Delft University of Technology
Master's Thesis in Computer Science

# Hardware acceleration of simulations of distributed systems

**Dániel Turi**

embedded software

TU Delft Delft University of Technology

# Hardware acceleration of simulations of distributed systems

Master's Thesis in Embedded Systems

Embedded Software Section
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Dániel Turi
D.Turi@student.tudelft.nl

29th May 2013

**Author**
Dániel Turi (D.Turi@student.tudelft.nl)
**Title**
Hardware acceleration of simulations of distributed systems
**MSc presentation**
29th May 2013

**Graduation Committee**

| | |
|---|---|
| Dr. K. G. Langendoen | Delft University of Technology |
| Dr. S. O. Dulman | Delft University of Technology |
| Dr. A. Iosup | Delft University of Technology |
| A. L. Varbanescu | Delft University of Technology |

**Abstract**

In recent years the study of complex systems gained prominence. Since it is usually difficult to use classical mathematic models to understand these systems, scientists and engineers have to resort to simulations. Currently programming languages like CUDA C and OpenCL are available to run large scale simulations on a GPU architecture. Alternatively simulations can be executed on agent-based simulators such as NetLogo.
The former enables the writing of highly performant programs whereas the latter offers a simple language and execution model which is accessible to people with little programming experience.
The purpose of this thesis is to devise a proof-of-concept simulator called *CudaSimulator* which attempts to create a middleground between NetLogo and CUDA, which on the one hand retains the simplicity of NetLogo and on the other hand executes simulations on a GPU architecture.

Apart from the general context this thesis is motivated by a specific demand. The Snowdrop project at the Embedded Software Group, TU Delft, aims to find algorithms that are written in the NetLogo programming language and exhibit certain emergent behavior. In order to find such algorithms Genetic Programming is used.
Since a Genetic Programming framework has to simulate and evaluate a lot of algorithms and this process is usually lengthy, it could be accelerated on a GPU architecture.

The *CudaSimulator* is therefore developed in such a way that it could serve as a backend to the Genetic Programming framework.

The thesis includes an evaluation of the simulator which shows the situations in which the *CudaSimulator* is more feasible than alternative solutions.

# Preface

This thesis is the last step towards my degree and starting a life as an engineer. On the one hand it makes me sad that the student years with all the joy and freedom are over. On the other hand I am eager to move on and explore new things.

I would like to thank my advisors Ana Varbanescu, Stefan Dulman and Andrei Pruteanu for helping me with ideas when I was stuck, warning me of pitfalls before falling into them, and most of all, for their infinite patience. I would like to thank my fellow students in the Snowdrop project, who were always keen to share ideas with me, whose progress motivated me and with whom I had a great time. Finally, I would like to thank my parents who supported me no matter what.

Dániel Turi

Delft, The Netherlands
29th May 2013

# Contents

# Chapter 1

# Introduction

The study of complex systems is a novel scientific field dedicated to understanding how parts of a system interact to generate (new) collective behaviors, and how the whole system interacts with its environment. It is often the case in these systems, that indirect, unpredictable effects appear when different action or events happen. Examples of complex systems are found in a wide range of domains, from social sciences to chemistry and biology, or from economy to technology. To study complex systems, two classes of methods are used: analytical methods, where the behavior of system is eventually modeled by in-depth analysis of patterns, and the simulation-based methods, where systems are described and modeled in terms of actions, and simulated to reproduce and eventually understand their behavior.

A widely used simulation-based method to study complex systems is *Agent-based Modeling* (ABM). In ABM, the system is modeled as a set of agents that are interacting with each other. Depending on the rules of this interaction, the agent-to-agent communications, and the interaction with the environment, the complex system evolves. Modeling these agents and simulating all these interactions using computer-based tools allows the observer to follow the evolution of the system.

There are many programs that support this modeling method. One of the most popular is *NetLogo* [37]. Its popularity is due to its ease of use and the expressiveness of the language (a variant of Logo [37]). NetLogo is accessible to non-IT professionals (e.g., biologists, economists, or chemists) to develop models, but it is often used by IT professionals as well, for fast-prototyping of distributed algorithms. Programming is helped by a large set of examples from various fields.

The main drawback of *NetLogo* (and of most other ABM systems) is that these complex simulations can be very slow. The simulation engine of NetLogo is a sequential one (i.e., it runs in a single-thread), which heavily underutilizes the processing reasources of the parallel processors we have today in all our machines. To make use of these parallel computing resources,

the inherent parallelism in ABM – there is a multitude of agents executing the same program – needs to be exploited. In fact, attempts are currently being made to use both clusters [9] and *Graphics Processing Units* (GPUs) [8] [13] [30] [40] for simulating agent-based systems. In this context, our work focuses on exploring the benefits of using parallel architectures for speeding-up agent based simulations. Specifically, we want to focus on the use of GPUs, which are small, cheap, versatile, and power-efficient platforms that provide very high computational throughput.

Until recently, writing any non-graphics application on a GPU was extremely laborious, since only the computations disguised as graphical operations could be carried out. However, when GPUs have started to significantly overpower CPUs in terms of computation throughput, *NVidia* (one of the two largest producers of GPUs) was the first to recognize the need for a GPU framework that supports general computation. The result was the *Compute Unified Device Architecture* programming model (i.e., CUDA). In CUDA, the underlying architecture of the GPU is exposed as extensions to general purpose languages such as C/C++ and FORTRAN. General purpose applications suddenly became easier to program on GPUs.

However, programming in CUDA is still by far not as accessible to users as *NetLogo*, since it is necessary to learn a relatively complicated programming language, the prototyping procedure is tedious and error-prone, and debugging can be very difficult. Thus, we believe the right way to make GPUs a feasible option for agent-based modeling is to combine the ease of use of systems like NetLogo (the user interface/language for building the models) with novel, highly parallel simulation engines that allow the simulations to use the parallel hardware to gain speed.

## 1.1 Problem statement

The main goal of this work is to efficiently combine the advantages of *NetLogo* (for user interaction) and CUDA (for the simulation engine) to improve the state of the art in parallel agent-based modeling.

Our approach is based on a framework called *CudaSimulator*, which enables the *NetLogo* engine to use the GPU architecture and thus to exploit its massive parallelism and compute power. The framework consists of two parts: (1) a *translator* built inside the NetLogo engine, which transforms NetLogo models into custom-designed *bytecode*, and (2) a *virtual machine* that interprets and executes this *bytecode* on the GPU. In this thesis, we present the design, implementation, and validation of the *CudaSimulator*. Furthermore, to understand the impact of such a framework in the field of agent-based modeling, we also analyze its limitations and efficiency, especially compared against the "'pure"' solutions - i.e., original NetLogo and CUDA-only solutions.

### 1.1.1 Research questions

This thesis answers the following research questions:

1. **Can *NetLogo* models be efficiently translated to CUDA?**
   We will summarize the basic features of both *NetLogo* and CUDA, and analyze the mismatches between the two platforms. We will show that the translation needs to be done at the compiler AST level, where the levels of abstraction of the two models are compatible.

2. **How should *NetLogo* be changed to accommodate the use of GPU architecture?**
   We will discuss the internal structure of the *NetLogo* engine and analyze the parts that need to be adapted for enabling GPU execution. We will further design a subsystem to support GPU execution and present a prototype that can be embedded inside the original software with very low redundancy and modification. We call this prototype *CudaSimulator*. We will further analyze the restrictions that need to be imposed on the NetLogo modeling to allow for a simple and efficient implementation of the CudaSimulator, and we will propose future research directions for relaxing these constraints.

3. **What are the capabilities and limitations of our CudaSimulator?**
   We will examine the capabilities of CudaSimulator by implementing a set of three representative test algorithms. We will analyze the size constraints for *NetLogo* models running on a GPU using our CudaSimulator, and briefly discuss the interesting scenario of the MetaCompiler, in which batch processing of simulations can be significantly improved by our CudaSimulator. Last, but not least, we will include a brief analysis of programming patterns that result in high performance loss, and show how to avoid them and write models that utilize the computational power of the GPU.

### 1.1.2 Thesis contributions

While answering these research questions, we make the following contributions:

1. We provide an alternative compilation and execution path for NetLogo simulations.

2. We design and prototype a GPU back-end for executing NetLogo simulations. Our parallelization choice allows for a high-thorughput solution, and focuses on batch processing of multiple simulations.

3. We analyze the capabilities and limitations of such a back-end in terms of modeling, simulation size, throughput, and latency.

4. We discuss alternative implementations that could alleviate some of these limitations.

## 1.2 Thesis organization

The structure of this thesis is as follows: In Chapter 2 we give an overview of the literature research of the relevant work. In Chapter 3 we investigate the main features after which we give an overview of the CUDA architecture. Finally we draw the conclusions on how the main functionalities of *NetLogo* could be mapped on the GPU, which results in the basic design of the system. In Chapter 4 we describe the software part that translates the *NetLogo* model into a *bytecode* that can be run on the GPU. In Chapter 5 we describe the *virtual machine* that is running the above mentioned *bytecode* and therefore executing the translated model on the GPU. In Chapter 6 we show the measurements of three test algorithms. We conclude this work in Chapter 7 in which we briefly summarize the thesis and give and outlook of possible future work.

# Chapter 2

# Related work

In this chapter we are aiming to give an overview of relevant work done by others, in order to understand the context and alternatives of the *CudaSimulator* framework.

In Section 2.1 we examine the usage of Agent-Based Simulations (ABMs) to research distributed algorithms. In most cases, a many agents are necessary for emergent phenomena to arise. Therefore, powerful computers are needed or else simulation can be a lengthy process. However the inherent parallelism in the models (e.g. agents are behaving largely independently with little interaction) can be utilized to be simulated with parallel architectures – e.g. computation clusters or GPUs. The possible ways to accelerate ABMs will be investigated in Section 2.2. In Section 2.3 we will give an overview of the Snowdrop project and how this thesis is integrated in it. Finally we show in Section 2.4 what conclusions were drawn from the available literature.

## 2.1 Agent-based Modeling

Recently natural [33], engineering [12] and social sciences [41] became interested in the emergent behavior of large-scale complex systems. One of the ways to study these systems is to simulate them using *Agent-based modeling* [6]. This method is usually simpler and in many cases yields better results than e.g. differential equations [32].

To simplify the development process of models Agent-based simulators were created. One of the most widely used Agent-based simulators is *NetLogo* [37]. Its popularity is due to the simple yet powerful language (a derivative of *Logo*), and a simple and intuitive GUI. It was conceived for educational purposes (both high-school [35] [20] and undergraduate [10]), but researchers from non-IT fields appreciate the ease of use as well, since it allows for rapid

development [34]. Figure 2.1 shows *NetLogo* running an Agent-based simulation.
There are other simulators as well. *Swarm* [24] is a programming library aimed at developing ABMs. It is also one of the oldest ABM framework, which partly inspired *NetLogo*. *Mason* [23] is a simulator with a GUI, however – as opposed to *NetLogo* – its language is based on *Java*. *Repast Simphony* [27] is a multi-language ABM simulator, with functionality similar to *NetLogo*'s or *Mason*'s – the supported languages are *Java*, *Groovy*, *ReLogo* and *Flowchart*. *Repast HPC* [9] is developed by the same group which develops *Simphony* with the intent to support computer clusters as well.

We chose *NetLogo* due to its simple language and its source code has been published under GPL [14]. Due to the popularity of *NetLogo* the findings of this thesis has the potential to make an impact in the above mentioned fields.

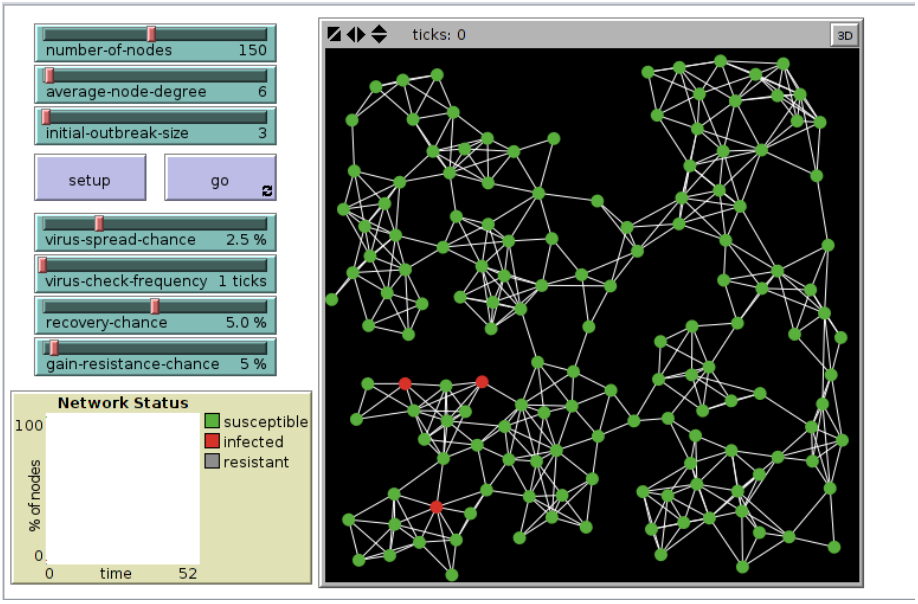


Figure 2.1: NetLogo running an Agent-based simulation

## 2.2 Acceleration of ABM Simulations

In order to leverage the inherent parallelism between agents, running the simulation on parallel architectures has been proposed before. Currently there are two options available to researchers: computation clusters, and GPUs.
*Repast HPC* [9] supports computation on clusters (large number of networked computers) with a programming library that is based on C++.

It includes mechanisms which take care of placing and moving the agents within the cluster, synchronizing their behavior and facilitating data exchange among them in a transparent manner.

Recently GPUs have gotten in the focus of attention since they are capable of running highly parallel programs as well, but they are cheaper than a computation cluster and therefore more accessible.

For a long time GPUs have been used to take over geometric computations from the CPU in order to generate 3D graphics. Since graphical computations can usually be deconstructed into smaller and data-parallel computations, the architecture of the GPUs evolved towards special architecture which is largely different from that of a regular CPU. In the case of a CPU there are few powerful cores which are designed to be able to carry out many types of programs (parallel or serial with intense branching). In contrast to that a typical GPU architecture consists of hundreds or thousands of less powerful cores designed to be very fast for simple applications. GPUs are designed for highly parallel workloads.

D'Souza et al. claimed to be able to run over a million agents on a GPU [11] cleverly using the Graphics API. However writing non-graphics application for these devices via the highly specialized graphics API was tedious or even impossible in some cases.

The GPU manufacturer *Nvidia* recognized this market gap and created the first General Purpose GPU (GPGPU) device and CUDA which is a non-graphic, general purpose programming framework [17] [28]. An extension of the C/C++ language [1], it allows programmers to control GPU specific functionalities in a familiar way and provides the toolchain that includes a compiler, a debugger and a profiler.

