Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelor Thesis

GPU Support for Software Defined Networking Controllers

submitted by

Tobias Theobald

on June 2nd, 2014

Supervisor

Prof. Dr.-Ing. Thorsten Herfet

Advisor

Dipl.-Ing. Michael Karl

Reviewers

Prof. Dr.-Ing. Thorsten Herfet
Prof. Dr.-Ing. Philipp Slusallek

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

Bachelor Thesis

UNIVERSITÄT DES SAARLANDES

Lehrstuhl für Nachrichtentechnik

FR Informatik

Prof. Dr. Th. Herfet

Universität des Saarlandes
Campus Saarbrücken
C6 3, 10. OG
66123 Saarbrücken

Telefon (0681) 302-6541
Telefax (0681) 302-6542

www.nt.uni-saarland.de

Bachelor Thesis
Tobias Theobald
Mat. Nr.: 2526256

GPU Support for Software Defined Networking Controllers

Modern communication networks become a highly critical part of global infrastructure. Billions of devices are connected over a set of heterogeneous and complex network segments, carrying millions of gigabytes across this global backbone. According to forecasts, the predominant portion of this traffic has its origin in multimedia applications that exchange audio visual content. This type of information has completely different requirements to the communication infrastructure than the initially used applications for pure text exchange in the beginnings of the Internet.

Networks continuously increase in size and complexity and adjust to the constantly arising requirements of new applications. The advent of Software Defined Networking (SDN) provides a new concept of centrally managed network logic and separated networking planes for both, controlling and pure forwarding. This eases the development of new and more effective network technologies as required by multimedia applications.

Routing of packets from one client to another thus unveils as a highly complex computational task since big networks contains a large number of possible connection paths between any two source-sink pairs. With SDNs, the task of finding appropriated paths in the network is completely left to the central controlling instance. Thus, it is critical to quickly perform management and routing actions. A possible approach is to massively parallelize operations. A suitable way is to transfer complex and expensive calculation on highly specialized hardware, such as GPUs.

This thesis focuses on the development of a controller instance for Software Defined Networks based on OpenCL technology for highly parallelized computing. In particular, the thesis includes the following tasks:

- Discussion of benefits and limitations of SDN components related to GPU processing.
- Implementation of GPU support for SDN controller.
- Performance evaluation of GPU- and CPU-based controller instances.

Advisor:

Dipl.-Ing. Michael Karl

Supervisor:

Prof. Dr.-Ing. Thorsten Herfet

Abstract

With the rise of network usage and Software Defined Networking (SDN), controllers that respond quickly become increasingly important. However, with increasing network size, these controllers need to process an ever growing amount of data. But as more and more powerful GPUs are cheaply available and usable for massively parallel computations, some problems of SDN may be suitable for GPU acceleration. This paper shows the basic concepts of such controllers and a concrete implementation in details. Afterwards, it evaluates the performance and applications in practice.

Contents

Chapter 1

Introduction

Computer networks have become a very large part of nearly every company. This is mainly due to the companies having more and more tasks automated by machines, that all have to be controlled somehow, get information from somewhere or report to a certain authority within or even outside the company. However, this dependency on network connections means, that hardly anything works when network fails. Of course, if the whole network fails, there is not much that can be done, but if only parts of the hardware fails, the network ought to be able to detect this problem and work around it. In order for this to happen, the network must be aware of its structure [13] and be able to detect a failing component, ideally before it is too late. Then, all traffic that was routed via the failed component has to be rerouted through different components, ideally within a very short time. Monitoring the network equipment and making all components in the network redundant is the key to this.

Another important task in this context is the configuration of the network equipment. While certain manufacturers have tools to do exactly that [32], it is hard to find a standard that more than one manufacturer implements. This is, however, important, as networks seldom use equipment from only one supplier for redundancy reasons. There are protocols that try to fill this gap, e.g. the *Simple Network Management Protocol (SNMP)* [14], which can be used to set most configuration options in switches, as well as monitor the topology. Unfortunately though, it lacks the flexibility that another currently up-and-coming technique called Software Defined Networking, offers.

Software Defined Networking(SDN) offers a high degree of freedom with respect to the configurability of the networking hardware. It enables network administrators to change the way, that network packets travel. It is even possible to modify the header of a packet or drop it entirely and has the ability to turn any SDN switch into a firewall or a router. Of course, the exact capabilities, depend on the specific API that is used to communicate with the switches.

One of the most prominent APIs for Software Defined Networking, is *OpenFlow* [33]. It allows the administrator to concentrate the control over the network on to a single, centralized server called *controller*, a central server that is sent every unknown packet and which can then dynamically decide where it goes. As every switch in such a network is connected to the central authority, it is able to detect switch failures and other topology changes at once and can act accordingly. Another feature is, that the controller has global knowledge over its network, which it can use to make better decisions when routing packets through the network.
Unfortunately though, these controllers can easily be overloaded when dealing with a large network or a great number of packets, because deciding the routes that these take through network is computationally expensive. With that challenge, an efficient way to load off this work has to be found. One candidate for this is massively parallel computing on GPUs. This technique allows programs to take advantage over the up to several thousand compute cores that are used on regular consumer level graphics cards. Another possibility to offload this work is using other controllers, which would also eliminate the single point of failure that having only one controller implies, but this work focuses on the former solution. More on the latter solution may be found in the section 1.1.
Graphics cards are specialized hardware that was, up to a certain point in time, only used in graphics environments, but the last years have shown a sharp increase in use of this hardware for high-performance computing. They have several gigabytes of dedicated memory, that the graphics processing unit can use. It has a large number of cores that can act in parallel and are specialized on executing the same instructions on every core. This leads to the necessity to design the algorithms used on GPUs specially for them.
With this large amount of computing power at hand, this thesis takes some of the load from the controller and puts it off to the graphics cards. For this, a controller that meets this goal is designed and implemented and put to the test to evaluate the feasibility of this method.

1.1 Related Work

- *Flowvisor* [1] is a software that acts as an intelligent proxy for multiple controllers. While switches connect to FlowVisor, it connects to every controller once for every switch. FlowVisor can then devide the network into multiple logical or physical slices. Controllers can then be assigned to specific slices, e.g. one slice might be a specific subnet. This functionality can e.g. provide load balancing functionality or to create fail-safety by using multiple controllers for the same slice.

- R. Yanggratoke [10] uses batch processing of packets on the GPU by implementing a learning switch on top of Software Defined Networking. The requests for where the packets have to be sent, are buffered and the processed as a batch. This has an impact on the performance of the controller as well, but a controller may be more

complex than implementing a learning switch, which makes a change in the approach necessary.

1.2 Structure

This thesis is structured as follows:
Chapter 2 contains background information about Software Defined Networking, specifically in the form of the OpenFlow API. It then continues to introduce GPU computing and two of the most used frameworks for that, CUDA and OpenCL, by giving an example and pointing out differences between developing programs for the CPU and for the GPU.
Chapter 3 takes a detailed look at the components that a typical SDN controller consists of and determines the feasibility for GPU acceleration of each component.
Chapter 4 describes the implementation details of the GPU accelerated SDN controller that is created for this thesis, how the components interact with each other, which components actually use the GPU and which algorithms are used.
In *Chapter 5*, the performance of the implementation is evaluated with regards to several criteria and under different conditions.
Chapter 6 concludes the thesis.

Chapter 2

Background

In order to be able to implement a GPU-accelerated SDN controller, the basics of Software Defined Networking and GPU computing first have to be understood to evaluate which kinds of problems can be meaningfully accelerated by the GPU.

2.1 Software Defined Networking

The big difference of *Software Defined Networking* (SDN), as opposed to the classical approach to networking, is the full separation of the so-called data plane from the control plane (fig. 2.1). The data plane is the part of the switch that receives the packets and outputs them on a specific port, while the control plane decides, usually depending on the information in the packet header, to which port these packets are actually output.
On a classical network switch, both of these planes are available on device and used for forwarding packets. The switch "learns" which MAC addresses are behind which port and saves this information in a *source address table*. When a packet then comes in for switching, a lookup in this table is performed in order to determine, on which port the particular packet is output (fig. 2.2) [26].
As opposed to that, SDN separates the control plane from the device and centralizes it onto one single server, which is called a *controller*. Every SDN-enabled switch in the network is connected to the controller and regularly reports information about itself and the links to other devices to it, which leads to the controller having a complete overview over the network topology. Therefore, it can quickly react to device or link outages. In order to that, it has several possibilities to control the flow of packets through the network, such as setting routes along several switches or, in case of an emergency, forwarding packets itself. But most modern switches, that are used in professional environments, such as data centers and large companies, have more features than just forwarding packets. They can use e.g. act as load balancing switches, communicate their status to a central authority, support VLAN tagging and advanced routing policies. Additionally, some of these capabilities are

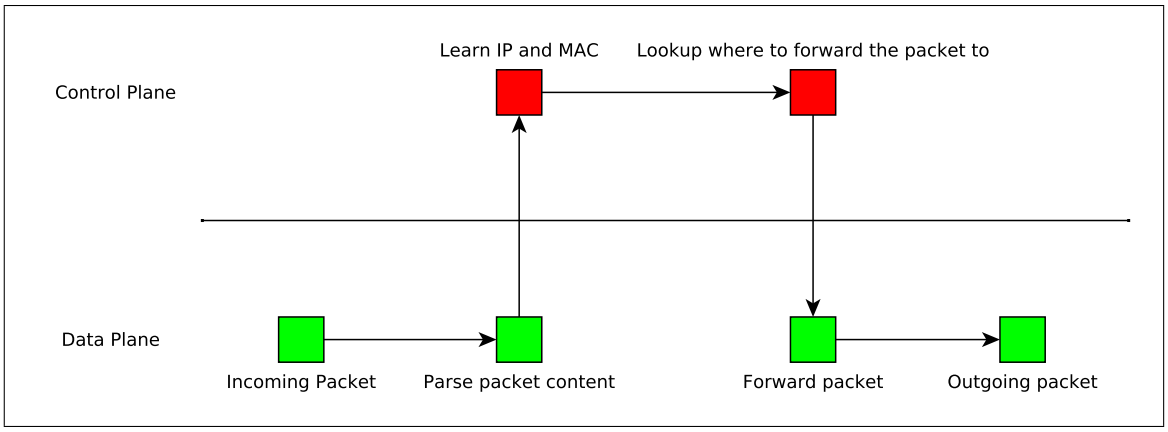

Figure 2.1: Data and control planes of a switch.

