sequential implementation is implemented in Python, while the parallel one is written in C. Python is an interpreted language and is executed slower than C, which is compiled. Therefore, the comparison is not exactly valid and ideally a sequential C implementation would be used instead. Nonetheless, the language difference, in theory, should not significantly affect the scaling factor of the solutions, if we assume that the execution between the two languages differs by a constant factor. Our implementation scales significantly better when simulating a fully connected network. If we reduce the degree of connective by a factor of ten in the network, we can see that the sequential implementation improves significantly, while the parallel one barely improves at all. We can explain this by considering how the two implementations represent the network in memory. The *Parallel SIR* simulator uses an adjacency matrix, therefore no matter how connected the network is, all possible edges are still stored in the network and are updated every timestep. However, the sequential Python simulator, most likely uses an adjacency list and thus only stores the actual edges. Therefore, then we reduce the connectivity of a network, the sequential implementation needs to check fever memory locations and thus performs faster.

Furthermore, when comparisons were made with regards to the varying simulation times, we observed a strange behaviour of the sequential SIR solution. The parallel simulator scaled linearly as the number of timesteps in the simulation increased for multiple degrees of connectivity. However, the simulation time of the sequential solution did not change and remained constant, as we increased the number of timesteps. These results seem to be counter-intuitive, as by increasing the number of simulations, we would expect the program to require more computations to account for the extra timesteps. Therefore, we believe, it is possible that the sequential simulator does not simulate every discrete time step and potentially skip some intervals where no changes happen in the network.

5.4 Analysis

After achieving promising evaluation results with the *Parallel SIR* algorithm, we performed an analysis to see if it could be extended to work with a wider range of problems. It seems that, the edge and probability maps used in the simulation could be generated based on the inputs that the user provides. Although, the current memory representation only allows 255 unique edge types, as a single edge is represented by a char variable. Furthermore, the underlying adjacency matrix structure allows to define not just simple networks, but also directed ones and it is possible to include self-loops. Therefore, it seems that the parallel SIR simulator could be used to efficiently simulate custom models and networks. However, certain limitations are present. Networks with a huge number of nodes will require a significant amount of memory. For example, a fully connected network with 100000 nodes will have about 5 billion edges and, with our current memory model, all of those edges will be stored in the memory of the GPU. Therefore, in our example, we would need about 5Gb of VRAM, which is still achievable. However, the memory required expands in a quadratic manner as the number of nodes increases and, thus, with the currently available hardware it might not be feasible to simulate significantly large networks.

Nonetheless, many improvements could be made to the current implementation to make it more efficient and scalable. The solution could be upgraded to work with multiple GPUs, that way solving the memory problem by allowing the user to scale horizontally. Furthermore, techniques could be investigated to compress the way edges are currently stored, especially if the network is loosely connected and the edge matrix is sparse. Moreover, the simulator could skip timesteps where no change happens in the network and reduce the simulation time as a result. This might be especially useful for models where updates in the network happen in short time intervals and for the rest of the time, the network stays unchanged.

Chapter 6

Conclusions

We developed a GPU based parallel network simulator for the SIR epidemic model, with hopes to generalise it, in case of successful results. The first approach, named *Naive Parallel SIR*, was inefficient and did not scale well with increasing network size. Simple analysis indicated that the approach was using implementation strategies that were not optimal in the GPU computing model and, therefore, low hardware utilisation was achieved. The second approach, named *Improved SIR Network*, improved upon the first and managed to achieve more parallelism by redesigning how the changes in the network were propagated during each timestep of the simulation and introducing low level optimisation techniques to improve performance of the execution threads. When compared to a sequential simulator, in certain simulation scenarios, it achieved an average execution time speedup of 87.4 times. Furthermore, it appeared to perform stably with varying network topologies and execution parameters. On the other hand, we observed some limitations as well. The current network representation uses an adjacency matrix to store the edges and in a scenarios where the network is loosely connected, the matrix will be sparse and multiple entries will be unnecessarily stored and processed. Furthermore, such network storage model does not scale well in terms of memory space required with increasing network size. Nonetheless, we introduced a list of possible improvements that could be made to address the current issues. In the end, we successfully investigated the applications of graphic processor units on network simulations and produced a prototype that could potentially be extended to simulate arbitrary networks with custom simulation models.