The relative simplicity of writing CUDA programs enabled scientists from a wide range of fields to develop their simulations, e.g. in biology [8] [13], in traffic engineering [30] and molecular chemistry [40].

The main drawback of the above examples is that it is still bug-prone and time-consuming to write individual simulations in CUDA as opposed to the ease of development in an ABM simulator. To simplify development while utilizing the power of CUDA, *HOOMD-Blue* [4] [2] was proposed. Designed with the intention to simplify model development, *HOOMD-Blue* can be programmed with *python* scripts. However, this solution focuses on a very special case of models – simulating many particle systems in potential fields – and therefore its usage is limited to the molecular chemistry community.

As we have seen, it is already possible to enable GPUs to develop individual simulations or specialized ABM simulators, but the development of more general simulators remains an active area of research [31]. This thesis is an effort to run *NetLogo* simulations on the GPU and try to create a simulator capable of running more general simulations. This also coincides with the intention of *NetLogo*'s developers to increase simulation speed [36].

## 2.3 Snowdrop project

This thesis is integrated into the Snowdrop project [16] of the Technical University of Delft. The main objective of this project is to design distributed algorithms that can be used in interactive environments in buildings. The solutions are based on embedding sensor nodes in the floors, walls and ceilings of buildings. The sensor nodes detect local pressure, temperature, and other environmental parameters. The measurement values are shared with other nodes in the vicinity (via wired or wireless connection). Each node is also equipped with actuators which are operated by algorithms using the values from different sensors and the values measured locally. Due to the large number of devices and distributed interactions, emergent phenomena occur. This leads to interesting possibilities for the system and the people in it. Nodes such as this were developed by Steffan Karger [18] and have been installed into the *Delft Proto Space 3.0* test-environment (see Figure 2.2).

Figure 2.2: ProtoSpace 3.0, Faculty of Architecture, TU Delft

The algorithms running on these nodes are written in *Proto* [5] which implements a novel paradigm called *Spatial computing* [43]. To help designers and other non-IT professionals who would be interested in designing interactive environments and do not have sufficient programming knowledge multiple solutions have been proposed.
Agostino Di Figlia has developed a framework called *Interactive Design Studio* which aims at abstracting from the technical details and letting non-IT professionals develop their *spatial computing*-based algorithms [19].
However it is very difficult to find an algorithm that displays certain emergent behavior [26]. Therefore Sjors van Berkel developed the *MetaCompiler* [38] which offers a top-down approach – finding the algorithm for the prescribed global behavior – instead of the bottom-up approach of Di Figlia. The method to achieve this is *Genetic Programming*.
However a major problem of the GP approach is that there is a massive amount of programs that need to be evaluated. Since these evaluations need simulations for multiple agents and iterations, the execution of the *MetaCompiler* needs to be accelerated.
This thesis is integrated to the *Snowdrop* project to provide GPU accelerated simulators that can serve as a back-end to the *MetaCompiler*, thus reducing search time by reducing the execution time of the simulations [39].

## 2.4 Conclusions of the literature survey

Our literature survey shows that the field of Agent-based simulation is very active in ongoing research motivated by the need to understand complex systems.

There are powerful, yet simple domain-specific languages which can be used by non-IT professionals to create models. We found however that these simulators tend to be slow for large-scale simulations.

In order to simulate large-scale systems researchers have to turn to parallel programming, which necessitates advanced programming skills, while development is slow and bug-prone. Furthermore, these solutions need to run either on clusters of computers, which are expensive, or on parallel accelerators (such as GPUs), which are cheap and efficient, but even more difficult to program.

We therefore identify a merger between these two solutions by a simulator that takes models written in a subset of the *NetLogo* language to provide simplicity and execute it on a CUDA capable GPU to provide execution speed-up. This way we are expecting to gain performance over the pure *NetLogo* while retaining its simplicity.

Finally, we found the Snowdrop project very useful to validate our simulator in a real use case.

# Chapter 3

# System design

In previous chapters we have seen the motivation to choose *NetLogo* and to accelerate it with CUDA. This chapter aims to provide an outline of the system of the accelerated simulator.

In Section 3.1 we investigate the main features of a *NetLogo* model and in Section 3.2 we look at the relevant features of CUDA programming. Section 3.3 shows how a *NetLogo* model could be mapped to the CUDA programming architecture and discuss the design of the system. Finally, in Section 3.4 we take a look at three algorithms that can be used to characterize *CudaSimulator*.

## 3.1 Overview of NetLogo models

NetLogo models consist of different types of agents, the geometric space that they inhabit, and their behavior. This means that they carry out operations (e.g. changing colors, moving, computing), affect their environment (writing global data) and influence each other (e.g. asking other agents to carry out pieces of programs, exchanging data, etc.). This behavior is defined by a program written in the *NetLogo* language.

The common structure of a *NetLogo* model is shown in Figure 3.1. There are two main procedures: one that initializes the model (*setup*) and one that runs a *tick* of the simulation (*go*) – this is usually run iteratively in a loop. Both of these procedures are executed by all agents in the simulation.
A *tick* is an iteration of the model execution, all agents run their program *once* within the tick. After that a counter is incremented. When the maximum number of iterations is reached, the simulation stops.

Agents are told to execute some program with *ask* blocks. This means that the target agent has to execute whatever is within the *ask block* (it can

be instructions and procedure calls as well). Agents can ask each other as well, which is the way they can exchange data.

### 3.1.1 Data representation

The data of *NetLogo* models are stored in different types of variables as shown in the following:

**Global variables** (*globalVar1*, *globalVar2*, *maxSimulationTime*) which are visible to all agents and exist only in one instance. These variables are either defined in the program code, or placed on the GUI (as slider or other input widgets). Apart from these, there are also default global variables, which influence the *NetLogo* environment, e.g. *ticks*, which measures the number of elapsed iterations and can be incremented by the *tick* command.

**Agent variables** (*agentVar1*, *agentVar2*) which other agents can access through the owner. These exist in one instance per every agent. These are mostly user-defined for a breed of agents explicitly. However, every agent has default shared variables, e.g. *color*, *heading*, etc.

**Locally declared variables** (variable $x$) which are defined for the current namespace and cannot be accessed from outside.

A variable can hold different types of data, e.g. numbers, character strings or an other agent. However, only numbers are supported in this thesis, the implementation of other data types are left for future work.

In *NetLogo* it is possible to use multiple kinds of agents in the same simulation. The difference among them is in the shared variables that they have and the behavior that they exhibit. Agents of the same kind are called a *breed*. For the sake of simplicity we will use in this thesis one breed per model only, since the algorithms that were used to test *CudaSimulator* (explained in Section 3.4) can be implemented with one breed. Implementing more breeds per simulation will require additional research. This extension is left for future work.

### 3.1.2 Concurrency model

*NetLogo* is a single thread application and concurrency between the agents is simulated via scheduling. *NetLogo* agents can execute programs two ways: if the code is in an *ask*-block the agents take turns and run exclusively, therefore there are no data hazards and no synchronization is necessary. Code that is inside of an *ask-concurrent* block [3], runs pseudo-concurrently. This means that the agents still take turns executing the code, but when they hit a so-called *switching command*, which can potentially change the state of a

```
globals [globalVar1 globalVar2]
breed [nodes node]
nodes-own [agentVar1 agentVar2]

to setup
  clear-all

end

to go
  ask nodes [
    if ticks >= maxSimulationTime [ stop ]
    let x 2
    doSomething
  ]
  tick
end
```

Figure 3.1: The usual structure of a NetLogo model

different agent, or the global state, execution is randomly transferred to another agent. An example for a *switching command* is move, which changes the coordinates of an agent, and thereby the global state.
Pseudo-concurrent execution emulates concurrency, because the *switching commands*, which can influence other agents, happen in a random order.

## 3.2 Overview of CUDA

For the purpose of explaining the modern GPU hardware, as well as understanding the CUDA programming model, we use the NVidia GTX480 GPU, the commonly available at the time the development and testing were performed. However, the same principles apply for both older platforms (e.g. GTX 295) and the newest ones (e.g. K20 and GTX680).

### 3.2.1 Hardware architecture

A GPU consists of multiple vector processors, called Streaming Multiprocessors – SMs, with 32 cores each. GTX480 consists of 16 SMs which are tightly connected to a local memory hierarchy. This comprises of global memory (accessible to all cores), local/shared memory (accessible to each SM) and caches. As a GPU is used as an accelerator, it can only work in the presence of a host(a CPU). The communication between the host and accelerator(s) is using the PCI Express bus of the system.

Since the placement of data has strong implications on the performance of the CUDA program, the programmer has to be aware of the memory ar-

chitecture, and has to define explicitly where a variable is placed. Therefore we list the possible types of memory below:

**On-board memory** is the most abundant storage of the GPU. It is accessible from the host computer thus it is used for communication between the host machine and the GPU (initialization values are copied into and results are fetched from global memory). This is also globally accessible to all the threads, however this is dangerous since it can cause data hazards. Global memory needs to be allocated by the host before the kernel is launched.

**Shared memory** is an on-chip memory which provides a method to communicate for threads in a block. It is used to pass information.

**Registers** are the scarcest and fastest way to store and access information. These are used to store temporary data and exclusively visible to individual threads.

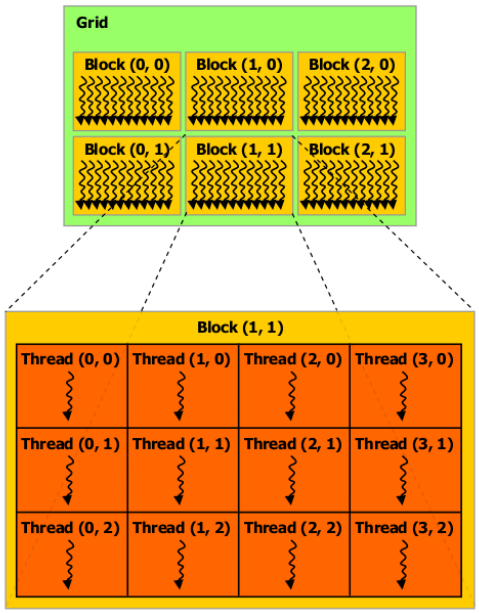


Figure 3.2: CUDA programming model

### 3.2.2 Programming architecture

CUDA is the software architecture that enables the GPU to execute programs written in C (and several other languages). The structural elements of CUDA programs reflect the hardware architecture.

We introduce these structural elements below (see Figure 3.2 for a depiction of the CUDA architecture):

**Kernels** are a portion of code that is to be executed on the GPU. The programmer can explicitly specify when a kernel is launched and it can be synchronized with the host machine code. A kernel is executed by threads on the GPU which can be arranged in a *grid* which specifies the mapping of the threads on SMs.

**The grid** consists of *blocks* which are units of scheduling and contain a set of threads. *Blocks* are scheduled independently from each other on a single SM. The programmer cannot make any assumptions about their order of execution, nor can he assume that a *block* finishes before the next one begins.

**Threads** are the basic parallel building blocks of CUDA execution. They can be scheduled with *zero overhead*. Thus a large number of threads is typically used as opposed to CPU programming. Every 32 threads are arranged in *warps*, which are executing in lockstep (their instructions are arranged in vectors in the SM). This brings restrictions for the efficient use of these GPUs. For example, in the case of branching the instructions in different paths are interleaved. This guarantees that the execution is still functionally correct, but the performance will be impaired.

As we can see memory can be read and written concurrently. In order to avoid hazards there are special instructions to implement thread barriers i.e. all threads that have hit the barrier wait until every thread in the block has reached the barrier. However there is no *global synchronization* available, i.e. the programming model does not include a way to block all the threads in the same barrier. Instead, atomic instructions are provided to access the memory.

The above mentioned architecture and limitations to the memory on the GPU is reflected in the placement of the data of the agents and the simulation. The global memory stores data that is rarely accessed (in order to save shared and local memory). It is also used to transfer data between the host and the GPU.
By contrast, the local and shared memory is used to store data that is accessed frequently. The local memory is used to store data exclusively owned by agents, whereas shared memory is used to store data that is shared among agents.

The memory allocation will be explained in detail in Section 3.3 in this chapter and in Chapter 5.

## 3.3 Design of the simulator extension

As we mentioned in the introduction, one of the purposes of *CudaSimulator* is to provide a *back-end* to the *Snowdrop MetaCompiler* which generates a multitude of models. In order to run these models on the GPU, they are first translated into a special bytecode and then executed by a virtual machine running on the GPU.
The system is made of two components: one of them extends the *NetLogo* engine and is in charge of translating the model into bytecode. The other part is the virtual machine that resides on the GPU and executes the bytecode program.

The reason for the separation of the two parts is to reuse the *NetLogo* engine as much as possible. *NetLogo* translates the models into an *Abstract Syntax Tree* representation and then executes them. Since the translation procedure is sequential in nature, it is more suitable for the CPU. On the other hand, it is relatively simple to translate the AST representation of the models into bytecode that can be downloaded and executed on the GPU.

As an alternative we considered using the *NetLogo Extensions* framework, which provides an API to the engine which can be used to carry out certain operations outside of the main engine. Therefore it would be possible to define operations that are to be carried out by CUDA through extensions. However the main obstacle is that the engine stores agent data internally, therefore large amounts of data would have to be extracted from the software engine, and moved between the GPU device and the PC. Since both of these operations are time consuming, they could outweigh the performance gains that were produced by the GPU. Therefore we made the decision to add our contribution to the *NetLogo* engine directly.

The mapping of the models generated by the *MetaCompiler* is shown in Figure 3.3 (cf. Figure 3.2). Since different models are independent of each other and agents in a model are only loosely dependent, there are two levels of parallelism that have to be considered: parallelism between the models and parallelism between agents. The easiest option is to map models on the blocks and agents to individual threads in that block. This also means that the execution order of the models is up to the internal scheduler of the GPU.

The data that is shared between the agents encompasses agent variables (such as color, heading, variables defined by the programmer), and flags for communication. In order to facilitate fast exchange of data, these variables and flags are placed in the shared memory. The exact layout of the communication flags is shown in Chapter 5.
As we have seen in Section 3.1 agents exist in a geometric space which

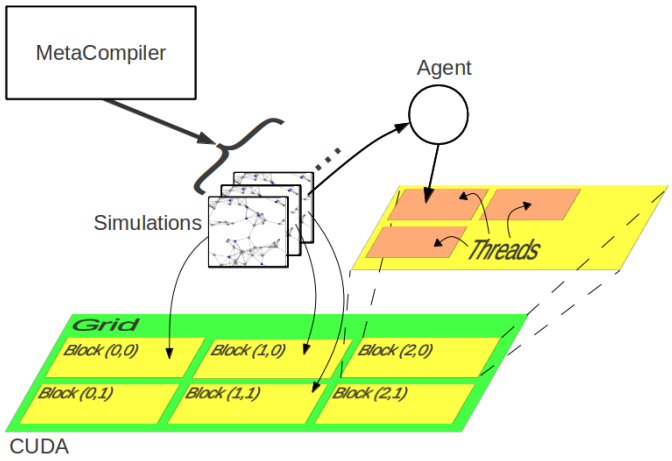


Figure 3.3: Mapping NetLogo models to the CUDA architecture

has different possible geometries. For the sake of simplicity we support only the fully toroidal shape, as it is enough to analyze the test algorithms (see Section 3.4). Implementing other geometries would necessitate some coding, which it is left for future work. The coordinates of the agent's position reside in the global memory in order to use shared memory sparingly, since the per block shared memory is usually scarce. We note however, that global memory has a higher latency, therefore frequent accesses can make the simulation slower.
The global variables and the fitness value are also stored in the global memory. All the data that has to be passed to the agents before the simulation starts are also stored in the global memory, since it is accessible by the host machine.