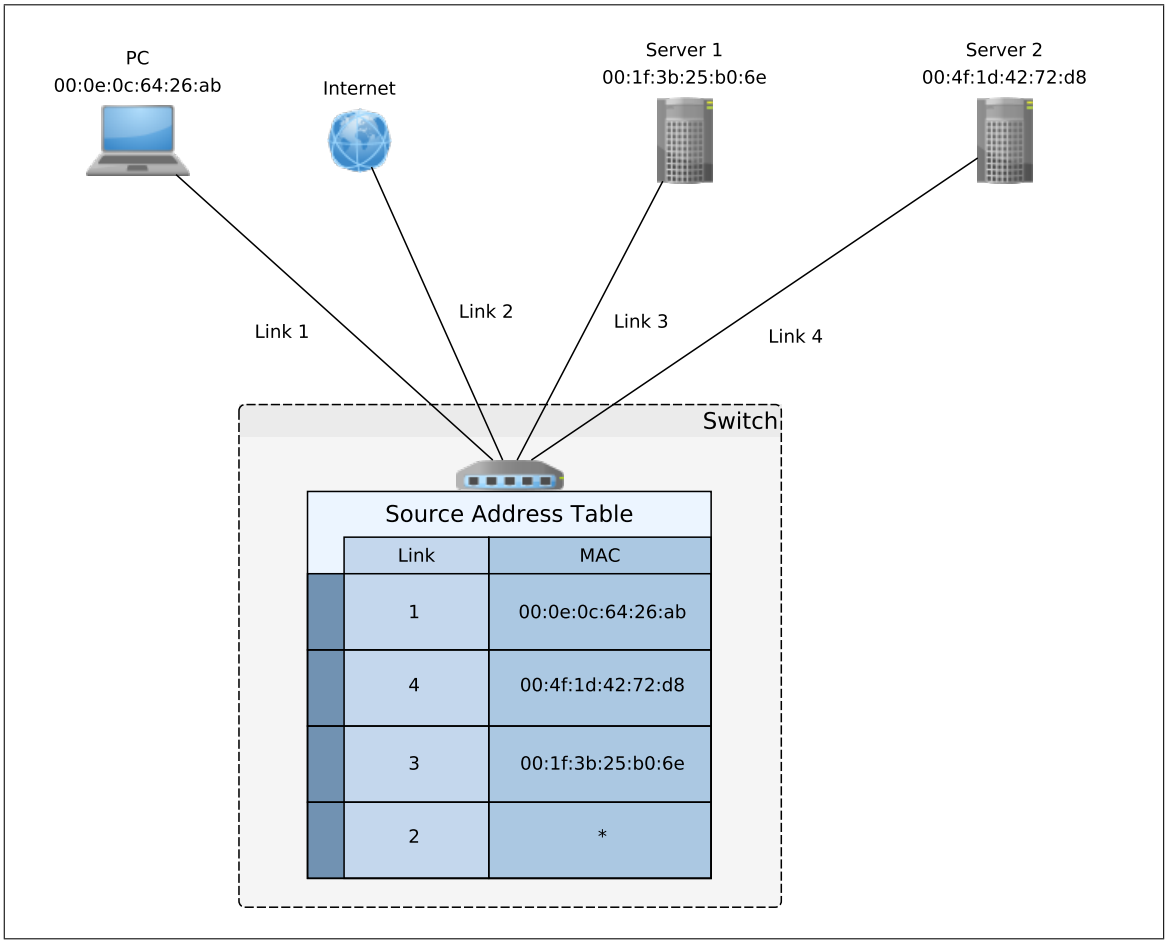

Figure 2.2: Conventional switch with source address table.

based on proprietary algorithms and may therefore only be supported by certain vendors. One example for that is link aggregation, which is specified in IEEE 802.1AX [12]. However despite that fact, there are also proprietary implementations by network equipment manufacturers, e.g. Cisco's EtherChannel [30] and ZTE's SmartGroup [31].
In order to be able to implement a superset of this functionality, the API that defines the communication between switches and the controller must be appropriately powerful. It should allow the controller to modify every aspect of the switches according to what is needed or even be able to replace some functionality as a whole. There are currently several SDN APIs available, such as Frenetic [3] and NETCONF [15], but this thesis focuses on OpenFlow [33], because there have been tremendous developments in the last years and because Saarland University already offers a fully functioning OpenFlow testbed.

2.1.1 OpenFlow

OpenFlow is an open SDN API, that can be freely used and implemented without fees under a BSD-like license [34]. It is maintained by an industry consortium, called the *Open Networking Foundation* [22], a non-profit organization to further the development of software defined networking. Its members include Cisco, Juniper, Google, Microsoft, Deutsche Telekom, Level 3 and Ericsson [35].
The OpenFlow specification defines a set of commands and replies that can be used to communicate with several switches and e.g. change settings on them and communicate link property changes or statistical data. For more information, see the OpenFlow specification in reference [36]
An integral part of the OpenFlow API is the concept of *flows*. A flow is basically a set of partly specified fields in the headers that match a certain pattern. Such a pattern is called a flow match and may contain a number of wild cards. It can be used to hit e.g. all TCP packets that go from one host to another or all UDP packets between two defined hosts and ports by defining some information on OSI layers 2 through 4. Switches have the ability to match packets to a flow, based on a so called flow table, that saves flow matches and one or more corresponding flow actions. These actions define what the switch has to do with an incoming packet that match the rule. Examples for such actions are "Output to port 2", "Rewrite source IP to 123.123.123.123", "Send to controller" or simply "Ignore". Using these rules, the switches are then able to decide which actions to take on which packets (see fig. 2.3).
The API also allows the controller to keep track of several other events and statistics, such as when a switch connects or disconnects, how many packets are matched by a certain flow rule and what the link type and the status of a switch's ports are.
As this is a very powerful API, several different types of controllers can be thought of. One may be a kind of controller that decides what to do with a packet on a per packet basis. Then every packet would be sent to the controller, which decides where to output it and orders the switch to do so. As this might easily overload the controller and does

not the use the switches capabilities to use the flow table at all, this approach is deemed impractical.

Another controller might just install static flows, that serve as "policies", with a high number of wild cards (e.g. "All HTTP traffic goes to server X") and not have traffic sent to it at all. This kind of controller may be suitable for large data centers, where it would be impractical to compute a route through the network for each flow. For smaller networks however, this approach is not necessary and completely ignores some benefits that SDN offers, such as more dynamic routing over different switches, e.g. when the links are overloaded.

Therefore, the controller that is regarded in this thesis, is sent every first packet of a flow, computes the route through the network based on several criteria and then installs flow rules on the switches that it is supposed to pass through.

Unfortunately though, this approach is computationally expensive. The route for each and every new flow has to be recalculated and it is hard to cache the routes due to the regularly changing link metrics such as available data rate, latency and loss rate. This leads to the controller being regularly overloaded, causing delays in the calculation of new routes and delaying packets. This, however, can be mitigated by offloading some of the computationally expensive calculations to a different device, such as an accelerating coprocessor (like the Intel Xeon Phi [37]) or a GPU.