In this work, our analysis is very limited and, in the future, we would like to investigate in detail how our solution performs with more varied simulation models and networks. Furthermore, the current implementation has significant limitations and many improvements could be made to make it more scalable and efficient. Finally, we would like to develop our solution into a library that could be used to efficiently simulate complex models on arbitrary networks using GPUs, without having the user to worry about how the simulations are parallelised and performed.

Appendix A

User guide

The project source is divided into four directories: *Results*, *Naive Parallel SIR*, *Improved Parallel SIR* and *Sequential SIR*. The *Results* directory contains all the measurements made during the implementation and evaluation processes. The other directories, contain source code for the appropriate algorithms.

To run any of the parallel simulators, go to their source folders then to *Release* directory and type in the terminal:

```
make clean
make
./network-simulator
```

The simulation parameters can be tweaked by editing the *benchmark.h* file.

For the sequential simulator, type in the terminal in its source folder:

```
python run_sim.py pInfected pInfect pRecover pEdge nodes timesteps
```

The simulation parameters have to be specified as input variables with the terminal command.

Bibliography

[1] Darius Bakunas-Milanowski et al. "Efficient Algorithms for Stream Compaction on GPUs". In: *International Journal of Networking and Computing* 7.2 (2017), 208–226.

[2] Ben Romdhanne Bilel et al. "Scalability demonstration of a Large Scale GPU-based Network simulator". In: *SimuTools '13, Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques* (2013), pp. 139–141.

[3] Simon Dobson. *Complex networks, complex processes*. URL: https://nbviewer.jupyter.org/github/simoninireland/cncp/blob/master/index.ipynb.

[4] Git. *Version Control System*. URL: https://git-scm.com/.

[5] Google. *A microbenchmark support library*. URL: https://github.com/google/benchmark.

[6] Kronos. *OpenCL*. URL: https://www.khronos.org/opencl/.

[7] Encyclopedia of Mathematics. *Arithmetic series*. URL: https://www.encyclopediaofmath.org/index.php/Arithmetic_series.

[8] Robert T. McGibbon. *Stackoverflow*. URL: https://stackoverflow.com/questions/27086195/linear-index-upper-triangular-matrix.

[9] Maxim Milakov. *Fast Dynamic Indexing of Private Arrays in CUDA*. URL: https://devblogs.nvidia.com/fast-dynamic-indexing-private-arrays-cuda/.

[10] M. E. J. Newman. *Networks An Introduction*. Great Clarendon Street, Oxford: Oxford University Press, 1993.

[11] M. E. J. Newman. "Spread of epidemic disease on networks". In: *PHYSICAL REVIEW E* 66 (2002). DOI: https://doi.org/10.1103/PhysRevE.66.016128.

[12] Nonius. *A C++ micro-benchmarking framework*. URL: https://nonius.io/.

[13] NVIDIA. *CUDA*. URL: https://developer.nvidia.com/cuda-zone.

[14] NVIDIA. *CUDA C Programming Guide 4.2*. URL: https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[15] NVIDIA. *Developer Forum*. URL: https://devtalk.nvidia.com/.

[16] NVIDIA. *Nsight Eclipse Edition*. URL: http://docs.nvidia.com/cuda/nsight-eclipse-edition-getting-started-guide/index.html.

[17] Yanxiang Zhou et al. "GPU accelerated biochemical network simulation". In: *Bioinformatics* 27.6 (2011), pp. 874–876. DOI: 10.1093/bioinformatics/btr015.

[18] Matija Šošić and Mile Šikić. "CUDA implementation of the algorithm for simulating the epidemic spreading over large networks". In: *MIPRO, 2012 Proceedings of the 35th International Convention* (2012).