From the concurrency models of NetLogo we focus on concurrent execution, since exclusivity would serialize the execution of agents and it is undesirable on a parallel architecture.

The code run by the agents is downloaded to the GPU as a two-dimensional array (Figure 3.4). Each row is an individual instruction. The first value (with thick border) is the ID of the instruction after which the parameters are listed. Each parameter has two numbers associated with it: the first one encodes the type and the second one is the value. The encoding is shown in Table 3.1.

This continuous two dimensional array stores the entire program the agents have to execute. This means that all procedures from the model are appended after one another and they are identified by the row number in the bytecode array. The *ask* and *ask-concurrent* blocks are regarded as

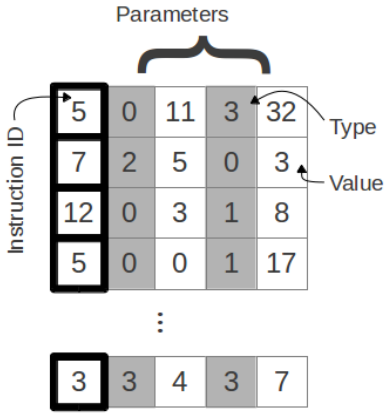


Figure 3.4: Structure of the bytecode

procedures as well. In Section 3.1 we mentioned the two main procedures: *setup* – which is always the first procedure on the list and contains an automatically generated jump command to *go* – and *go* – which contains a jump command at the end which points back to the beginning of the procedure (thus simulating the *forever button* functionality).

The simulation is terminated when all the agents have hit *stop*. This is made possible with barrier synchronization after which all threads but the main thread exit. The main thread is a designated thread (usually with $thread\_ID = 0$) that carries out global operations e.g. aggregating information. After the simulation stopped this thread carries out the bytecode of the fitness function and places the result in the last global variable. After this the CUDA kernel stops as well, and the host machine retrieves the fitness value and submits it to the *MetaCompiler* (also running on the host) which uses it to create the next generation of models. We note that *CudaSimulator* is designed to work with the *MetaCompiler*, but the actual coupling is left for future work. Figure 3.5 gives an overview of the complete system.

We note that the data transfer between the GPU device and the host machine is relatively slow, therefore it should be minimized whenever possible. This is the reason why the fitness value is calculated on the device. If it would be calculated on the PC, the values of all the necessary agent variables should be transferred back to the PC, which can be lead to a significant performance loss. The disadvantage of this is however, that the expression used to calculate the fitness value is limited by the capabilities of the *CudaSimulator* virtual machine (e.g. no lists or other complex data structures can be used, etc.).

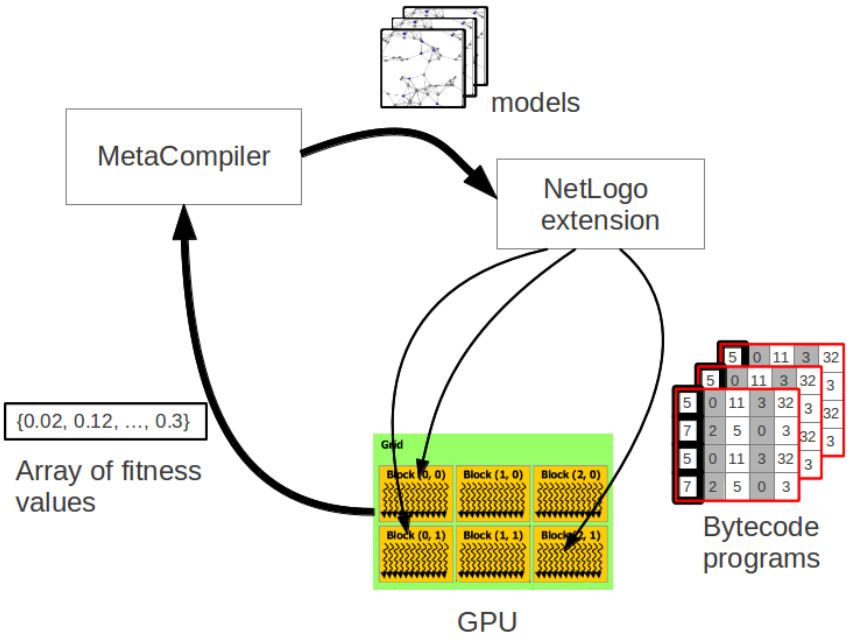


Figure 3.5: Diagram of the system

## 3.4 Test algorithms

We mentioned at the end of Chapter 2 that the *CudaSimulator* is integrated into the *Snowdrop* project. This integration provides us with a specific use case, i.e. the simulation of large-scale MANETs (Mobile Ad-hoc Networks).

Obviously, there are many algorithms used in MANETs. To illustrate the functionality and performance of the *CudaSimulator*, we chose three of them: *Leader election* [15], *Random walk* [7] and *Firefly synchronization* [25]. These algorithms are representative, since they employ different network models.
In the case of *Random walk* the network is completely disconnected, whereas

| Type | Encoding number | Description |
|---|---|---|
| Local | 0 | Variable that is stored in the local memory |
| Shared | 1 | Variable that is stored in the shared memory |
| Global | 2 | Variable that is stored in the global memory |
| Constant | 3 | Constant value |
| Agent identifier | 4 | Identifies an agent |

Table 3.1: The possible data types of parameters

in the case of *Leader election* it is connected but static. Finally *Firefly synchronization* works on a dynamically changing network (the sensor nodes are connected only if they are within the transmission range of their radio unit). The three test algorithms are presented in detail in Appendix A.

In the following chapters we will elaborate on how the above mentioned functionalities were developed and how well the test algorithms are performing in the resulting framework.

# Chapter 4

# NetLogo to bytecode translator

This chapter gives a detailed description of the process that translates a *NetLogo* model into its bytecode representation for the CUDA VM.

Section 4.1 gives a detailed insight into the relevant parts of the *NetLogo* execution engine. In Section 4.2 we will see how the source code of the translating software is structured. Section 4.3 presents the naming conventions which users of this software have to adhere to in their models. Finally we will look into the details of the translation mechanism in Section 4.4.

## 4.1 Overview

Two of the most important components of *NetLogo* are the compiler and the NVM (NetLogo Virtual Machine).

The compiler processes and converts the code of the model into an AST (Abstract Syntax Tree) representation. The nodes of the AST are called primitives. These are very simple operations which serve as the instructions of the NVM. Under the hood, these primitives are *Java* classes which are compiled into *Java bytecode* and executed by the NVM one-by-one [36].

The NVM executes the *NetLogo* program for the agents. The program is divided into a set of *jobs*, which are smaller portions of the code (usually the contents of an *ask*-block, since *ask*-blocks are used to initiate some action in *NetLogo*). There is a set of agents that are assigned to each *job*. The NVM includes a scheduler, which puts the *jobs* in some order and feeds them to the NVM to be executed. During the execution of a *job*, the engine iterates through the agents in the agent set, carries out the instructions, and changes the local data of the respective agent. With this structure the NVM can

execute an arbitrary number of agents in a single thread.

As we mentioned in Chapter 3 there are two types of execution: exclusive and concurrent. An exclusive job is comparable to *critical sections*, agents are selected in a random sequence and they carry out every instruction of the job before a different agent is executed. In the case of a concurrent job the agents are executed until a *switching statement* occurs. A switching statement is a statement which changes global state or the state of a different agent. Changing the executing agent at a switching statement emulates concurrent execution, since the order in which data is changed is random at any point.

*NetLogo* is primarily used with a GUI. However, it is possible to start it in *headless* mode which means that there is no GUI and the application is controlled programmatically. This usually means that certain functions (e.g. open model, call *setup* and *go*) are called from an external launcher program written in Java. It is used when models are run in a batch, and they need to be started automatically. This is the way the *MetaCompiler* connects to *NetLogo* and therefore we use the *headless* mode in this thesis.

The engine of *NetLogo* is written in two languages: Java and *Scala* [29]. Since both of these languages run on the JVM (Java Virtual Machine) [22] one could use JNI (Java Native Interface) [21] to connect the *NetLogo* engine with the CUDA kernels (which is implemented in native code). Since programming the JNI tends to be complicated and bug-prone, we chose the simpler and more convenient JCuda library [42] instead. JCuda hides the JNI glue-code and provides an interface to most of CUDA's functionality seamlessly with Java.

## 4.2 Structure of the source code

One of the main requirements during the development was for the software modules added to *NetLogo* to facilitate GPU execution to interfere with the rest of the application as little as possible. This means that the modifications in the original code are kept to the bare minimum. This should enable us to commit this software into the codebase of *NetLogo*.

The GPU extension resides in its own package, separated from the other code. We only introduced two modifications in *NetLogo*'s application code:

**In the compiler:** the compiler translates the program code of the model first into an AST representation, further translated into a list of primitives, which are then compiled into *Java bytecode*. However due to

encapsulation, any intermediate data is hidden. Therefore we added a method to be able to extract the generated AST tree.

**In the execution engine:** Since the GPU execution replaces the normal execution mechanism of *NetLogo*, the NVM is no longer necessary for the execution of models. The normal execution path is not disabled however, since we use the engine to extract information from the model (e.g. initialization of global variables). Therefore, the *setup* procedure is executed by the NVM, after which the GPU execution commences. Execution for any other procedure (e.g. *go*) by the NVM is disabled in the *CudaSimulator*.

## 4.3 Naming conventions

Before the *NetLogo* can be executed on the GPU the CUDA kernel has to be set up. To set up a kernel certain parameters are necessary (e.g. number of threads, size of shared memory, etc.). These parameters are static and need to be known before launching the kernel.

Contrary to this, parameters can be changed dynamically (in *NetLogo* agents can be created and destroyed at any point). The change of parameters is usually done by executing certain commands in the program code.

This means that the kernel parameters have to be inferred by analyzing the program code. However, to simplify the analysis certain restrictions and naming conventions are necessary. We list here the elements that are inferred from this analysis and the associated analysis restrictions.

**Number of agents:** Agent creation can only take place in one procedure, which has to be called *gpu_initCreateNodes*. The *NetLogo* command *create-turtles* – which is responsible for agent creation – consists of the number of agents and some initialization code. The translation process extracts this initialization code and inserts it into the AST of the *setup* procedure directly.

**Transmission range:** It is used in simulations where agents communicate with other agents, who are within a given range. The transmission range is set by the programmer in a global variable called *gpu_transRange*.

**Aggregation functions:** These are functions in which an agent gathers information about the others, and are therefore used to collect data about the status of the simulation (e.g. the average of some agent variable at all agents, etc.). In order to avoid read-write hazards, only

the master thread (*thread 0*) is allowed to read the data of all the agents while the others wait. Procedures that are executed by one thread only are enclosed by *thread-guards* (see Section 4.4.3). These procedures have to have names that begin with *gpu_aggr*.

**Fitness function:** This is a special instance of an aggregation function. This procedure has to be executed by the master thread after all threads have reached the *stop* instruction (i.e., the end of the simulation). After the threads have reached the last synchronization barrier the master thread jumps to the address which was passed to the CUDA kernel separately. In the fitness function the master thread collects data from all the agents in the simulation and computes a fitness value from it. The fitness function has to be called *gpu_aggrGetFitness*.

## 4.4 Translation into bytecode

We have seen already how the AST is obtained. Next, we discuss how it gets be translated into bytecode.

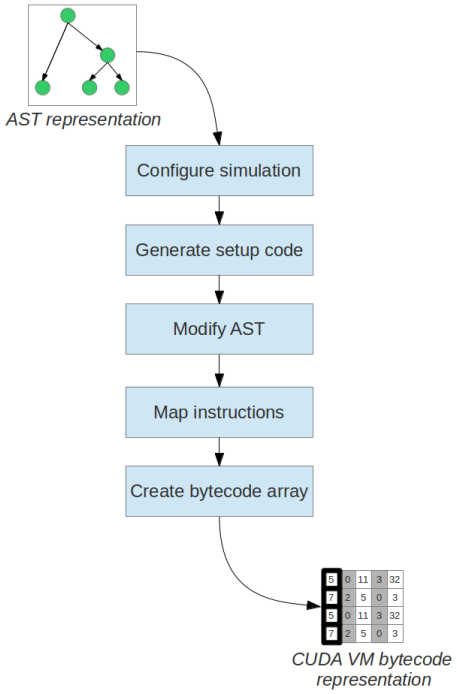


Figure 4.1: Overview of the translation process

Figure 4.1 shows the basic steps of this translation. The translator software processes the AST in multiple passes, i.e. it iterates through all nodes of the AST to process and extract information. In the next iteration it changes nodes when necessary. At the end of the AST processing steps we have a syntax tree that includes only nodes that are relevant to the GPU execution and can be directly translated into CUDA VM bytecode. We note that a new AST is generated as a result of each step before *"Map instructions"*.

The part of the translation process that is responsible for bytecode generation starts with the "*map instructions*" phase, which maps the AST nodes to bytecode instructions. The result is a collection of multiple small arrays of bytecode instructions. This is because *NetLogo* stores the AST of each procedure separately, so the mapper creates a bytecode array for each of them. At the end of the "*create bytecode array*" phase, all these separate instruction arrays are concatenated, and a continuous bytecode array is formed.

Since the row number of the instructions in the final array is unknown until the separate arrays are assembled, the branching and jumping instructions (e.g. *ifelse*, *for*, etc.) are also stored in a separate array. The targets of jumping and branching instructions in these operations are initially substituted by symbolic names. In the *branch assembly* substep, these names are replaced by row numbers.

The arguments of the resulting bytecode instructions are still in human readable format (e.g. *AGENT.HEADING*) in order to simplify debugging. These are renamed at the end in the *"create bytecode"* step, so that the result is a numeric array (see Chapter 3).

In the following sections we present the steps and substeps of the translation process in more detail.

### 4.4.1 Configure simulation

The very first step of the translation is to extract all information necessary to set up the simulation. This means that operations that include critical parameters (see Section 4.3) are extracted. There are two operations in this step: the extraction of the number of agents and the transmission range.