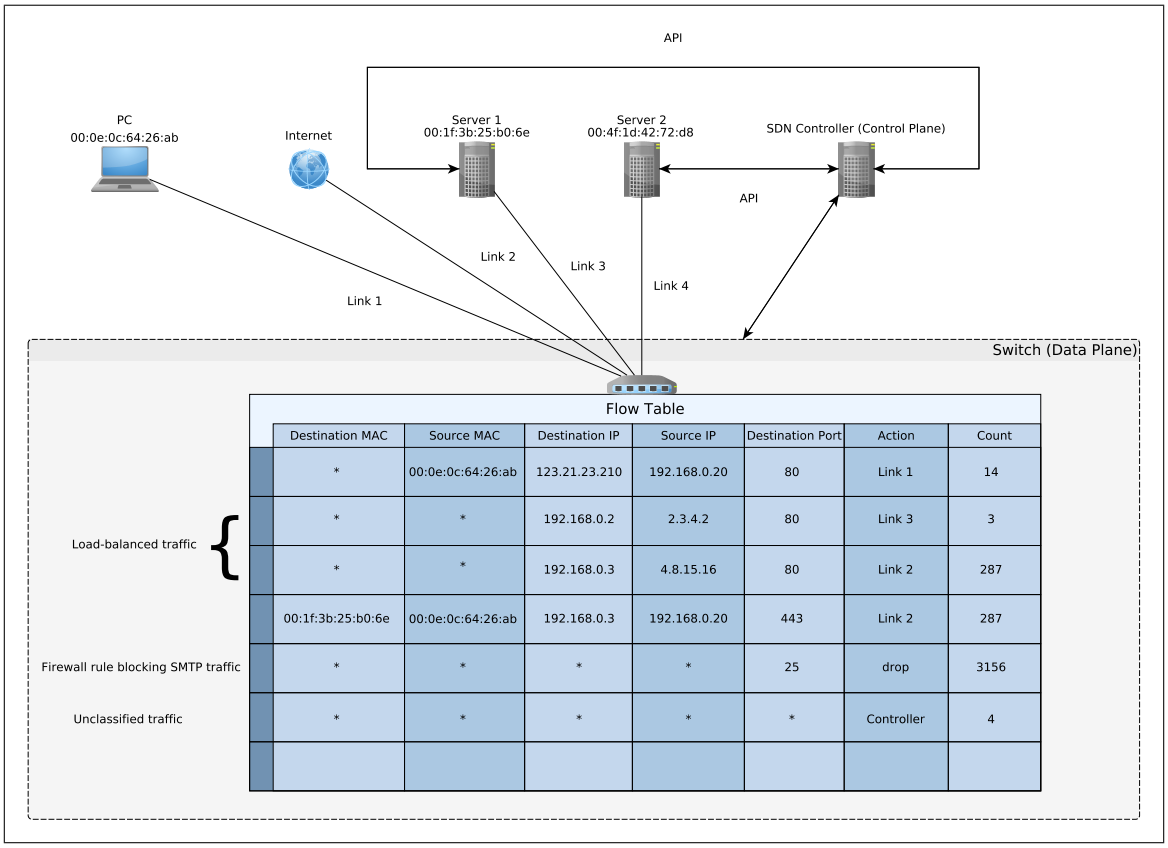

Figure 2.3: SDN switch with flow table.

2.2 Massively parallel computing on GPUs

Most modern desktop computers and servers have at least a simple *graphics card* that is used to paint the content of the display to the screen, be it a console, a more or less simple desktop environment or a graphically highly challenging 3D game. The graphics card is given a specialized part of the computation from the CPU and performs them in a more optimized way than a CPU ever could. This is due to the design of a GPU: It consists of a large number of computation cores, that work in parallel to, basically, put a color on a pixel.

Such an operation is called shading. The small program, that is executed on each of the GPUs compute cores, is therefore called a shader. A shader supports a special set of operations, such as graphics memory reads and writes, integer and floating point operations and loops. The CPU programs usually load object models and textures in the graphics card's memory and then run these shaders, one for each pixel on the screen, in order to determine the final picture. Such shaders can be written in several different programming

language, the most dominant of which are the *High Level Shading Language (HLSL)* [24] for DirectX and the *OpenGL Shading Language (GLSL)* [23] for OpenGL. These languages are, usually at runtime, compiled to GPU machine code and then executed.
In order to shade a modern FullHD 1080p screen, these shaders have to be executed for each of the 2073600 pixels for FullHD resolution several times a second, which is something, that requires a special kind of architecture. Shaders are designed to work without influencing each other while executing. This yields the possibility to execute multiple of them in parallel, which was a major design influence for modern graphics cards. They have up to several thousand compute cores (2048 cores at 1 GHz each for the AMD Radeon HD 7970 [25]) that are specialized on all running the same code on different data. Such an architecture is called *Single Instruction, Multiple Data (SIMD)*, as opposed to normal CPUs, which are mostly *Single Instruction, Single Data (SISD)*[1], especially meaning, that in multi-core CPUs, all cores are able to do their own computations, independent of the other cores.
When the science community realized, that this architecture holds a big potential for a special set of problems, such as simulations, they started to investigate the possibilities of using SIMD architectures for their advantage. *General Purpose GPU Computing (GPGPU)* became a field of research, that started out with "shading" a pixel with the result of a computation: Problems were programmed in pixel shaders and the color of a pixel after the execution would mean a certain result. Such programs were called *kernels* and were then very hard to create and to debug. This soon lead to the creation of other means of developing such programs.

2.2.1 Technologies

Currently, there exist two major technologies for developing GPU-based programs, CUDA and OpenCL. Both are designed around the same principles of GPU programming that exist due to the architecture, but differ in certain points. Both frameworks are shortly introduced before moving on to programming principles on GPUs.

CUDA

In 2007, the graphics cards hardware manufacturer NVIDIA published the first version of their GPU programming framework called *CUDA*[2]. It supports easy development of massively parallel programs on NVIDIA graphics cards. All cards, workstation cards as well as consumer models, since the introduction of CUDA are able to run programs developed with it, but there are specialized models that offer even more computational power and are designed to run CUDA programs.

[1] Special extensions, such as SSE, add a number of specialized SIMD instructions to the regular x86 instruction set [16]

[2] originally Compute Unified Device Architecture, now only CUDA

CUDA offers its programmers a set of programs, such as a compiler, a debugger and even an IDE, that can be used to efficiently develop GPU-based applications. These are written in a language that is a subset of C++, except for some features, such as full recursion support[3] [38] and fully compatible floating point numbers with regards to IEEE 754 [29]. In its current version 6, CUDA even supports transparent host memory access (access to the computer's RAM as opposed to the dedicated graphics memory) and execution on mobile NVIDIA-based devices. In order to ease the pain of transitioning a "normal" C++ program to one with GPU-computing support, there even exists an STL-compatible library called "Thrust" [28].

As a whole, CUDA makes it very easy for beginners in the GPU computing world to start successfully creating GPU-accelerated programs. It takes the work of detecting GPUs, choosing which unit to run on, initializing the device and creating program queues, but also gives the user the possibility to do so if he wants to.

Unfortunately though, being developed by NVIDIA for its own hardware, CUDA could not, and still can not run on graphics hardware of other manufacturers. This caused the need for the development of another, more open language for creating GPU-based programs, called OpenCL.

[3] currently, only recursion up to a fixed level is supported

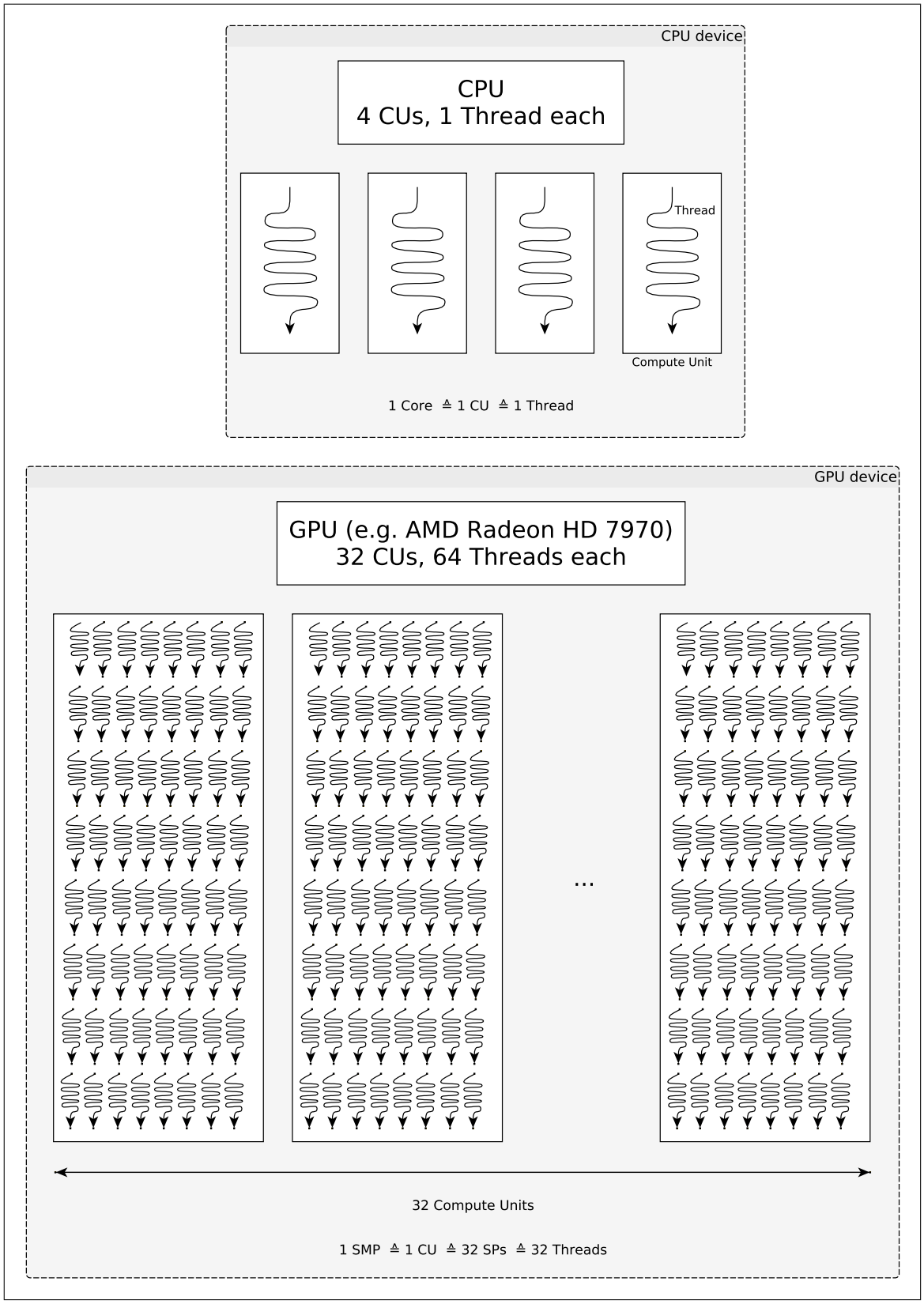

Figure 2.4: SPs and SMPs in a CPU and a GPU

OpenCL

OpenCL is defined and maintained by an industry consortium called the Khronos Group [27]. Being designed to work on everything that has the ability to do computations, OpenCL kernels are able to run on several types of devices, including, of course, GPUs, but also CPUs, mobile devices(OpenCL Embedded Profile) and special accelerators, such as the Intel Xeon Phi [37]. The Khronos Group also defines the OpenGL standard, as well as several others.