### 4.4.2 Generate setup code

In this step the AST corresponding to *NetLogo*'s *setup* procedure is generated (see Figure 4.2). Initialization code for the shared variables of the agents and the global variables of the simulation are generated in the sub-

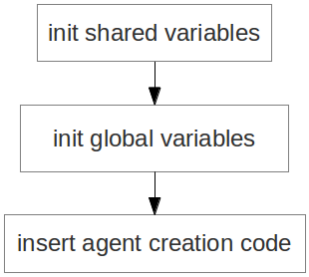


Figure 4.2: Generate setup code process

steps *"init shared variables"* and *"init global variables"*. This is done by extracting the list of variables from the procedure node of the AST generated by *NetLogo*. After extraction an assignment expression is generated for each variable, where variables receive the value zero. The instructions from the *create-turtles* instruction are extracted and inserted into the AST in substep *"insert agent creation code"*.

### 4.4.3 Modify AST

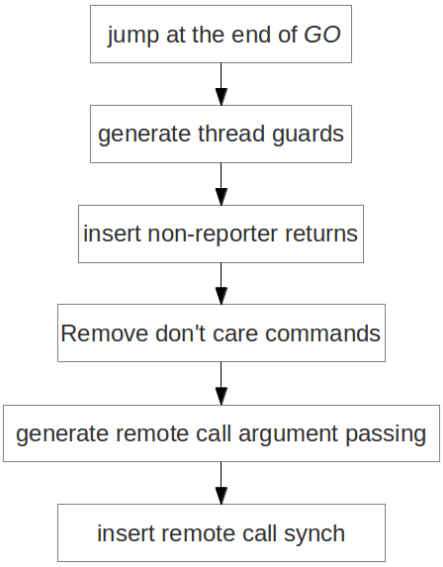


Figure 4.3: Modify AST process

This is the part where the ASTs of the procedures are modified such that they can be directly mapped into bytecode instructions.
First, a jump command is inserted at the end of the *go* procedure which points to the beginning of *go* in order to create an unconditional loop. This

is necessary since in the usual *NetLogo* models the *go* procedure is implemented to be run once, and the button that is assigned to it on the GUI runs it over and over again (the so called *forever button*).

Next, the translator looks for *aggregating functions* (see Section 4.3) in the substep *"generate thread guards"* and places so called *thread guard* instructions at the beginning and end of each one of the function bodies. Executing these instructions causes the agents to set/reset a locking flag. Where this flag is set, only the master thread can carry out operations, the rest are blocked. In this way, read-write hazards can be avoided.

The return statements are inserted at the end of *non-reporter procedures* in the substep *"insert non-reporter returns"*. *Non-reporter procedures* are procedures that do not return values. The return commands at the end of those procedures, which do return values, are translated at a later step.

Commands that have no meaning in the context of the GPU simulation (e.g. *clear-turtles*, *reset-ticks* are removed in the step *"remove don't care commands"*).

We designed communication between agents to happen through a so-called *remote call* in the *CudaSimulator*. *Remote calls* reflect *NetLogo*'s way of facilitating communication between agents.

In *NetLogo*, an agent communicates by instructing a target agent to do something. This behavior is implemented by *ask*-blocks. Also, *ask*-blocks are generally the way to initiate action in *NetLogo*. Therefore, a conventional model consists of higher level *ask-blocks* in which the simulation tells agents to carry out certain actions (e.g. move in random walk). Agent communication is implemented by lower-level *ask*-blocks embedded in higher-level *ask*-blocks (simulation asks agents to ask another agent to do something). The translation software notices *nested ask-blocks* and translates them into *remote call* instructions.

In *NetLogo ask*-blocks values can be passed to the target agent if a variable that was declared in the outer *ask*-block is referenced in the inner *ask*-block. The translation software notices value passing in substep *"generate remote call argument passing"*. Explicit instructions are inserted in the AST to place and fetch parameters in shared memory such that it can be used to pass parameters in *remote calls*.

The last substep is *"insert remote call synch"*, in which synchronization barriers are generated at the beginning and at the end of remote calls. Synchronization barriers around remote calls are necessary to avoid hazards

when agents write each other's memory in the course of communication (see Chapter 5 for a detailed explanation).

### 4.4.4 Map instructions

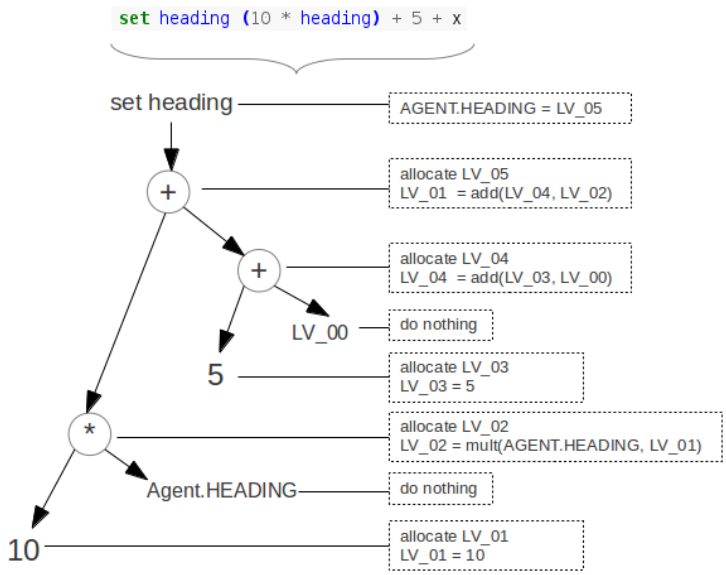


Figure 4.4: Example translation of an AST command node. Note local variables are called *LV_x* for the sake for brevity.

The modified AST is traversed again in this substep. At every node one or more bytecode instructions are generated. Since commands depend on their parameters (reporters, as they are called in *NetLogo*) the traversal is *bottom-up* i.e. beginning with the children.

*NetLogo*'s reporters are statements that report a value. The values can be a variable (global, agent, etc.), a constant value, a random value, or an arithmetic expression which has other reporters as its parameters. The translation process allocates a variable on the GPU for each reporter.

Figure 4.4 illustrates the process of the translation of a command node and its children (reporters). The diagram is to be read bottom up.

Let us assume that the variable $x$ has been allocated earlier to local variable *LV_00* when a command assigned a value to it. All agents have the variable *heading* (a direction in which agents can move in the geometric space they inhabit) by default therefore it has already been allocated in the

shared memory in an earlier step.

As we can see, a local variable is allocated for every reporter and the bytecode for the operation that assigns this variable is generated. In the case of random reporters (e.g., *rand-xcorr*, which reports a random coordinate), a *rand* command is generated to fill in the value.

Local variables are allocated in a similar fashion when they are directly declared in *NetLogo* (e.g., with the command *let*).

The names of the allocated variables are stored in an associative array in the translation program, such that later references can be resolved. The name of the allocated variables consists of the name of the containing procedure and an identification number (e.g. the $5^{\text{th}}$ variable that is allocated in the procedure *MYPROC* will get the name *MYPROC.LOCAL5*) or in the case of an agent variable (e.g. heading) the name will be *AGENT.HEADING*. The naming provides readability for debugging and will be renamed into an identification number at a later step.

We note that, there are reporters whose mapping is not straightforward. Such are the aggregating reporters which collect data from the agents, e.g., *mean*, *max*, *min*, *countwith*. Aggregating reporters collect the value of a given variable from all agents and either calculate the mean, maximum or minimum of them, or, in the case of *countwith*, count agents where an expression over the variable is true (e.g. count how many agents are out there with an $x$ variable s.t. $x > 0$). These reporters are translated into a *for loop* which iterates through all agents, reads the value of the variable of interest and finally evaluates the expression on this variable. This final result expression can be a simple arithmetic operation in case of *mean* or *max* – or an arbitrary chain of reporters in case of *countwith*.

Similarly, there are commands which need to be translated into complex structures as well.

*For loops* are translated into a structure that consists of three parts. The first part includes chained reporters (a predicate expression) which are evaluated to decide whether the loop should be broken or continued on the next iteration. The second part is the looping instruction itself, which evaluates the local variable that contains the result of the reporter chain. If it evaluates to *true*, jump is required to a specified adress, that contains the third part – the loop body. The loop body contains the instructions executed in the loop and it is translated as a procedure, (i.e. it is a separate list of bytecode instructions which will be assembled with the rest of the program in the last step see Section 4.4.5).

In the conventional *NetLogo* programming style, the execution by the agents is initiated by an *ask*-block in the *go* procedure (see Figure 3.1). This *ask*-block is disregarded, and the code inside of the block is directly placed into the bytecode program.

The *ask*-blocks inside of the code executed by the agents are however important as these are used to facilitate agent to agent communication. In this case the target is extracted (either *ask neighbors* or *ask agents in the transmission range*) and *remote call* instructions are created. The *remote call* instructions are embedded in a loop which iterates through all neighbors or all agents in the transmission range. The structure of the looping block is similar to that of for loops, albeit it is surrounded by synchronization instructions (see Chapter 5 for further explanation).

The remaining commands are directly mapped to the corresponding bytecode instructions.

### 4.4.5 Create bytecode array

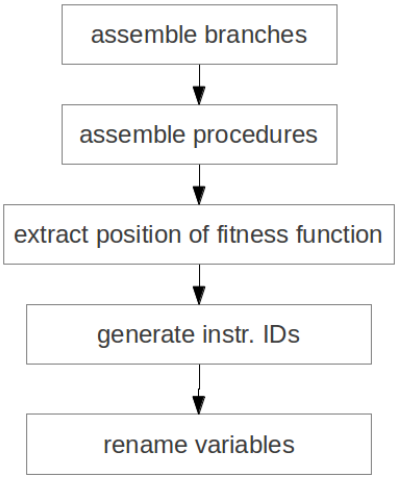


Figure 4.5: Create bytecode array process

The final step organizes the generated bytecode instructions into a format that is directly executable by the CUDA VM.

Since *NetLogo* stores the ASTs for the procedures separately, all the above mentioned operations run on the procedures one-by-one. Therefore the result is a list of bytecode fragments corresponding to each procedure. More-

over, the branches of the *ifelse* instructions are also in separate fragments. The reason for this is that until all procedures are assembled, the bytecode instructions do not have addresses, so the branches are only referenced symbolically.

Thus in *"assemble branches"* and *"assemble procedures"*, a continuous stream of instructions is generated and the symbolic references are changed to "real" addresses. At this point the *fitness function* has an address as well, so it is extracted and saved into a variable to be passed to the CUDA VM kernel when it is launched.

Also here, the names of the variables are changed into an ID number, in the order of appearence. Thus, local variables will be (*L00, L01, L02, ...*), shared variables (*S00, S01, S02, ...*). Of course L and S are represented as type numbers as well, according to Table 3.1. The final representation, that is to be sent to the *virtual machine*, is in the case of the first local variable *00*, the first shared variable *10*, and so on.

Up until this moment the instructions were referenced by names (again for human readability). In the *"generate instr. IDs"* substep, these names are replaced by ID numbers from a lookup table. At the end of this substep the program has been translated into a two-dimensional array of numbers that can be directly downloaded and executed by the CUDA VM.

# Chapter 5

# CUDA Virtual Machine

This chapter describes the part of *CudaSimulator* that runs on the GPU. This component of the framework is a virtual machine (VM) which is designed and built to execute the bytecode generated by our translator, as presented in see Chapter 4.

In Section 5.1 we explain how the *CudaSimulator VM* works and what elements it consists of. In Section 5.2 we explain the need for synchronization between agents and describe how it happens. In Section 5.3 we give an overview of the layout of the shared memory which facilitates communication between agents and in Section 5.4 we show the way this communication happens.

## 5.1 Overview of the CudaSimulator VM

The purpose of the *CudaSimulator VM* is to iterate through the *bytecode array* of *NetLogo* models and execute the instructions.

In essence, the virtual machine runs in a loop. In each iteration the current instruction ID is read from the bytecode array and a corresponding function is called. The function implements the given instruction. The addressing of the current instruction happens with an *instruction pointer* which can be modified from the instructions (to facilitate jumps and function calls). This way, the virtual machine is similar to a CPU which reads instructions from its program memory with the help of the program counter.

From the CUDA perspective, each agent is implemented as a CUDA thread. We remind the reader the fast local and shared memory hosts agent data, which is accessed frequently and used for agent-to-agent communication. The relatively slow global memory hosts simulation data (e.g. the tick

counter), which is read and written less often.

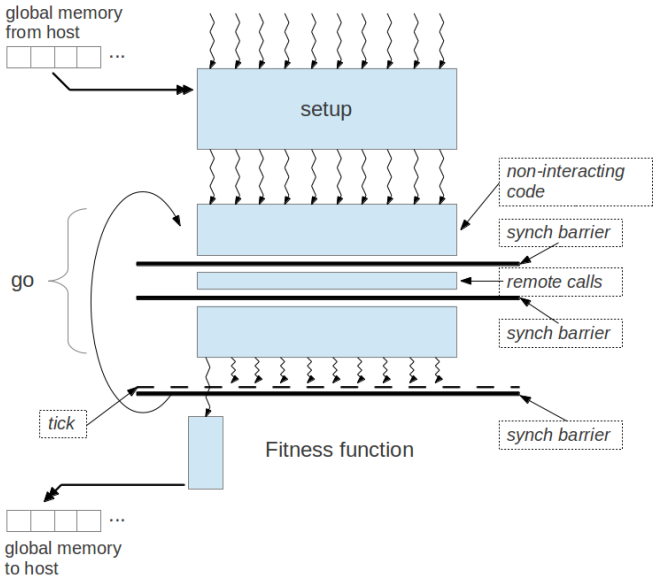


Figure 5.1: Overview of simulation on the CudaSimulator VM

Of course the VM has to do more that just executing the instructions: moving data between host and GPU, initializing agents and providing synchronization to agents. The run-time operation of the VM can be represented in a sequence of stages as shown in Figure 5.1.

**Kernel initialization (setup):** Data from the host is copied to the device: e.g. the initial values of the global memory, the address of the fitness function, and the *bytecode array*. The kernel is launched, the local data of the agents is set up.

**Simulation execution (go):** The main loop starts and executes the instructions one-by-one. The agents break out of the main loop when they encounter the *stop* instruction. After this, they synchronize (wait until all agents have reached the *stop* instruction).

**Fitness function execution:** Finally, the *master thread* jumps to the specified address in the *bytecode array* and executes the instructions of the fitness function. After this, the fitness value is copied in the global variable that is reserved for it. By convention, the last global variable is reserved for the fitness value. After the fitness value is calculated, all threads exit and the host composes the fitness values from different simulations and passes them to the *MetaCompiler*, which will eventually generate the next generation of programs.

### 5.1.1 Components of the VM

As we mentioned previously, the *CudaSimulator VM* can be seen as a model of a rudimentary processor. It executes a sequence of instructions in the order determined by the *"instruction pointer"*.

Furthermore, our VM also has different kinds of memory: local, shared and global. Local memory is used internally, shared memory is used for agents to communicate, and global memory is used to communicate with the host machine. It has an instruction memory which contains the instructions and their parameters immutably. Furthermore, there are memory places which store the spatial position of the agents.

Finally, we have included two stacks: one is the *callstack*, and the other one is the *parameterstack*. In the case of a function call (including *remote calls*) the address of the current instruction is pushed onto the *callstack* before the *instruction pointer* is modified. When the VM encounters the *ret* instruction, the top of the stack is popped, and the *instruction pointer* is restored to the popped value + 1. The *parameter stack* stores the parameters in a function call (*remote calls* not included). This way it is possible to send multiple parameters to the agents. The instructions to push and pop parameters are inserted by the translation software (see Chapter 4).

## 5.2 Synchronization

In the CUDA concurrency model, a *barrier synchronization* is the simplest synchronization primitive offered (*__synchthreads()*). A synchronization barrier means that when a thread reaches this synchronization barrier, it has to wait until all the other threads have reached it in the block. Care must be taken with branching, since if the barrier is placed in an execution path which is not taken by all threads, the system goes into a deadlock state.

A further option provided by CUDA to avoid data hazards is atomic operations. The disadvantage of these is that these are generally slower and only a few operations have atomized versions (e.g. arithmetics, swapping, and comparisons). We use atomic synchronization when accessing the shared flags of the agents.

The *CudaSimulator VM* contains multiple levels of synchronization. The primary level is instruction-level synchronization which is necessary to ensure memory consistency. In this scheme, all instructions are split into two parts (see Figure 5.2): a part in which only reading from variables and calculations happen, and a part in which calculation results are written back. This way data hazards are avoided.

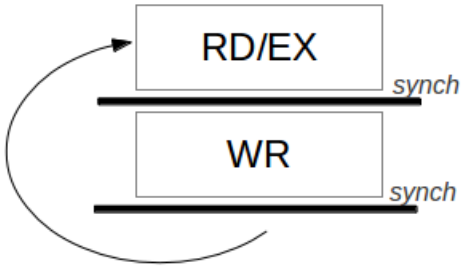


Figure 5.2: Instruction level synchronization

The secondary levels are of *tick synchronization*, *thread stop synchronization*, and *remote call synchronization*. These are all barriers as well, but they are implemented using flags in the shared memory instead of the primitive available in CUDA. This is necessary, because when an agent has not yet reached the secondary barrier it still has to meet primary ones (of the individual instructions). The two levels are necessary since barriers cannot be nested in CUDA.

To understand this, let us suppose that all but one thread have finished execution (reached *stop*). Those threads who are finished will set the flag corresponding to *thread stop synchronization* in the shared memory and then check out the flags of the other agents. After they have found out that one thread has not set its flag yet, they will carry on looping and reaching the *instruction level synchronization* barriers and check the flags at the end of each iteration. When the last thread finally finishes it sets its flag which will let all threads break out of the loop at the same time.

*Tick synchronization* implements the same scheme when the *tick* instruction is reached (at the end of an iteration of *go*). *Remote call synchronization* synchronizes threads before and after remote call blocks which facilitate communication between agents (see Section 5.4).

## 5.3 Shared memory flags

In the *CudaSimulator VM*, the communication between threads in the same block is facilitated by the shared memory. Figure 5.3 shows the layout of the shared memory, which contains the aforementioned synchronization flags (*thread finished*, *tick finished*, *ask finished*) and *remote call slots*.

Remote calls correspond to the *NetLogo*'s *ask*-blocks in which an agent asks a different agent to carry out a procedure. To facilitate this behavior

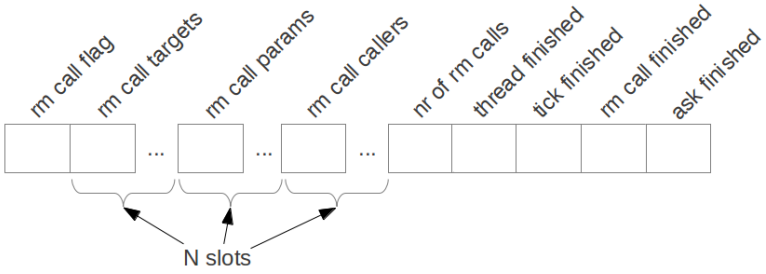


Figure 5.3: Layout of the shared memory flags

in the *CudaSimulator*, the calling agent allocates a *remote call slot* in the shared memory of the target agent. This means saving *nr of rm calls* and incrementing it atomically. The saved value determines the slot of the caller in the callee's shared flags. Slots consist of *rm call targets*, *rm call params* and *rm call callers* (shown on Figure 5.3). These flags exist in multiple instances, one for each slot. After this the caller places the address of the target procedure (in *rm call targets*), the ID of the calling agent (in *rm call callers*) and one parameter (in *rm call params*).

Slots are necessary in order for each agent to enqueue the incoming remote calls. The number of slots should reflect the worst-case number of incoming calls in a tick. This corresponds to the number of agents for a *NetLogo* model where an agent is connected to all other agents (e.g. *Firefly synchronization*). However, if an agent needs only one connection (e.g. *Leader election*) the number of necessary slots is one. If the agents in the model are fully detached (e.g. *Random walk*), no slots are necessary as no communication takes place. Currently, the *CudaSimulator* cannot infer the number of necessary slots from the code as this is a parameter of the model. We believe however that this can be done by the user or by overprovisioning. Given that the memory resources on the GPU are scarce, we prefer to avoid overprovisioning and rely on the user to provide this value. We believe this should not be too difficult, as the intended communication pattern is already familiar to the model writer.

The flag *rm call* is set by the caller to signal a new remote call. When the callee is finished with the execution, it sets the caller's *rm call finished* flag.

The flags *thread finished*, *tick finished* and *ask finished* implement the higher level barrier synchronization: synchronization at the end of the program (exiting thread), at the end of ticks, and before and after performing remote calls, respectively.

## 5.4 Agent communication

As we have seen in Section 5.3 agents communicate by sending *remote calls* to each other.
Figure 5.4 shows how *remote calls* are placed. When the caller executes the instruction for *remote calling*, it needs to check whether it has allocated a slot already. The reason is that placing parameters is implemented in a separate instruction and if it precedes the call, the slot allocation has happened already. Slot allocation happens by atomically incrementing the counter in the target's *nr of rm calls* flag. After this the caller writes the corresponding values in the shared memory of the callee. The caller waits (loops in the same instruction) until the call is finished (i.e. the *rm call finished* flag is reset by the callee) then proceeds and then continues executing the instructions of its program.

Figure 5.5 shows how instructions are executed and what happens when an incoming *remote call* is received. The agent executes instructions in the main loop until the *rm call* flag is set. The incoming *remote call* forces the agent to jump to the procedure address what was passed to its shared memory (*rm call target*). The callee saves data from the current slot and atomically decrements *nr of remote calls*, and begins to carry out the instructions of the remote call. At the end of each remote call the callee sets the callers *rm call finished* flag, thereby signaling to the caller that the call is finished. When there are no more incoming remote calls the agent resets its *rm call flag* and resumes the execution of the instructions of the normal program.

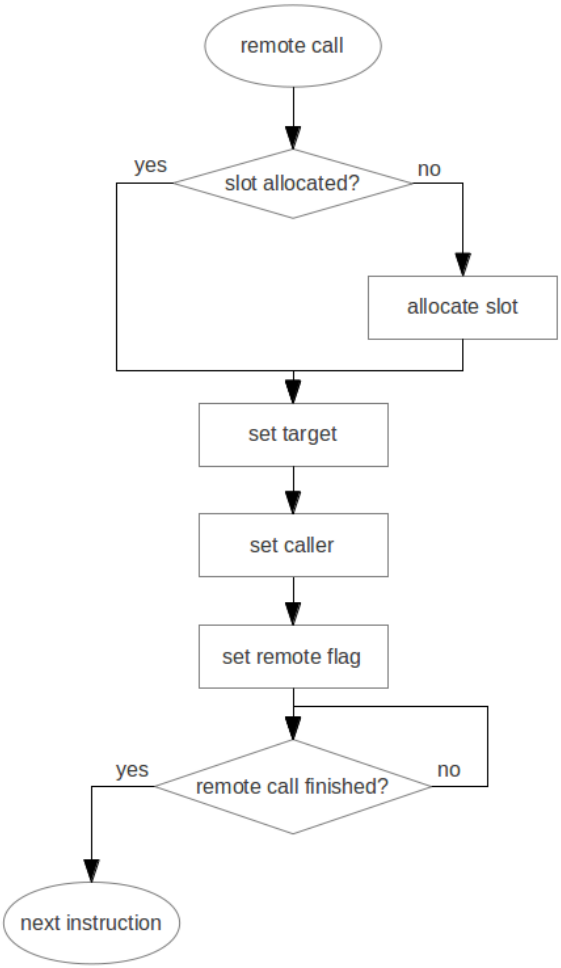


Figure 5.4: Diagram of remote calling

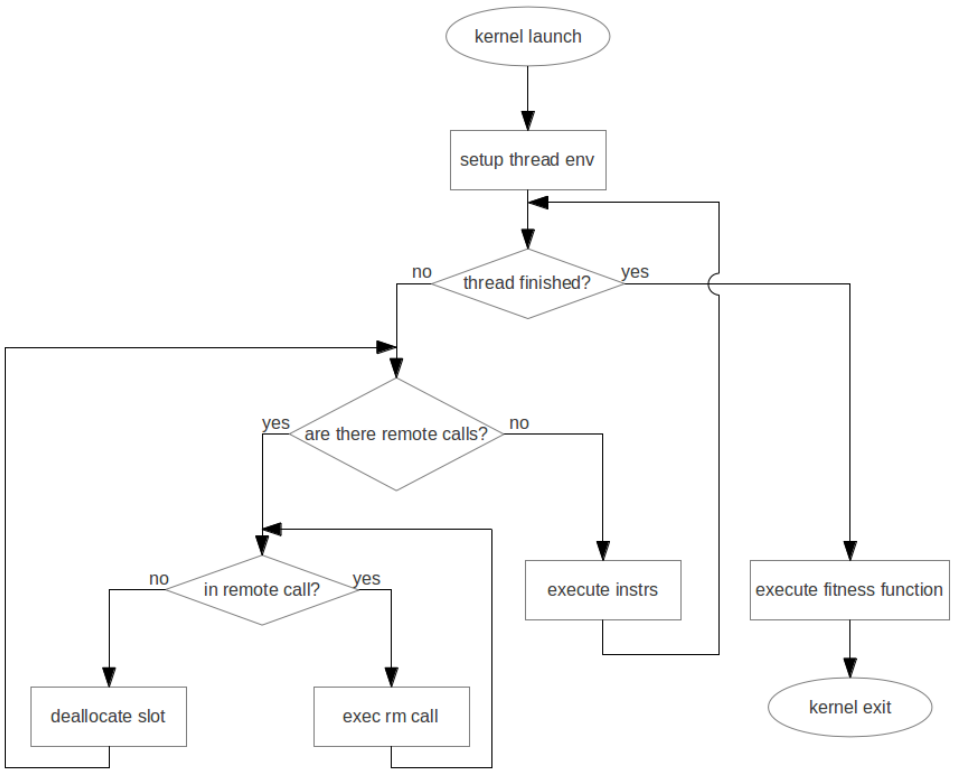


Figure 5.5: Diagram of instruction execution

# Chapter 6

# Experiments and results

This chapter provides a set of experiments and measurement results in order to quantify the capabilities and shortcomings of our *CudaSimulator*.

As the first step, we explain the setup we used to carry out the measurement, in Section 6.1. Then, we examine the limitations of the models in Section 6.2 – i.e. how many simulations can be launched at the same time as well as how many agents can those simulations have. In Section 6.3 we examine the execution time taken by the simulations of various lengths and simulation sizes. The measurements in Section 6.4 show the time needed by the *CudaSimulator* translation component which generates the bytecode representation of the models. In Section 6.5 we compare the performance of *CudaSimulator* to alternative simulation methods. Finally in Section 6.6, we take a look at models which are functionally correct, but the way they are programmed causes the simulator to run inefficiently.

## 6.1 Experimental setup

The measurements and experiments were carried out in the computation cluster of VU Amsterdam on a pool machines equipped with NVidia GTX 480 GPUs. The pool consists of computers with the same parameters. We note that programs are submitted as jobs in a queue wich selects a random computer from the pool. This means that programs might have been run on different computers of the same kind.

It is also important to note, that the two parts of *CudaSimulator*, the translator and the virtual machine were executed separately, such that the bytecode array generated by the translator was saved into a file, which was opened by a C program launching reading the bytecode and launching the virtual machine. This was necessary in order to be able to measure their execution time separately. However, the sum of the execution times is expected

to correspond to the execution time of the whole *CudaSimulator*, since the exact same operations happen when the virtual machine is launched automatically at the end of translation.

Although it would be possible to run different models at the same time, we launched multiple instances of the same model, because this way we can analyze the relation between a specific test algorithm and the measurement results.

## 6.2 Limitations

The number and size of the models that can be executed simulataneously by *CudaSimulator* are limited by the available resources on the GPU. The resource usage of a simulation depends on the algorithm that is simulated and the placement of the data in the memory. In order to investigate the optimal resource usage, three configurations were compared, in which the data was placed in different ways.

### 6.2.1 Register intensive configuration

The first configuration is called *register intensive*, because a lot of data is stored in the GPU registers. This is the configuration that was described in Chapters 3 and 5 – the local variables, the callstack, etc. reside in the registers allocated to each agent. The shared memory contains the shared variables and the shared flags (as shown in Chapter 5).

Table 6.1 shows the maximum number of threads and blocks (the number of agents and the number of simultaneously executed simulations). It is clearly visible that the *Leader election* is algorithm has the smallest memory usage. The last column shows the size of the allocated shared memory. In this configuration the *Firefly synchornization* algorithm was not able to start.

| **Algorithm** | **Max. agents** | **Max. blocks** | **Shared memory** |
|---|---|---|---|
| Leader election | 32 | 1710 | 24 kB |
| Random walk | 38 | 1190 | 24 kB |
| Firefly synch | N/A | N/A | N/A kB |

Table 6.1: Limits with register intensive configuration

### 6.2.2 Shared memory intensive configuration

Registers are scarce compared to shared memory, since they are used to store data that was generated by the compiler, e.g. loop counters, stack-frames, etc., – and therefore less of it is available altogether. One way to save registers is to put some of the local variables in the shared memory. Note that if this not done by the programmer by hand, any register spills that the compiler needs to perform are using the global memory for temporary storage. The performance penalty can be, in that case, 2-3 orders of magnitude larger than when manually using the shared memory.