OpenCL also offers its own programming language, OpenCL C, which is like C, but incorporates several new syntactic elements specialized for developing GPU kernels. For executing those kernels, OpenCL offers libraries for C, but wrappers for all major programming languages exist.

As OpenCL supports so many different types of compute devices, it is possible to design kernel specifically for a certain type of device. This is usually necessary, as most of the performance can only be harvested by specializing on the specific type of execution architecture (SIMD vs. SISD, see [4]). This means, that kernels designed for CPUs are usually significantly slower when executed on GPUs and vice versa. That fact also becomes clear when looking at the thread model of OpenCL. It distinguished between so-called compute units and threads. Compute units are a number of processor cores that all execute the same code, as opposed to threads, which are executed on each compute core. As can be seen in figure 2.4, in CPUs, the term compute unit corresponds to a thread, as a CPU core is only able to execute one thread independent of others, while GPUs consist of several compute units, which each consists of a certain number of threads itself. On GPUs, a compute unit is a so-called *Stream Multiprocessor*(SM) or in more abstract terms a SIMD unit, meaning that all compute cores in it execute the same code.

2.2.2 Design principles of GPU programming

As previously mentioned, it is important to design programs specifically for execution on GPUs. What is probably most important to notice is, that in usual, CPU-based programs, data is processes linearly, one piece of information after another. This is a good way to handle data in a linear fashion, but if such a program is executed on a GPU, it would be slow[4]. On the GPU, all of this data can be processed in parallel.

For example, suppose all values of an array called "buffer" are to be combined with a commutative and associative function such as summation or XOR. A reasonable way to do this on the CPU might be with the code in listing 2.1.

Here, all data is being linearly added to the first value of the array. This leaves the final value in the first row. This method of adding values can be visualized as in figure 2.5.

[4]This is also why it is important to define separate OpenCL kernels for execution on CPU and GPU

```
for (int i = 1; i < buffer.length; i++) {
        buffer[0] += buffer[i];
}
```

Listing 2.1: Pseudo-Code for reducing of an array with addition on a CPU

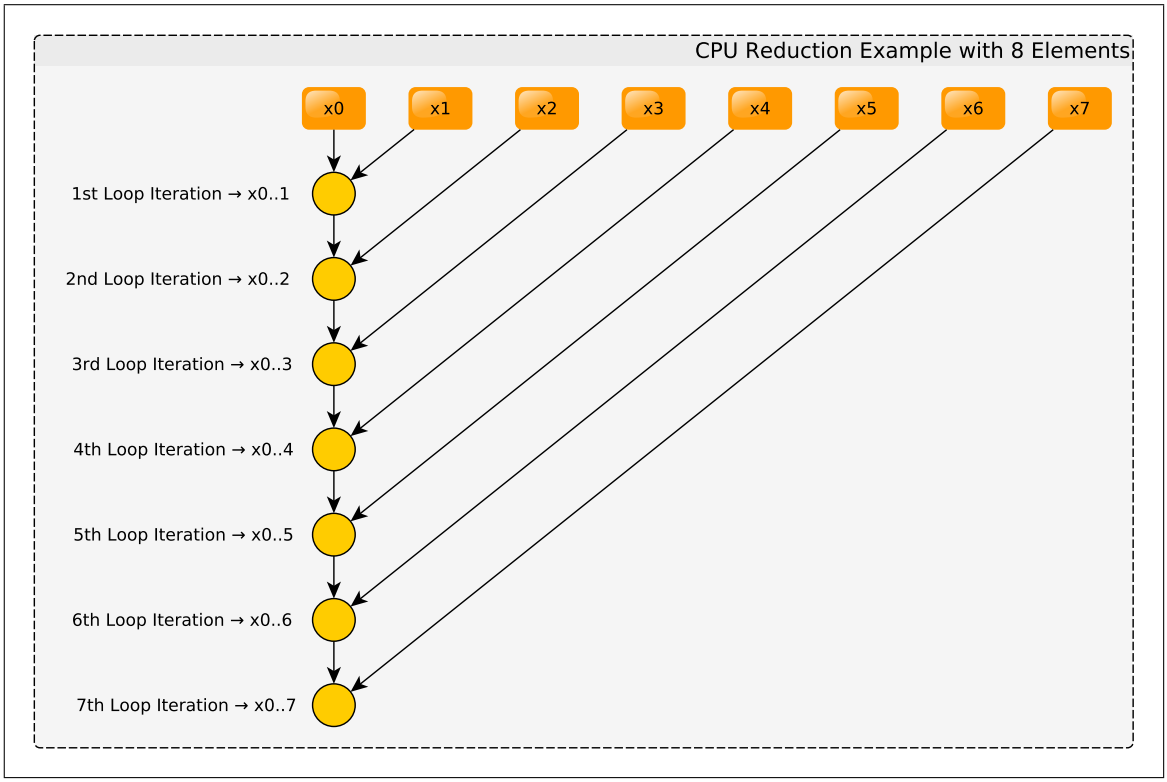

Figure 2.5: Reduction on the GPU

As opposed to that, the GPU version in listing 2.2 should look completely different. In order to stick to the design principle of massively parallel programming, all operations that can be made in parallel should be executed in such a way. In practice, this means first having each core add two values x_i and x_{i+1} and combine them to a temporary sum $x_{i..i+1}$. Then, every core adds two of those temporary sums again to create $x_{i..i+3}$. This process is then repeated until the final value $x_{0..n}$ is computed. A visualization of that process can be found in figure 2.6.

```
for(int offset = buffer.length >> 1; offset > 0; offset >>= 1) {
        if (local_index < offset) {
                int other = buffer[local_index + offset];
                int mine = buffer[local_index];
```

```
                buffer[local_index] = mine + other;
        }
        barrier();
}
```

Listing 2.2: Pseudo-Code for reduction of an array with addition on a GPU. The id of the current thread can be accessed via local_index

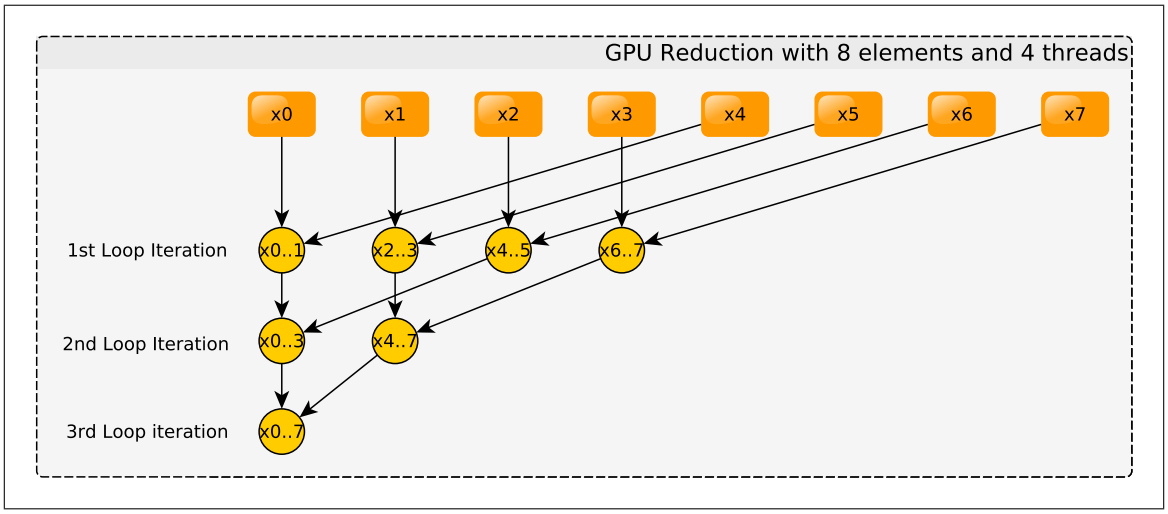

Figure 2.6: GPU reduction

There are several things that can be seen in this simple example, but that also apply to larger kernels:

- While the number of operations is the same in both programs (n-1 additions with n elements), the GPU executes most of them in parallel and therefore needs fewer loop iterations.
- For a small set of items, the CPU version may yet be faster to execute due to two reasons: First, in order to process data on the GPU, this data has to be copied to the GPU and the GPU has to be initialized in the OpenCL / CUDA context. This overhead becomes most dominant when large amounts of data have to be copied to the GPU via the PCIe bus, which is comparatively slow. Second, GPU cores are usually clocked significantly lower than CPU core and therefore execute fewer instructions than CPUs at the same time. Those factors have to be taken into account when thinking about shifting work off to the GPU.
- Executing the CPU version on the GPU would result in only one compute core actually doing work, while the other cores would stay idle. This results in very bad

performance, when compared to executing the fitting version on the corresponding device.