The configuration in which more data is placed in the shared memory is called *shared memory intensive*, and Table 6.2 shows the limits in this configuration. Note that all algorithms could start, but the number of agents remain relatively small.

| **Algorithm** | **Max. agents** | **Max. blocks** | **Shared memory** |
|---|---|---|---|
| Leader election | 32 | 2833 | 46 kB |
| Random walk | 38 | 2330 | 47 kB |
| Firefly synch | 24 | 750 | 48 kB |

Table 6.2: Limits with shared memory intensive configuration

### 6.2.3 Shared memory intensive with instructions in global memory

In the shared memory intensive configuration the instruction list and the parameters are stored in the shared memory for each block. In order to save space, instructions and parameters can be placed in the global memory. The maximum possible agents and simulations are shown in Table 6.3. We note that there is no significant improvement over the register-intensive or the shared memory intensive configurations. In fact, the number of agents per block is not changing due to an unknown reason. However, the shared memory solutions do allow more concurrent simulations.

### 6.2.4 Global memory intensive

Finally, it is also possible not to use the shared memory and registers at all. All the data is stored in the global memory instead. While this takes toll on the speed of the program, there is more space. The number of possible agents is the same as in the case of previous configurations for *Leader*

| Algorithm | Max. agents | Max. blocks | Shared memory |
|---|---|---|---|
| Leader election | 32 | 2830 | 42 kB |
| Random walk | 38 | 2380 | 46 kB |
| Firefly synch | N/A | N/A | N/A |

Table 6.3: Limits when instructions and params are in global memory

*election* and less for *Random walk*, which is counterintuitive, because we expected the number of agents to rise. However, for the most complicated algorithm, *Firefly synchronization*, the number of possible agents did rise. Table 6.4 shows the limits in this configuration. The usage of the global memory is added in a separate column because the shared memory is not used at all.

| Algorithm | Max. ag. | Max. blk. | Sh. mem. | Glob. mem. |
|---|---|---|---|---|
| Leader election | 32 | 2800 | 0 kB | 128 MB |
| Random walk | 32 | 2800 | 0 kB | 111 MB |
| Firefly synch | 32 | 2800 | 0 kB | 168 MB |

Table 6.4: Limits when everything is in the global memory

### 6.2.5 Conclusions

This analysis shows that the *Random walk* algorithm uses the least of resources among the three algorithms, whereas *Firefly synchronization* uses the most, reflecting the complexity of the algorithms. In the case of *Random walk* agents have very few agent variables and less local variables are allocated, since this algorithm involves almost no calculations. *Leader election* uses few agent variables but a little more local variables are allocated. *Firefly synchronization* has both a lot of agent variables and involves a lot of calculations, therefore is is the most resource intensive.

The memory configurations that allow the largest simulations to be executed are the *shared memory intesive* and the *global memory intensive*. This is especially true for the *Firefly synchronization* algorithm, which is relatively complicated, and uses a lot of data. This is the reason why this algorithm was not able to run in some configurations in the first place.

We also see that the maximum number of blocks and the maximum number of threads are somewhat independent. The maximum number of blocks is limited by the shared memory usage, and the maximum number of threads

is limited by the register usage. The comparison is however not always straightforward, since the data layout is partly up to the compiler, and programmers cannot influence it.

## 6.3 Simulation execution time

In this section we analyze the dependence of the execution time on the simulation length (the number of iterations). These measurements reflect the impact of the different configurations on the execution speed. We discuss these results per algorithm.

The configurations presented in the Section 6.2 differ in the placement of variables: these variables that are placed in slower memories can lead to large time penalties. If a specific algorithm accesses these variables frequently, the entire execution might slow down substantially.

It has also been shown in Section 6.2 that different configuration can accomodate different model sizes and numbers, so in order to get a fair comparison, all the measurements were made at the lowest of these limits to allow all models to accomodate all configurations.

### 6.3.1 Random walk

Random walk is an algorithm in which the agents move in the direction of their current heading one step in each tick. The heading is changed randomly at the beginning of the next tick, whereas the step size remains constant. See Appendix A.2 for pseudocode of the algorithm.

Figure 6.1 shows the measurements for the *Random walk* algorithm. For the sake of brevity, *shared memory* denotes the *shared memory intensive configuration*, *register* corresponds to the *register intensive configuration*, *global instr.* stands for the *shared memory intensive configuration with instructions in the global memory*. Finally, *global everything* means the *global memory intensive configuration*.

It can be clearly seen that the *register intensive* configuration is the fastest, since registers have the lowest latency. The difference between the *shared memory intensive* configuration and the one where instructions are placed in the global memory is relatively small which is due to the fact that instructions and parameters are fetched relatively infrequently from the memory. Thus the *shared memory intensive configuration with instructions in global memory* should be used. It might be surprising that the execution speeds up if the instructions and parameters are moved into the

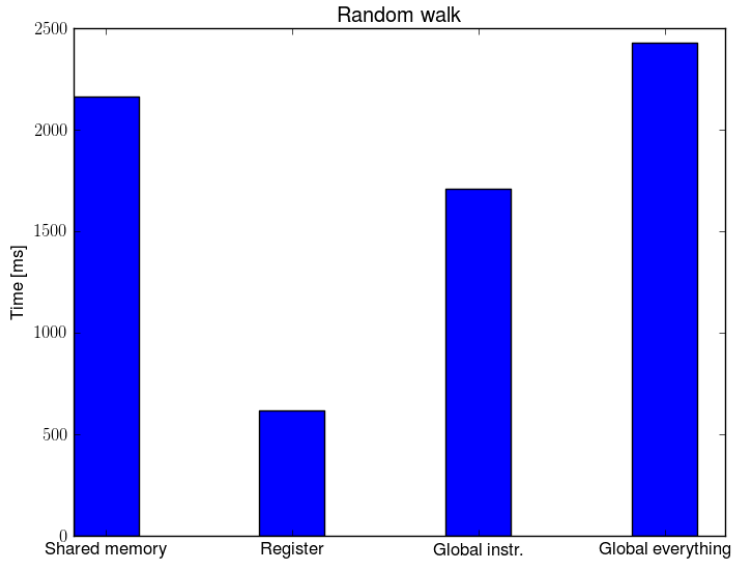


Figure 6.1: Execution time for random walk

global memory. A possible explanation is that agents are accessing this data simultaneously, and if the data is not well aligned, it can result in bus conflicts.

Finally if all the data is placed in the global memory the execution will be slower since the global memory is very slow compared to others.

### 6.3.2 Leader election

Leader election is an algorithm in which agents exist in a static network, in a ring topology. Since agents do not move, their spatial coordinates are not important and are initialized to some random value. Agents send their ID to their neighbor. At the same time they receive their neighbors ID, as well. If the received ID is smaller than their own they will set an internal flag that shows their leadership *true*, otherwise false. At the end of the algorithm only one agent will have the flag with the value *true*.

Figure 6.2 shows the results for the *Leader election* algorithm, the legend of the *x-axis* of the graph corresponds to the different configurations the same way it was described in Subsection 6.3.2. The relations between the different configurations are similar to those of the previous algorithm. The *global memory intensive* configuration is however a lot slower than in the

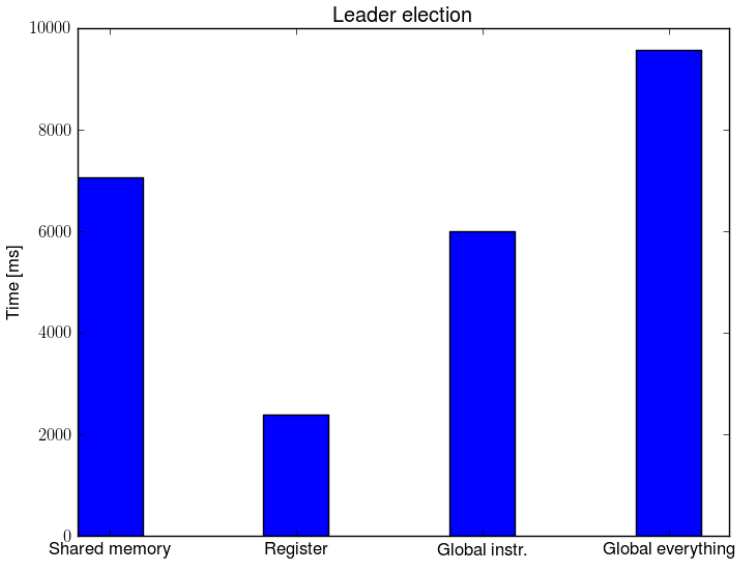


Figure 6.2: Execution time for leader election

previous case. This is due to the general slowness of the global memory.

### 6.3.3 Firefly synchronization

Figure 6.3 shows the measurement results for the *Firefly synchronization* algorithm. The two missing bars indicate that the algorithm could not be started in the respective configuration.

The placement of all the variables in the global memory causes a significant performance deteoriation. This is due to the fact that this particular algorithm includes a lot of operations with variables, and the number of instructions executed in each loop is a lot higher than with the rest of the algorithms.

## 6.4 Translation speed

In this section we present measurements that show the time needed by the translation process of the *CudaSimulator*. These measurements are necessary to be able to compare the performance of our *CudaSimulator* to alternative solutions since the translation time has to be added to the pure

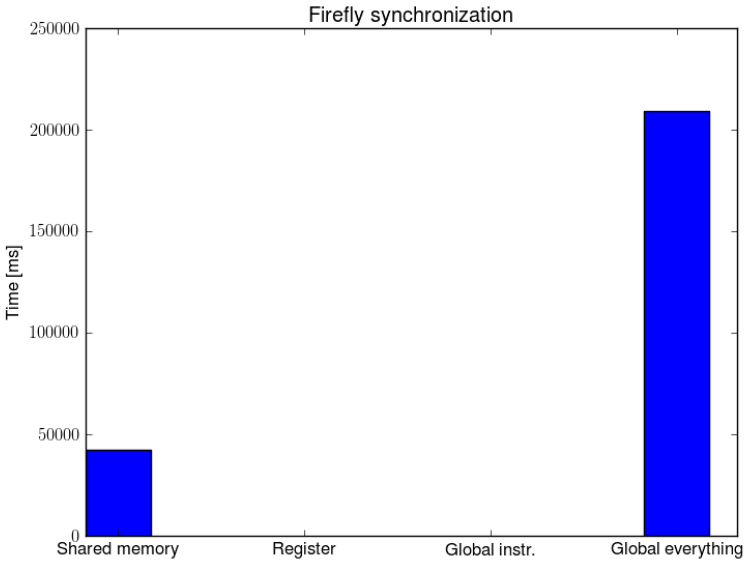


Figure 6.3: Execution time for firefly synchronization

execution time.

| **Algorithm** | **Translation time** | **Execution time** |
|---|---|---|
| Leader election | 6166 ms | 7057 ms |
| Random walk | 6309 ms | 2162 ms |
| Firefly synch | 6865 ms | 42426 ms |

Table 6.5: Time taken by the translation of the algorithms

Table 6.5 shows the execution time of the translation phase and the comparison with the execution times measuremed in the *shared memory intensive* configuration. The proportion of the translation time to the execution time is significant in the case of *Leader election* and *Random walk*. We note however, that since the engine can run different models at the same time, it is not fair to compare translation time directly with the total execution time. We will use the normalized execution time to yield a fairer comparison in Section 6.5.

We can see that there is very little difference between the translation time of different algorithms. Therefore it is advised to run longer simulations with many agents, in order to amortize the translation time. The slowness of the translation phase is the result of single core execution of *Scala* code. The

code that is processing the AST is written functionally with immutable data, which results in frequent allocations.

## 6.5 Comparison with alternative solutions

To decide whether our *CudaSimulator* is competitive alternative for performing fast simulations using NetLogo models, we compare its performance against two alternatives: the original *NetLogo* and the solution when simulations are written directly in CUDA (in the following we refer to it as *Pure Cuda*).

There are four measurements per method for each algorithm: 10, 100, 1000 and 10000 iterations. These points are represented by dots on the graphs, and lines connecting them were created by interpolation.

There are two sets of graphs in this section: The first set shows the total execution times that were measured running the whole simulation. For a fairer comparison, the second set of graphs shows the normalized execution times. Normalization means that in the case of the *CudaSimulator* and the *Pure Cuda* solutions, the execution times were divided by the number of simulations running in parallel, whereas the exectution times with the original *NetLogo* remain unchanged. Note that in the case of normalized diagrams the translation time was added after normalization.

### 6.5.1 Leader election

Figure 6.4 shows the total execution time in the case of the *Leader election* algorithm. While execution time is steadily rising when increasion the number of iterations, the execution time of the *Pure Cuda* solution is significantly smaller (by 2-3 orders of magnitude) than the *CudaSimulator*, because the overhead of the virtual machine (extra instructions, and frequent synchronization barriers between bytecode instructions) is not present.

For simulations with a few iterations the *CudaSimulator* performs sligthly better than the *NetLogo* engine. This is due to many factors that slow down the *NetLogo* simulation. For example, JIT optimization in *NetLogo*'s engine needs multiple executions of the same code to take effect. However, for larger numbers of iterations the pure *NetLogo* outperforms the *CudaSimulator*.