- Executing the GPU version on a CPU would, in turn, result in a large amount of threads being spawned and having to be scheduled, each with its own context switch, causing cache misses in the CPU, comparatively slow main memory accesses and so on.

This all shows, that the way a problem is tackled on the GPU differs significantly from the way it is done on the CPU. It is important to note, that not all problems have a structure that can be parallelized in a meaningful matter. Examples for such problems are those that require disk or network I/O, which can not be done from the GPU at all, but also too much dependency on previous results of an algorithm may make a problem unsuited for execution in massive parallelism. Also, there must be multiple pieces of data, on each of which a thread can run.

Chapter 3

Components of SDN controllers

An SDN controller's logic may be as simple as just acting like a network hub, broadcasting every incoming packet to every port or simply dropping everything. Usually, however, controllers are more complicated than that. This makes it necessary to have a good design for the software. This is why all big SDN controller frameworks use a modular design, that enables the administrator to write new components for his special design needs, e.g. interact with server software to implement load balancing. This flexible design allows for several different types of controllers as mentioned in section 2.1.1.

This chapter gives some examples for components that may be found in a typical SDN controller installation. It also discusses, which of these are well suited for GPU acceleration and where exactly it can be used. As an example, operations that depend on network I/O can not be outsourced to the graphics card, because it has no access to the networking device. Therefore, not all components have the potential for GPU acceleration.

3.1 Network Topology Monitoring

In order to be able to make decisions about the state of the network, the controller has to know, what the network topology is. Usually, this is done by sending LLDP packets out to every port of a switch. If another switch is connected to that port, the packet is sent back to the controller, which can then deduce that there must be a link between these switches.

This way, the controller is able to build a topology graph, containing the switches as nodes and the connections between them as vertices. Other modules can then access this graph in order to e.g. calculate routes or find out which links exist.

As this component must rely on network operations in order to have the switches send the discovery packets between the each other and to be sent the packet in events by the receiving switches, it is unsuitable for GPU acceleration. Additionally, there are no computation intensive operations that can be accelerated.

However, it is necessary to store this graph structure somewhere in order to enable other components to access it. Depending on the architecture of the controller framework, a separate component has to be used for that.

3.2 Datastore

The datastore is a central component, that all other components interface with to save their data. While it is possible to have each modules save its data as local variables in the component, having one centralized unit yields several benefits: Amongst others, it is possible to have different kinds of storage backends, e.g. simply local variables, but also SQL databases, NoSQL databases or saving the data on the graphics card. When using a database as backend, it might synchronize several instances of the controller to create a controller cluster that eliminates the single controller point of failure.
As previously mentioned, it is possible to store the data on the graphics card, which is then used to process this data. It can easily be processed on the GPU, handling up to several thousand data items at once. There are specialized algorithms for filtering data, searching for items and for other group operations that have to be executed on all items, some of which are introduced in sections 2.2.2 and 4.4. This makes the datastore an ideal candidate for GPU processing.
Another benefit for that is, that this approach makes processing data on the GPU easier, as copying data on the device is significantly faster than copying it from host to device[1].

3.3 Client Discovery

To enable the controller to track clients in the network (Non-switching hardware, i.e. end-user devices and servers connected to the network), it has to keep track of the link status of switches, inspect incoming packets and possibly pro-actively send *ping* packets to determine the status of the clients. Another indicator for the liveness of a client may be flows with a weak timeout (Timeout after x seconds if no packet is sent via this flow).
Clients are however, unlike switches, not entered into the connection graph because they would only increase the complexity of it significantly. This is something that is to be prevented when creating such a graph because algorithms that operate on it become slower without benefiting from the client information.
Arguably, this module works in a similar fashion as the topology module, but the differences in tracking which clients are active and in the lack of implications of the connection graph lead to this being considered as a separate module. However, this does not change the unsuitability for GPU acceleration as it also depends heavily on network operations.

[1] 264GB/s on the AMD Radeon HD 7970 as opposed to copying data via PCIe 3.0 16x [25], which is only 15.75GB/s [17]

3.4 Unicast Routing

The routing module is central to the process of delivering packets from one edge of the network to another. Whether this is done by setting "static" flows, in which the routes are precomputed and no packets are sent to the controller, or by dynamically computing the routes, the controller has to have some means of calculating the shortest path between two points in the network.

This is done by using some shortest path algorithm, such as *Dijkstra's algorithm* [39], on the network topology graph. Note, that the cost function that is needed for such algorithms has to be defined in order for the algorithm to work meaningfully. If one were to use a constant cost function, then the deterministic algorithm would always choose the same route across the network, so a better cost function has to be found. For simple load balancing, the utilization of the link may be used, so the link with more capacity is preferred over the link with less capacity. Another possibility is regarding the *Quality of Service* QoS tag of a packet [18] and using link properties as the cost function.

As an example, assume being connected to the Internet via both a high latency but high data rate satellite link, as well as with a low latency but low data rate ISDN connection. The routing algorithm should then decide to send latency-sensitive voice over IP packets via the ISDN connection and traffic with a large amount of transferred data, such as FTP, via the satellite link.

Such considerations have to be taken into account when designing a controller. Fortunately though, shortest path algorithms are well-suited for GPU acceleration and although not everything about routing, especially installing flows, can be offloaded to the GPU, the computationally expensive shortest path algorithms can.

3.5 Multicast and Broadcast Routing

Multicast and broadcast routing in a controller can be implemented by flooding out packets to all clients, thus making this module optional. But as this method is not optimal, it should be part of an SDN controller.

First regarding broadcasting, this problem might be modeled as a minimum spanning tree [13] across all switches, which then in turn output the packet to each client. However, as an SDN controller might be handling a larger network with multiple subnets or as some switches might not have clients connected to them at all, the problem should rather be regarded like multicasting packets.

Multicast routing can be modeled as *general Steiner Tree problem* [5], which creates a minimum spanning over a weighted subset of nodes in a network. The algorithms that solve this problem though, are only heuristics [40] and of those, none have been ported to the GPU as of the writing of this thesis.

As a workaround, an approximation of the tree can be built using shortest path algorithms, but this is not part of the controller built for this thesis and is therefore not discussed here. With this approximation though, the algorithm can partly be executed on the GPU.

3.6 Upper level APIs

One of the design goals of Software Defined Networking, was to expose some of the functionality to the upper levels of the network. For example, an HTTP server could report to the SDN controller, that its load is currently high, to which the controller could respond by routing new traffic to a different server.
This so-called *northbound API* is optional, but some controller frameworks already provide it, e.g. as a REST API [41]
One way to implement such changes is by modifying the cost function of links or by having special support for such load-balancing features in the code. Depending on exactly how it is implemented, GPU support may be available, but it is not a feature that this controller supports.

3.7 Routing

As of the time of this writing, most traffic on the Internet still uses IPv4 [43]. As there is a global lack for public IPv4 addresses [42], the concept of private address spaces and *Network Address Translation* (NAT) [19] was introduced and is used in most private networks [6], such as consumer home networks or even for companies or universities.
Without going into detail, it can be said, that these operations can easily be done by an SDN controller as well, without needing specialized hardware. A switch and one port on this switch can simply be defined as a border switch / port, and then the controller may route traffic to this switch and rewrite it accordingly on the switch, before forwarding it to the next network.
Unfortunately, these operations are hardly worth being done on the GPU as they involve mostly network operations.

3.8 ARP and rARP Responder, DHCP and DNS server

Having global knowledge over the network, a controller may act as a responder to ARP and rARP queries. The necessary information is already in the information, that the controller may collect about the network, so when detecting such a packet, it might respond to this packet.
Respectively, in a network environment that uses DHCP for IP address assignment, the controller may choose to manage the address pool by itself, collecting host name information as a byproduct of the DHCP protocol. It can then use this information to respond to DNS

queries or choose to forward them to external DNS servers. Being a central entity, it may even cache this information for the entire network, thus reducing the traffic to external networks such as the Internet.
These types of queries can easily be translated into queries in a centralized datastore and can therefore be GPU accelerated together with the datastore.

3.9 Batch processing of packets

At this point, it should be noted that these components describe a rather sophisticated SDN controller that relies heavily on the global knowledge it has about the network.
As was previously mentioned though, it is possible to operate an SDN controller with only little more functionality than that of a conventional network switch. When doing so, the controller keeps a source address table for each switch (functionality explained in section 2.1) and forwards packets accordingly, as depicted in figure 3.1.
Such a controller may be efficiently optimized for GPU acceleration by collecting packets over a certain amount of time or number of packets and then looking up the next hop on the GPU for each batched packet. Afterwards, the flows are installed on the switches.
This method was already covered in a related work by R. Yanggratoke [10]. This implementation though, uses route calculation over the network as a whole, and therefore optimizing packet processing in such a way is not feasible here.

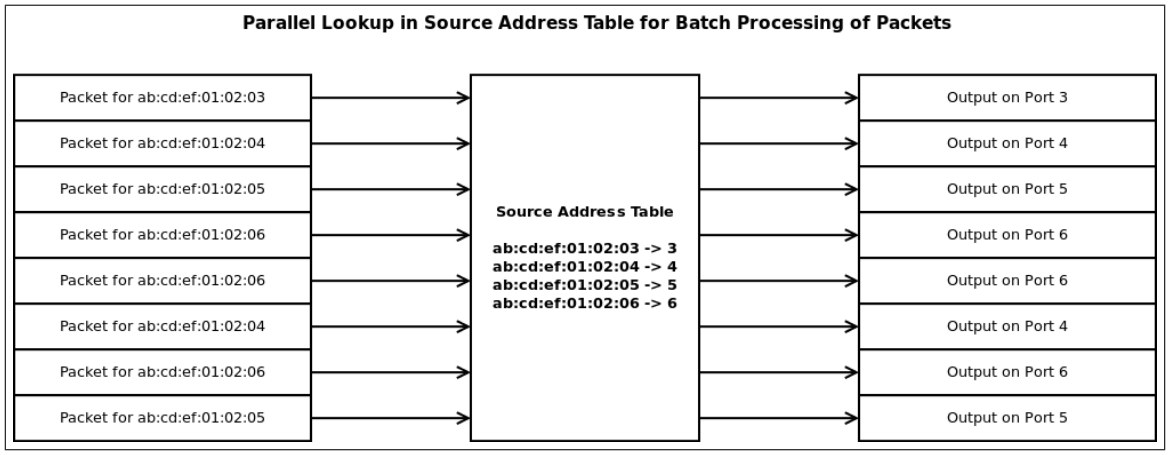

Figure 3.1: Batch processing of packets by parallel SAT lookup

Chapter 4

Bonfire - A GPU accelerated SDN controller

As a concrete implementation of a GPU accelerated SDN controller, the *Bonfire controller*, that was built for this thesis, incorporates massively parallel computation as a solution for several problems that occur in Software Defined Networking controllers. This chapter describes the design of this controller, explains the GPU-based algorithms used in the components and discusses the problems that occurred during the implementation.

4.1 Overview

The central component in Bonfire is the *datastore*. Most components are connected to it in order to store its data and receive it afterwards. It also has an event-based API for which other components can register to e.g. extend or invalidate caches.

The *network topology* and *client discovery* modules keep track of existing and newly discovered switches and clients and enter this data into the datastore.

Using this data, the *routing* component has multiple unicast algorithms[1] at its disposal, in order to compute and set a route for a flow through the network. These algorithms are, in the interest of modularity, also separate components.

Additionally, in order to have a centralized management component for the OpenCL compute resources, the *compute device manager* offers the functionality to initialize the compute devices, compile and manage kernels and dispose of these resources again after the execution.

[1] In the future, multicast algorithms may be implemented and easily integrated

4.2 Beacon Controller Framework

The *Beacon Controller*, created by David Erickson at Stanford [44], is a complete OpenFlow controller framework. In particular, Beacon is already a fully runnable controller and able to perform on its own, but it also allows the users to write their own plugins, or even rewrite most of the code, as was done for this thesis.

The framework itself uses the Java OpenFlow library *OpenFlowJ* [45]. OpenFlowJ is a library that helps applications connect to OpenFlow devices by offering classes for wrapping the structs that are defined in the OpenFlow specification.

Beacon then uses these classes to communicate with the switches in the network and builds another wrapping layer that, in its core functionality, offers these channels to the other modules and by keeping track of the switches. It then offers an event-based API on which other components can register. They are then called when the framework receives something. Beacon also offers the functionality to define a priority ordering in which the external modules are called.

Bonfire uses only the most basic parts of the Beacon Controller framework because the preexisting structure saved the state of the network in local variables in the corresponding modules. As this thesis has the goal to unify and accelerate data storage on the graphics card, Bonfire does not use those components. It does, however, register its components for all events it is concerned with, e.g. switches being added or removed or packet in events at the switches.

Whenever data is received by the Beacon framework, it wraps it into Java object with the help of OpenFlowJ and then analyzes its content. In case it is an event[2], it notifies all components that registered for it in the predefined order. However, a component may declare the event *handled* or *consumed*, which will stop the Beacon framework from passing it to registered event recipients. This process is visualized in figure 4.1.

[2] As opposed to e.g. a ping or a features reply

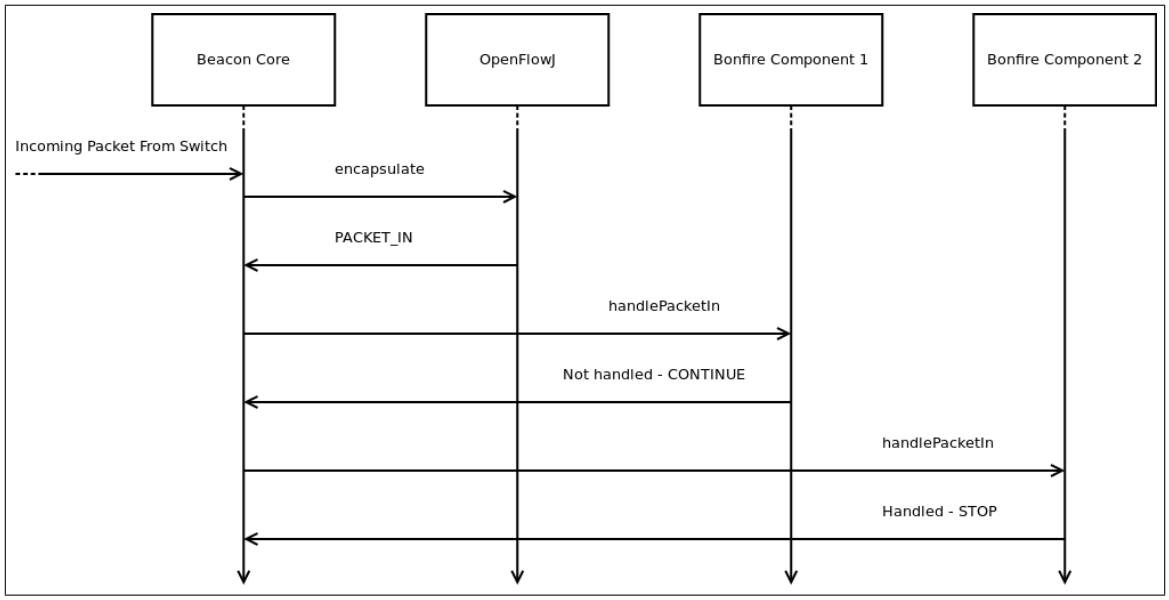

Figure 4.1: Role of Beacon Framework Core in Bonfire

4.3 Network Topology and Client Discovery Modules

In order to be able to create a network graph later, Bonfire has to keep track of all switches and clients in its network. For this, it has two modules called the topology module and the client discovery module, that both work in a similar fashion: Both listen to packets from their respective subjects and keeps a record of them in the datastore.
The topology module does so by registering with Beacon for switch added and removed events, as well as by keeping track of the times they last communicated with a switch. When the time difference between this last communication and the current time becomes too large, it interprets the switch as disconnected and removes it from the datastore. This timestamp is reset every time that a switch answers an OpenFlow ping packet that is automatically sent by the Beacon framework.
In a similar way, the client discovery module also keeps track of all clients by regularly sending ARP [20] packets to them as a means of pinging them. This allows the module to meaningfully declare a client as timed out, which is necessary, as clients are not obligated to sign off with the controller.
In order to achieve the timeout functionality, every object that represents a switch or a client has a field that contains the last modification time. This time is updated whenever a packet from that particular network entity is received. A separate thread then runs every few seconds in the corresponding components and checks every item for a timeout.
Additionally, the topology module also sends LLDP [21] packets to all switches on all ports. Clients will simply ignore these packets, while other OpenFlow switches will send

these packets as a packet in event back to the controller. By putting a special signature in the optional TLV field (the payload) of the LLDP packet, the Bonfire controller can then determine, from which switch the packet originated and build a link between these two switches. At the same time, these LLDP packets also serve as keep-alive packets for the links and are therefore also used for recognizing link failures.
It should also be noted here, that, due to the special design of the used OpenFlow testbed, switches that are not part of the OpenFlow network, also send LLDP and other packets such as loop detection [13] packets as a means of detecting other network components and the topology. These packets can, however, be recognized and are ignored in the Bonfire controller.

4.4 Datastore Module

As the datastore is the central component of the controller that does most of the communication between the modules, designing it is challenging. There are several requirements the datastore has fulfill and that have to be kept in mind when creating such a design:

- Offer interface for storing data of all existing modules
- Easily extensible for new modules
- Accesses synchronized for parallel execution of modules
- Must support different means of saving the actual data
- Keep data consistent

As it is hardly possible in a well designed program to satisfy both general extensibility and specialized consistency rules, the decision to split the datastore component into several components, that fulfill certain interfaces, was made.
The datastore in the Bonfire controller has one back-end and can have one or more front-ends. In this version of the program, there is one front-end that serves all internal components of the controller and takes care of data consistency.
When designing additional components in the future, other users may choose to use and possibly extend this front-end for their needs. Another possibility would be writing an additional front-end to keep the possibility to receive updates for the already existing components.
The back-end takes care of actually saving the data in a generalized way. The interface resembles the API of a database, trying to be as unspecific as possible. It has several tables, each entry of which has multiple columns (attributes) and is represented by a Java object.
In order to access the data, the back-end interface offers two functions: The first function supports queries in disjunctive normal form (DNF). An example for such a query might be

"Give me all client for which the IP is in the 10.2.1.0/24 subnet, as well as all clients that have not been active for 60 seconds and that were created more than 120 seconds ago". The DNF representation of this query would then be as follows:
$(\text{IPv4Address} \geq 10.2.1.0 \bigwedge \text{IPv4Address} \leq 10.2.1.255) \bigvee$
$(\text{LastModTime} < 1401437860000 \bigwedge \text{CreationTime} < 1401437800000)$[3]

This kind of queries can easily be executed on both the GPU and the CPU, as well as represented in other query languages such as SQL.
The other query type exists specifically for getting a single item on which a key attribute[4] is matched. An example for that is:
$ClientMAC == 12:34:56:78:90:AB$
This query can only yield exactly one or zero results[5] and is easier to execute on both sequential, as well as concurrent versions of the datastore.
In its current version, the Bonfire controller has two back-ends to choose from. The sequential, CPU-based version saves all data as Java objects in arrays (See figure 4.2). These are located in the normal RAM of the machine and are accessed sequentially from within "normal" Java. DNF queries are executed by iterating through the table in a row-wise fashion and keeping track of which of the elements fulfills the query. The query for single objects also iterates through the table, but stops when the queried element is found.