In contrast to the previous graph, Figure 6.5 indicates that the normalized execution times of the *CudaSimulator* are rising only very sligthly. In

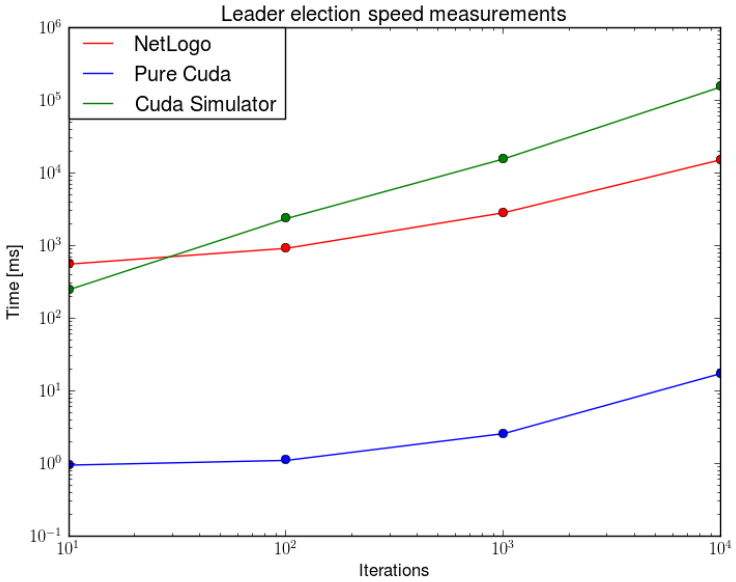


Figure 6.4: Comparison of different executions for leader election (note the logarithmic scales on both axes)

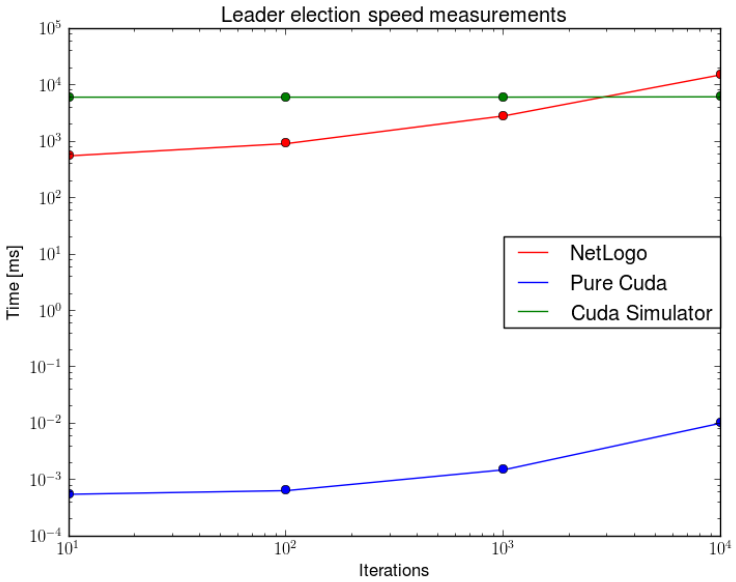


Figure 6.5: Comparison of different executions for leader election normalized (note the logarithmic scale on both axes)

fact, most of the time is due to the translation into bytecode in *CudaSimulator*. Therefore it becomes advantageous to use the *CudaSimulator* at larger number iterations.

The normalized execution time of the *Pure Cuda* solution is even smaller (the sub-microsecond execution times are represented by negative powers of ten on the logarithmic scale).

### 6.5.2 Random walk

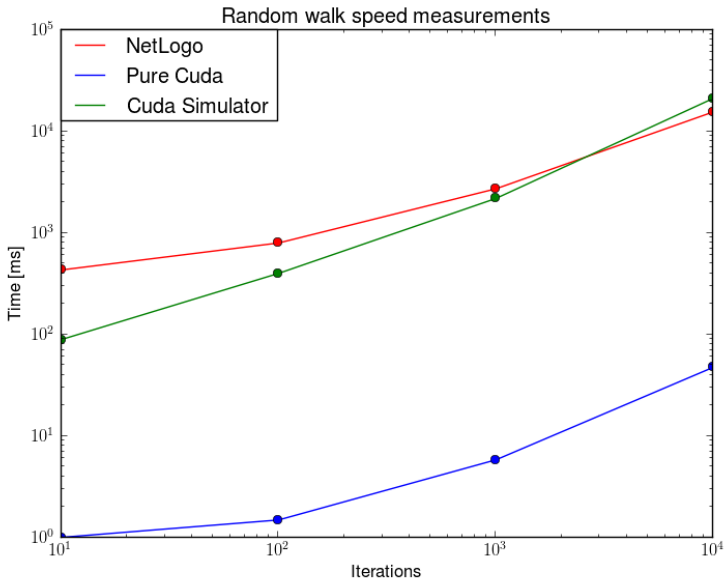


Figure 6.6: Comparison of different executions for random walk (note the logarithmic scale on both axes)

The measurements for the *Random walk* algorithm paint a similar picture to those for the *Leader election*. Figure 6.6 shows the total execution times. The difference compared to the previous algorithm is that the "crossover" between the *CudaSimulator* and *NetLogo* occurs for higher iterations (i.e. where the *CudaSimulator* becomes faster).

The normalized execution times are again close to the translation time (see Figure 6.7). The execution time appears to be constant because of the logarithmic nature of the graph, in reality there is a small increase. Therefore the *CudaSimulator* is not expected to be faster than the *Pure Cuda* solution for very large iterations.

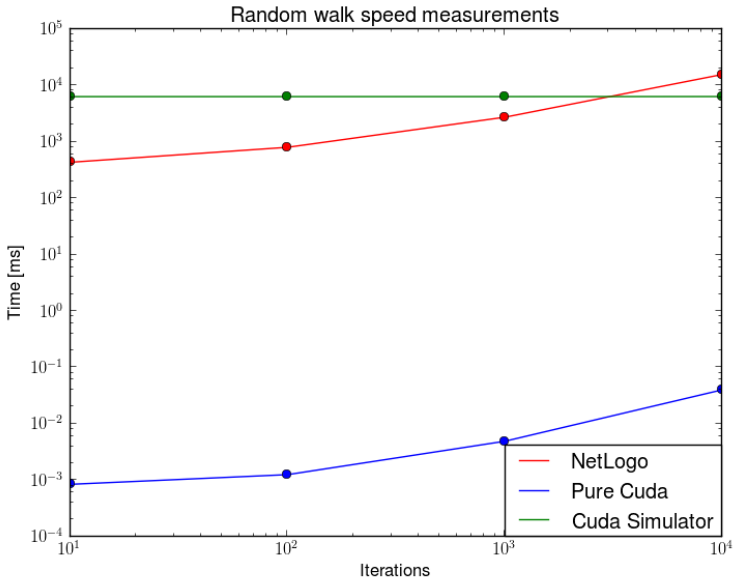


Figure 6.7: Comparison of different executions for random walk normalized (note the logarithmic scale on both axes)

We can observe that in the case of the normalized execution time the crossover between *NetLogo* and the *CudaSimulator* happens in the same interval (around 3000 iterations) as in the case of *Leader election*.

### 6.5.3 Firefly synchronization

The measurement results for the *Firefly synchronization* algorithm are depicted on Figures 6.8 and 6.9. The differences between the three solutions are more significant in this case.

First, the execution time for *CudaSimulator* is larger than all the other solutions even for few iterations. Above 1000 iterations the execution time was too long to measure, so the measurement at 10000 iterations is generated by interpolation (represented by the dashed line).

It is also interesting to note that *PureCuda* execution time is also substantially larger as compared to previous algorithms which reflects the complexity of the algorithm.

The normalized execution times show that below 1000 iterations, *Cud-*

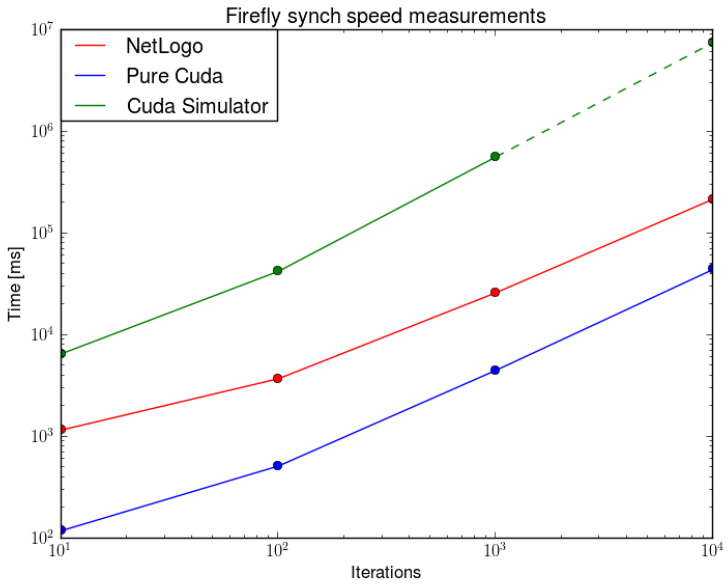


Figure 6.8: Comparison of different executions for firefly synchronization (note the logarithmic scale on both axes)

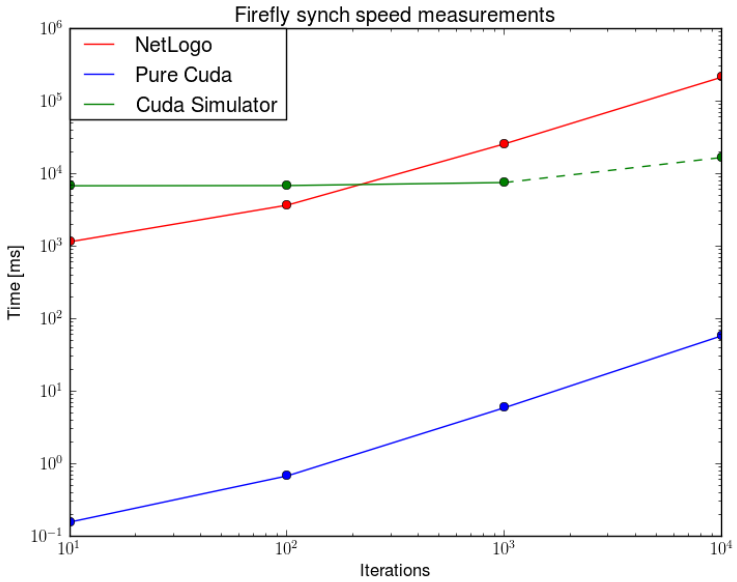


Figure 6.9: Comparison of different executions for firefly synchronization normalized (note the logarithmic scale on both axes)

*aSimulator* consumes most of the time to translate the simulations. Abover 1000 iterations the execution time starts to rise slightly, however the rate of increase is smaller than that of the *NetLogo* engine.

The crossover point between *NetLogo* and *CudaSimulator* is at lower iterations (between 100 and 1000) which again is due to the fact that JIT needs a number of iterations to start to have an effect.

## 6.6 Pathological models

In this section we examine different ways to write *NetLogo* models which are limiting the performance of *CudaSimulator*.

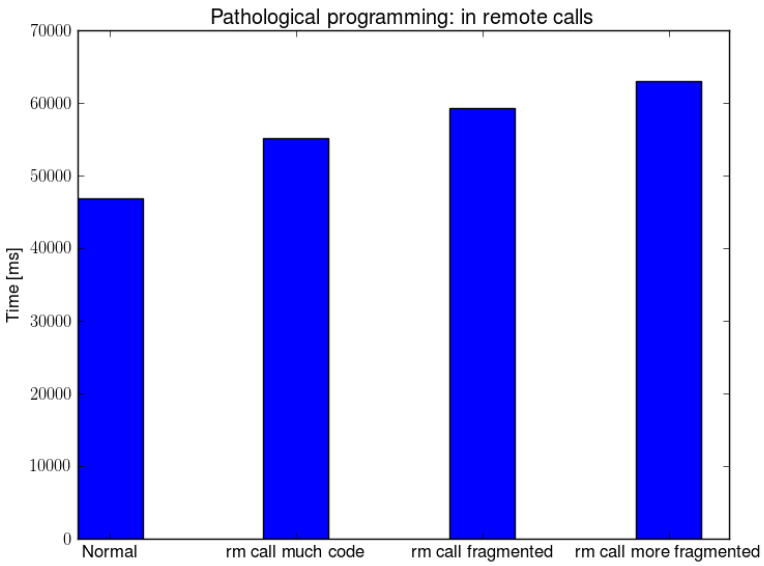


Figure 6.10: Pathological programming: remote calls

The first set of measurements shows the way remote calls are used. Three different scenarios are shown in Figure 6.10. Each scenario is created by slightly modifying the *firelfy synchronization* model by adding code to the part where agents exchange information with others in a remote call. The measurements were taken at 1000 iterations in order to magnify the differences.

The *normal* scenario is created by simply adding extra code outside of the body of the remote call. This version is to compare to others such that

the amount of the executed code is always the same.

The scenario called *rmcall much code* is created by moving the extra code inside of the remote call. The increase in latency is visible, but not very significant.

The scenario called *rmcall fragmented* is created by putting the extra code a separate remote call. This increases the execution time even further, as a remote call is an expensive operation. In order to test the effect of fragmentation, the extra code is divided into two parts each in a different remote call. The result is shown on the bar titled *rm call more fragmented*.

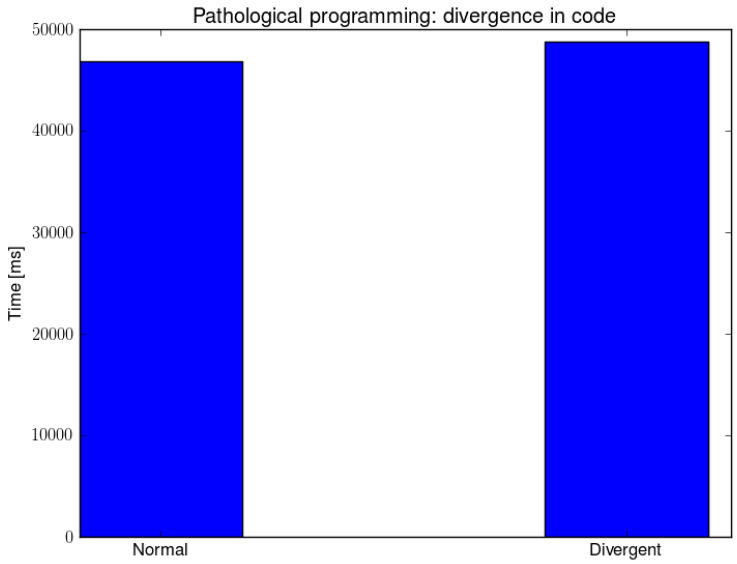


Figure 6.11: Pathological programming: divergence

Figure 6.11 represents the second set of measurements which examines the effect of divergence (branching code).

Since CUDA executes instructions on a SIMD architecture (the same machine instruction is executed on different data), divergent instructions have to be serialized, and threads in the other branch are simply disabled. As a result of this execution time might rise significantly. Measurement has been made by modifing the code such that the extra code is duplicated in two branches. As it can be seem from the *Divergent* bar, the latency grows slightly.

The last scenario is to put the remote call instruction in one branch of an

*ifelse* instruction. As a result the simulation hangs because of the remote call mechanism synchronizes the threads first. In CUDA this results in a deadlock since the threads in the other branch never meet the thread barrier.

# Chapter 7

# Conclusions and Future Work

This chapter concludes this thesis. In section 7.1 we give a brief overview of the features of the *CudaSimulator*. We also compare its performance to alternative means of running *NetLogo* simulations. We finally provide an outlook of possible ways to improve the simulator we proposed in this thesis.

## 7.1 Conclusions

This section gives a brief overview of the salient features of the *CudaSimulator* which are then compared to alternatives.

### 7.1.1 Overview of features

*CudaSimulator* was integrated in the *Snowdrop* project as a back-end to the *MetaCompiler* [38] in order to accelerate the evaluation of programs generated by *genetic programming*.

The simulator consists of two parts: one is an extension of the *NetLogo* engine and translates the model written by the user into a special bytecode we have designed. The bytecode is then executed by the other part which is written in CUDA C and runs directly on the GPU. The results of the execution is then collected by the former part.

In our design, the capabilities of the *NetLogo* engine to compile models can be initially reused. Then, the engine is bypassed and the execution is rerouted to the part that is running on the GPU.

This thesis answered three research questions:

- **Can *NetLogo* models efficiently translated to CUDA?** We investigated the mapping of simulations and agents to CUDA execution units. We found that due to dependencies and synchronization possibilites between blocks and *Streaming multiprocessors*, individual simulations are executed on blocks. We investigated the mapping of *NetLogo* variables to different types of memory. Therefore *global variables* (which are accessed infrequently) are placed into the slower global memory. Exclusive agent variables are placed in the local memory and shared agent variables are placed in the shared memory in order to facilitate fast communication. We have seen that the compiler generates an AST representation of *NetLogo* models. Most of the instructions can be directly translated into bytecode, with a few exceptions (for example, *aggregation functions*).

- **How should *NetLogo* be changed in order to accomodate the use of GPU architecture?** We have found that *NetLogo* consists of a compiler and an execution engine. We found that it is possible to disable the execution engine and use the compiler to create an AST representation. With additional modules the AST can be translated into bytecode, which is then passed to the virtual machine. This way we could reuse a large portion of *NetLogo*'s code and thus avoid redundancy. We also imposed restrictions on this engine (only one breed can be used, only one type of metric space is implemented) in order to simplify development.

- **Limitations of *CudaSimulator:*** We identified three common algorithms used in MANETs to characterize the performance of our simulator. We examined the size constraints of *CudaSimulator* i.e. how many blocks and threads can be launched simultaneously. We found that the limitations depend on the different ways the data can be placed in the memory of the GPU.We also compared the the *CudaSimulator* with the original *NetLogo* and simulations directly implemented in CUDA, with respect to performance. We found that when running a large number of smaller sized simulations, *CudaSimulator* has a better performance than *NetLogo*, however direct CUDA solutions are always faster.

### 7.1.2 Comparison to alternatives

In Chapter 6 we compared the performance of the *CudaSimulator* to two major alternatives: using the original *NetLogo* simulator and writing the simulations in CUDA C.

The measurements show that the simulation time for the *CudaSimulator* is generally larger than that of the original *NetLogo* program. However if we take into account the simulations running in parallel on the *CudaSimulator* it is easy to see that the time is amortized among the models running in parallel. In other words, our *CudaSimulator* is a high-throughput solution, but is not very useful as a low latency solution. As a result of this, when running many models, there is a point where *CudaSimulator* begins to outperform *NetLogo*.

The measurements in Chapter 6 show that if the models are re-implemented in CUDA C, the performance is always better, usually by several orders of magnitude. This is because it is possible to write highly optimal code in CUDA C. Also, the overhead of the virtual machine in *CudaSimulator* is not present in pure CUDA C solutions.

The above observations present a trade-off in which the user has multiple choices. If the focus is on performance, then the best solution is to rewrite the models in CUDA. If the focus is on simplicity, the user should use *NetLogo*. For a compromise solution *CudaSimulator* might be a good choice. The user will have to avoid certain programming patterns and commands that are not implemented, but they might gain substantial performance in the form of very high throughput.

## 7.2 Future Work

This thesis outlines a proof-of-concept version of a software system. There are a lot of possible ways to develop this project further. In this section we give an overview of some of these ways.

The first and most obvious way to develop the *CudaSimulator* is to widen its support of *NetLogo* commands and concepts. Currently only one agent type is supported whereas *NetLogo* has four: turtles, links, patches and the observer, each capable of executing code. There are some widely used algorithms that use multiple types of agents, e.g. usual implementations of *ant colony optimization* use patches.

Next, run-time error checking should be implemented. Genetic Programming guarantees that the generated code will be syntactically correct, but it does not account for the meaningfulness of it. This means that run-time errors can occur in the code, e.g., division by zero. Running the code in *NetLogo* the JVM will catch these errors and generate exceptions. Contrary to that, *CudaSimulator* will generate wrong results or *NaN* values instead.

Finally, we have seen that the simulations are severely limited by their resource usage. As we have seen in Chapter 4, a separate variable is allocated to store the results of each arithmetic/logic operation. This results in a lot of allocations, most of them used only once. Therefore the resource usage could be significantly improved. Thus, the *CudaSimulator* simulations could be larger.

Overall, we believe our *CudaSimulator* can be an efficient solution for high-throughput applications that use *NetLogo* models. However, for cases where low latency is necessary a different design might be needed.

# Bibliography


[1] CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[2] Hoomd-blue web page. http://codeblue.umich.edu/hoomd-blue.

[3] Netlogo: Ask-concurrent documentation. http://ccl.northwestern.edu/netlogo/docs/programming.html#ask-concurrent.

[4] Joshua A. Anderson, Christian D. Lorenz, and Alex Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342–5359, May 2008.

[5] J. Beal and J. Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *Intelligent Systems, IEEE*, 21(2):10 – 19, march-april 2006.

[6] M. J. Berryman and S. D. Angus. Tutorials on agent-based modelling with netlogo and network analysis with pajek. http://ccl.northwestern.edu/papers/2010/chapter_ABM_NA.pdf.

[7] Tracy Camp, Jeff Boleng, and Vanessa Davies. A survey of mobility models for ad hoc network research. *Wireless Communications and Mobile Computing*, 2(5):483–502, 2002.

[8] Wenan Chen, Kevin Ward, Qi Li, Vojislav Kecman, Kayvan Najarian, and Nathan Menke. Agent based modeling of blood coagulation system: Implementation using a gpu based high speed framework. In *Engineering in Medicine and Biology Society,EMBC, 2011 Annual International Conference of the IEEE*, pages 145 –148, 30 2011-sept. 3 2011.

[9] N.T. Collier and M.J. North. Parallel agent-based simulation with repast for high performance computing. *Simulation: Transactions of the Society for Modeling and Simulation International*.

[10] Matthew Dickerson. Multi-agent simulation and netlogo in the introductory computer science curriculum. *J. Comput. Sci. Coll.*, 27(1):102–104, Oct. 2011.

[11] R. M. D'Souza, M. Lysenko, and K. Rahmani. Sugarscape on steroids: Simulating over a million agents at interactive rates. *Proceedings of the Agent 2007 Conference, Chicago, IL*, 2007.

[12] Matt Duckham. *Decentralized spatial computing*. Springer, 2013.

[13] U. Erra, B. Frola, V. Scarano, and I. Couzin. An efficient gpu implementation for large scale individual-based simulation of collective behavior. In *High Performance Computational Systems Biology, 2009. HIBI '09. International Workshop on*, pages 51 –58, oct. 2009.

[14] Center for Connected Learning and Northwestern University Computer-Based Modeling. https://github.com/NetLogo/NetLogo.

[15] Greg N. Frederickson and Nancy A. Lynch. Electing a leader in a synchronous ring. *J. ACM*, 34(1):98–115, Jan. 1987.

[16] Snowdrop Project @ googlecode. https://code.google.com/p/snowdrop/.
[17] Michael Garland Kevin Skadron John Nickolls, Ian Buck. Scalable parallel programming with cuda. *ACM Queue*, 6(2):40–53, March/April 2008.
[18] Steffan Karger. An embedded spatial computing platform for interactive environments. MSc thesis in Embedded systems, Delft University of Technology, 2012.
[19] S. Karger, A. Di Figlia, M. Bos, A. Pruteanu, and S. Dulman. Spatial computing for non-it specialists. *Spatial Computing 2012 colocated with AAMAS*, page 33, 2012.
[20] S. Levy and U Wilensky. How do i get there...straight, oscillate or inch? high-school students' exploration patterns of connected chemistry. *2007 annual meeting of the American Educational Research Association, Chicago, IL*.
[21] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.
[22] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.
[23] S. Luke et al. Mason: A new multi-agent simulation toolkit. SwarmFest Workshop, 2004.
[24] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The swarm simulation system: A toolkit for building multi-agent simulations. 1996. Santa Fe Institute, Santa Fe.
[25] R. Mirollo and S. Strogatz. Synchronization of pulse-coupled biological oscillators. *SIAM Journal on Applied Mathematics*, 50(6):1645–1662, 1990.
[26] Radhika Nagpal and Marco Mamei. Chapter 1 engineering amorphous computing systems.
[27] M.J. North, T.R. Howe, N.T. Collier, and J.R. Vos. A declarative model assembly infrastructure for verification and validation. In S. Takahashi, D. L. Sallach, and J. Rouchier, editors, *Advancing Social Simulation: The First World Congress*.
[28] NVIDIA. Nvidia's next generation cuda compute architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.
[29] Martin Odersky. The scala language specification. http://www.scala-lang.org/archives/downloads/distrib/files/nightly/pdfs/ScalaReference.pdf, January 2013. version 2.8.
[30] K.S. Perumalla. Efficient execution on gpus of field-based vehicular mobility models. In *Principles of Advanced and Distributed Simulation, 2008. PADS '08. 22nd Workshop on*, page 154, june 2008.
[31] K. S. Perumalla et al. Data parallel execution challenges and runtime performance of agent simulations on gpus, November 2007. Oak Ridge National Laboratory.
[32] Steven Railsback. *Agent-based and individual-based modeling : a practical introduction*. Princeton University Press, Princeton, 2012.
[33] Jerry M Rhee and Philip M Iannaccone. Mapping mouse hemangioblast maturation from headfold stages. *Developmental Biology*, 365(1):1–13, 2012.
[34] J. Schellinck and Tony White. Use of netlogo as a rapid prototyping tool for the creation of more rigorous spatially explicit individual-based biological models. In *Open International Conference on Modelling and Simulation-OICMS 2005. Blaise Pascal University, France*, 2005.

[35] Pratim Sengupta and Uri Wilensky. On learning electricity with multi-agent based computational models (niels). In *Proceedings of the 8th international conference on International conference for the learning sciences - Volume 3*, ICLS'08, pages 123–124. International Society of the Learning Sciences, 2008.

[36] F. Sondahl, S. Tisue, and U. Wilensky. Breeding faster turtles: Progress towards a netlogo compiler. Agent 2006, Chicago, IL., 2006.

[37] S. Tisue and U. Wilensky. Netlogo: Design and implementation of a multi-agent modeling environment. SwarmFest, Ann Arbor 2004, 2004.

[38] Sjors van Berkel. Automatic discovery of distributed algorithms for large-scale systems. MSc thesis in Computer Science, Delft University of Technology, 2012.

[39] Sjors van Berkel, Daniel Turi, Andrei Pruteanu, and Stefan Dulman. Automatic discovery of algorithms for multi-agent systems. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, GECCO Companion '12, pages 337–344, New York, NY, USA, 2012. ACM.

[40] J.A. van Meel, A. Arnold, D. Frenkel, S.F. Portegies Zwart, , and R.G. Belleman. Harvesting graphics power for md simulations. *Molecular Simulation*, 34(3):259–266, 2008.

[41] Harvey Whitehouse, Ken Kahn, Michael E Hochberg, and Joanna J Bryson. The role for simulations in theory construction for the social sciences: case studies concerning divergent modes of religiosity. *Religion, Brain & Behavior*, 2(3):182–201, 2012.

[42] Y. Yan, M. Grossman, and V. Sarkar. Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 887–899. Euro-Par 2009 Parallel Processing, Springer Berlin / Heidelberg, 2009.

[43] Franco Zambonelli and Marco Mamei. Spatial computing: An emerging paradigm for autonomic computing and communication. In Michael Smirnov, editor, *Autonomic Communication*, volume 3457 of *Lecture Notes in Computer Science*, pages 44–57. Springer Berlin Heidelberg, 2005.

# Appendix A

# Test algorithms

## A.1 Leader election

**Algorithm 1** Leader election

```
1:  procedure SETUP
2:      if myID ≠ 0 then                ▷ Connect to neighbor (one-direction)
3:          connectTo(myID − 1)
4:      else
5:          connectTo(nrOfAgents − 1)
6:      end if
7:      inMsg ← 0
8:      outMsg ← 0
9:      isLeader ← false
10:     ticks ← 0
11: end procedure
```

**Algorithm 1** Leader election (Continued)

```
1: procedure GO
2:     if ticks > maxTicks then
3:         exit
4:     end if

5:     neighbor.inMsg ← outMsg

6:     if inMsg > myID then
7:         outMsg ← inMsg
8:     else
9:         outMsg ← myID
10:        if inMsg = myID then
11:            isLeader ← true                    ▷ I am the leader
12:        end if
13:    end if

14:    ticks ← ticks + 1
15: end procedure
```

## A.2 Random walk

**Algorithm 2** Random walk

1: **procedure** SETUP
2: $myPosX \leftarrow rand()$ mod $MAX\_X$
3: $myPosY \leftarrow rand()$ mod $MAX\_Y$
4: $heading \leftarrow 0$
5: **end procedure**

6: **procedure** GO
7: **if** $ticks > maxTicks$ **then**
8: **exit**
9: **end if**

10: $heading \leftarrow rand()$ mod 360
11: $myPosX \leftarrow myPosX + STEP * \cos(\frac{\pi * heading}{180})$ ▷ move in X direction
12: $myPosY \leftarrow myPosY + STEP * \sin(\frac{\pi * heading}{180})$ ▷ move in Y direction

13: $ticks \leftarrow ticks + 1$
14: **end procedure**

## A.3 Firefly synchronization

**Algorithm 3** Firefly synchronization

```
1: procedure SETUP
2:     myXPos ← rand mod MAX_X
3:     myYPos ← rand mod MAX_Y
4:     myColor ← 0
5:     updateEnabled ← 0
6:     clock ← rand mod ROUND_LEN
7:     prevTickClock ← 0
8:     ticksClockDiff ← 0
9: end procedure

10: procedure GO
11:     if ticks > maxTicks then
12:         exit
13:     end if

14:     moveRandomWalk()
15:     clearConnections()
16:     reconnectWithinRange(TRANSMIT_RANGE)        ▷ bi-direcitonal
    connections

17:     if isNeighbCountSwitchedOnLargerThan() then
18:         if clock > ON_TIME then
19:             clock ← ON_TIME
20:         end if
21:     end if

22:     updateColor()
23:     incrAgentsClock()
24:     updateTicksClockDiff()
25:     prevTickClock ← clock                       ▷ store clock in prev. tick

26:     ticks ← ticks + 1
27: end procedure
```

**Algorithm 3** Firefly synchronization (Continued)

```
procedure updateColor
    if clock < GLOW_TIME then
        myColor ← ON_COLOR
    else
        myColor ← OFF_COLOR
    end if
end procedure
procedure incrAgentsClock
    if clock < ROUND_LEN then
        clock ← clock + 1
    else
        clock ← 0
    end if
end procedure
procedure updateTicksClockDiff
    if abs(clock − prevTickClock) = ROUND_LEN then
        ticksClockDiff ← 0
    else
        if clock < prevTickClock then
            ticksClockDiff ← 1
        else
            ticksClockDiff ← abs(clock − prevTickClock − 1)
        end if
    end if
end procedure
```