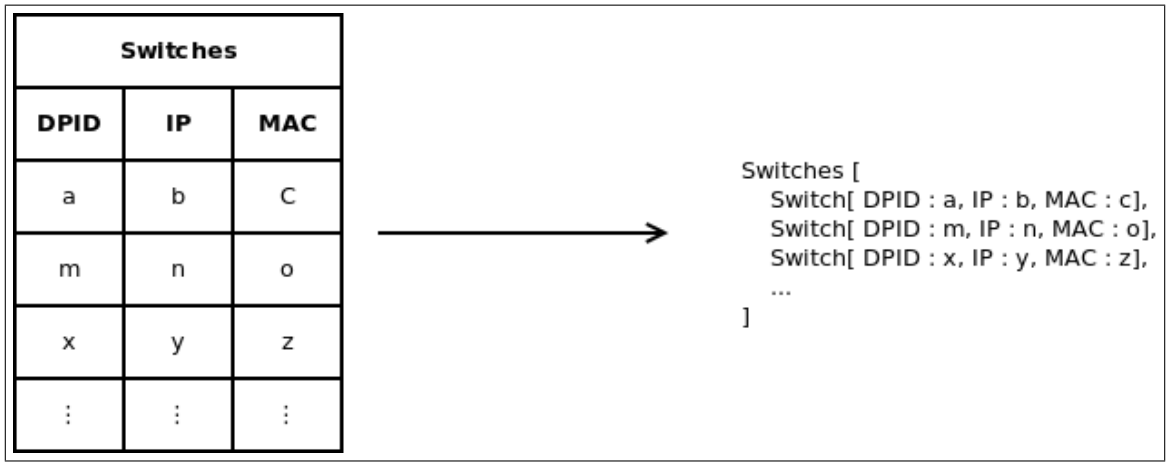

Figure 4.2: Data storage in CPU version of Datastore as row store

The other back-end is graphics card based. All items are completely kept on the GPU and are accessed via the Java OpenCL bindings, that the JOCL library [46] provides. In order

[3]Time is given in milliseconds since January 1st, 1970.
[4]Every key attribute can only occur once in the table
[5]Not considering that clients might fake their MAC addresses for this example

to optimize access times, the data layout is specifically designed to fit the requirements for GPU programming concerning data locality.
On the GPU, it is beneficial to save the data that each thread needs close by, as memory fetches can only ever fetch blocks of data and the more unneeded data is fetched, the more fetch operations are needed. For the concrete implementation, this means that, in order to store multiple tuples of data, each value of a single attribute in a tuple is stored in an array, as is depicted in figure 4.3. Such a database is called a column store, as opposed to the method of saving tuples as one, which is called a row-store [7].

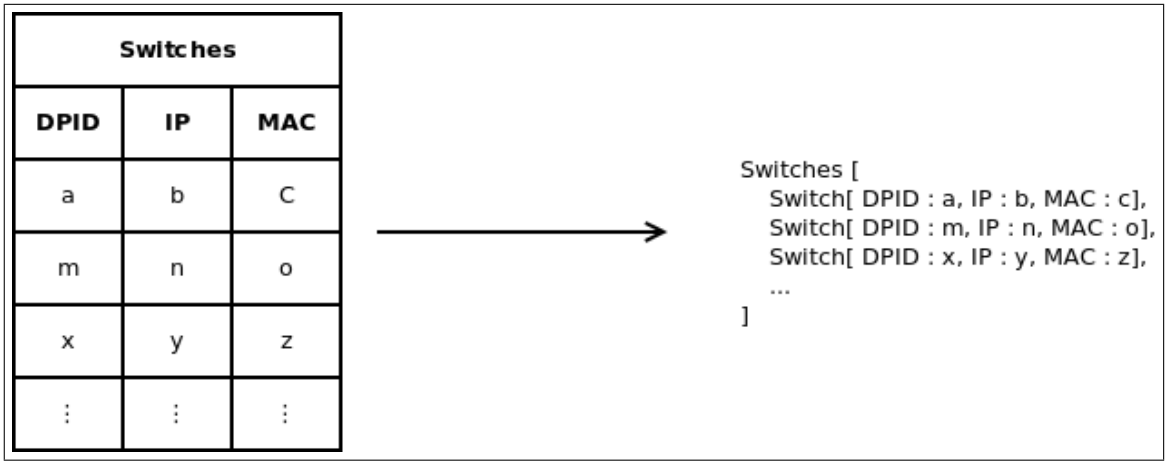

Figure 4.3: Data storage in GPU version of Datastore as column store

Whenever a value is set, the appropriate OpenCL function is called and the value is written to the graphics card. Queries are executed on the graphics card itself and return a number of indeces. These indeces correspond to internal IDs, each of which corresponds in to a tuple of attributes, representing an object. When running a query, the resulting objects are built by creating transient objects that "know" their internal ID. As soon as a value is actually read from this transient object, it is copies off of the graphics card and cached in the object. This has the advantage, that no values that are not needed are read from the graphics card.
How these queries are actually executed on the GPU is described in the following section.

4.4.1 Description of GPU-based filtering

As the values are saved in a column based fashion, queries in DNF have to be split up into their components, which are then worked off one after another, but each component in parallel. This process is visualized in figure 4.4.
In DNF, every disjunctive part is called a clause an every conjunctive component is called a literal. For example, in the DNF query $(a \bigwedge b) \bigvee (a \bigwedge c)$, the clauses are $(a \bigwedge b)$ and $(a \bigwedge c)$

and the literals are a, b, c and d. These literals can be comparisons with the operators $<$, $\leq$, $>$, $\geq$, $=$ and $\neq$. For ease of execution on the GPU, the queries are pre-processed to contain only $<$, $>$ and $=$, as well as a "not" flag (e.g. "$\leq$" becomes "NOT $>$").
Then, every literal is processed on the GPU. At the beginning of processing a clause, an array is created on the graphics card, containing values of 0 or 1, denoting whether a value is still "active". A "one" means, that it is still considered as a candidate for a clause. In the figure, this is the lower array between steps 1 and 2. Note, that initially, this array is only 0 for deleted elements. These are not actually deleted, but rather marked as such.
The GPU then receives the information on the query and executes it on all active entries in parallel, marking entries that do not match as not active. This process is repeated for all literals in a clause, each literal potentially reducing the number of active entries.
The actual code of the filter kernel is depicted in listing 4.1. *buffer* contains the column data has has *length* elements. The same length has the *active* array. *compareRVal* is the value, the query compares against, *compareOp* is an encoded form of the compare operator and *comparePos* denotes the negation value of the literal.
The kernel only executes for active values and writes the result of the comparison back to the active value. This process is repeated for every literal in the clause.

```
__kernel void filterInt(
 __global int* buffer,
 __global int* active,
 __const int length,
 __const int compareRVal,
 __const char compareOp,
 __const char comparePos) {

 int gid = get_global_id(0);
 if (gid < length) {
  if (active[gid]) {
   char result;
   if (compareOp == 0) {
    result = buffer[gid] == compareRVal;
   } else if (compareOp == 1) {
    result = buffer[gid] < compareRVal;
   } else if (compareOp == 2) {
    result = buffer[gid] > compareRVal;
   }
   active[gid] = result == comparePos;
  }
 }
}
```

Listing 4.1: Code of filter kernel for integer values

After processing the clause, the array with active values is "scanned", which means that the prefix sum of it is created. In a prefix sum, every field contains the sum of all previous

fields. This effectively yields the number of entries in the final array, as well as the position of each active index in the final array. This is depicted in steps 2 and 3 of the figure.

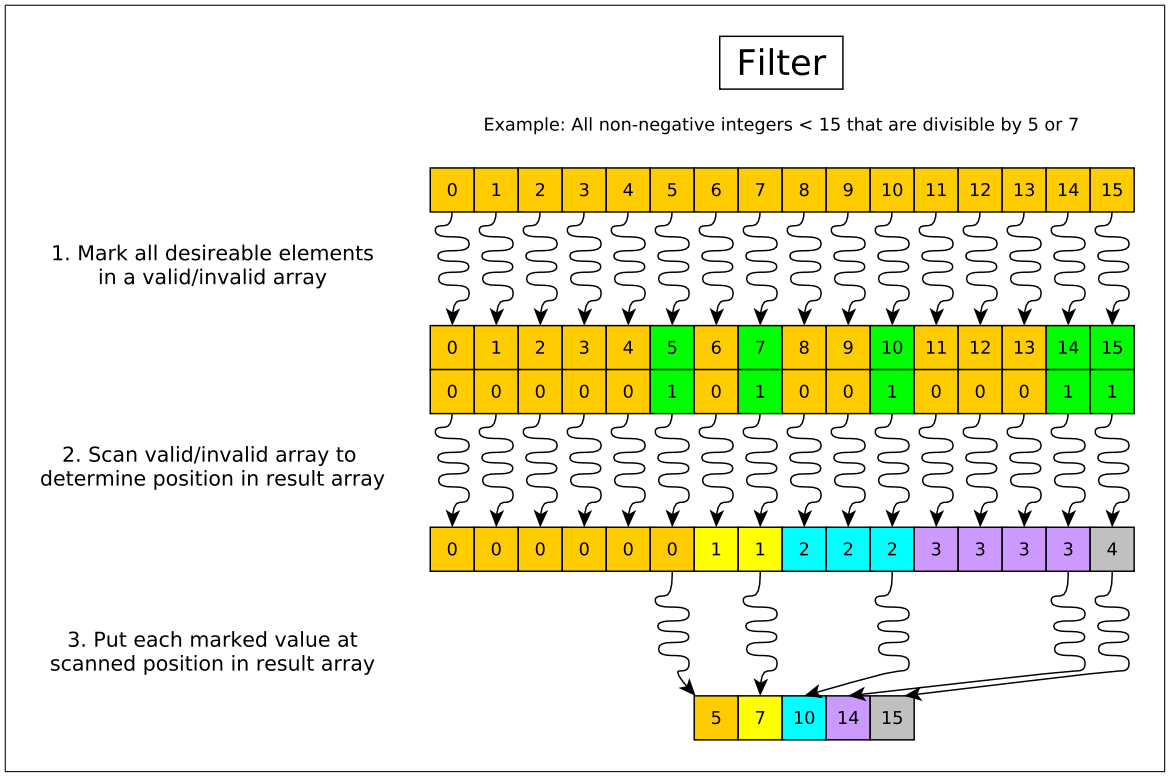

Figure 4.4: Filtering data on a GPU

While, for completely processing a DNF query, it would now be necessary to also process the other clauses, but it turns out, that for the Bonfire controller as it currently is, all DNF queries only comprise of one clause. However, the procedure is sketched out here:

For effective DNF query processing, the former algorithm has to be used until one clause is completely processes with the filtering kernel. Then, instead of scanning the active array, it is kept as the result of the clause. New active arrays are then created for each clause and every clause is then executed on each of these arrays. In the end, there are n active arrays for n clauses. Then, the GPU has to create the disjunction of each of these arrays and write them to a final active array. The value $active_{final}[i]$ is then $active_0[i] \bigvee active_1[i] \bigvee ... \bigvee active_n[i]$ for every entry i.

This final active array is the scanned and output as previously described.

4.4.2 Description of GPU-based Singleton Search by Key Element

When searching for an element that is known to only occur at most once, the whole process becomes dramatically less computationally expensive. It then suffices to keep a single variable for the result, that will only be written to when the value is found. This completely eliminates any data races[6] that this method may lead to when having more than one result. Note, that this also removes the need for scanning the array and copying over the results afterwards, as the internal id of the object is immediately returned.
The code for the kernel is depicted in listing 4.2. The *items* array contains all values in the data column and is of length *length*. The reference value that every value is compared to, is saved in *ref* and the return value is saved in *returnValue*.

```
__kernel void containsInt(__global int* items,
 __const int length,
 __const int ref,
 __global int* returnValue) {

 int gid = get_global_id(0);
 if (gid == 0) {
  returnValue[0] = -1;
 }
 if (gid < length) {
  if (items[gid] == ref) {
   returnValue[0] = gid;
  }
 }
}
```

Listing 4.2: Code of "contains" kernel

4.5 Routing Module

The routing module receives all packets that are meant for routing. It is prepared to support multicast and broadcast routing, and it already fully supports unicast routing and controller broadcasting, i.e. the controller sends packet out commands to all switches to flood the packet to all non-switch ports.
It also defines several interfaces for single-source shortest path algorithms, all-sources shortest path algorithms and multicast-tree algorithms, making the implementation of different algorithms for the same purpose easily possible.
When receiving a packet for unicast routing, it currently uses a single-source shortest path algorithms to compute a path through the network for this flow, installs it on the intermediate switches and sends the packet on its way, as depicted in figure 4.5.

[6] A data race occurs when two or more threads try to write to a value at the same time. Afterwards it is not clear, which value was actually written in the variable

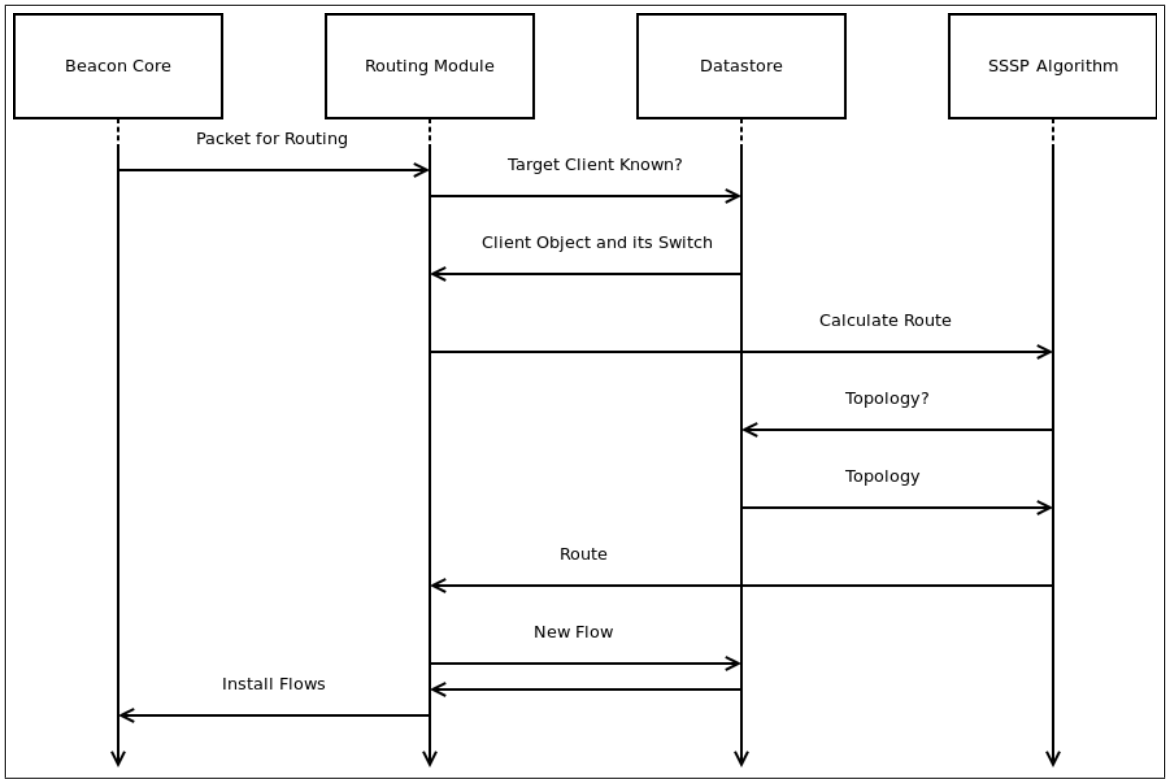

Figure 4.5: Routing Module with packet for Routing

To take some of the computations from the device, it buffers the calculated routes for a short time, which is particularly used when the request in the routed packet has to be answered. With flows only working unidirectional, it is possible to cache the reversed way. Another possible way of action to accommodate that behavior would be proactively installing a flow for the reversed direction, but as some streams only work unidirectionally, e.g. some multimedia streams, this is not done. Caching the routes also has the advantage, that other flows between the two machines can easily be installed this way.
In this version of the Bonfire controller, there exist two single-source shortest path algorithms (SSSP), both implement Dijkstra's algorithm, but one does so on the CPU with Java objects, the other operates on the GPU and is thus modified for optimized runs on the graphics card.

4.5.1 Description of GPU-based Shortest Path Search

A detailed description of a sequential version of Dijkstra's algorithm can be found in [39], the Java version for the algorithm is a slightly modified version of the algorithm by Lars Vogel [47].

The basis for the algorithm is a graph, which, in our case, is comprised of the switches and links between these switches. It has to be directed and weighted. As network links are always bidirectional, the network is actually undirected, and therefore has to be converted to a directed on by replacing each undirected edge with two directed edges, one in each direction. Also, the graph obviously has to be connected, i.e. there has to be a path between any two vertices.
The algorithm basically looks for each node, if the cost of going from the currently regarded node to every neighboring node plus the cost of going to the current node is greater than the cost of going to the neighboring node at that moment. This step is called "relaxing" the edges and is repeated several times, until nothing changes any more. The result is a mapping of edges to costs, as the algorithm actually computes the minimum cost of all other nodes to a single source.
In its original form, the algorithm only computes the cost, but it can be easily extends to also calculate the route that has this particular cost by also noting which edge was taken in order for this cost to be used.
For the GPU version, all relaxation steps are executed in parallel. The basis for this version is by Harish et al. [8], who wrote it for CUDA, the adaptation for OpenCL is by Ginsburg et al. for the book "OpenCL Programming Guide" [9]. Unfortunately though, this algorithm has some problems, which is why a heavily modified version of the algorithm was created for this thesis and is used in the GPU version of the SSSP algorithm for the routing component. Details for that can be found in section 4.8.3

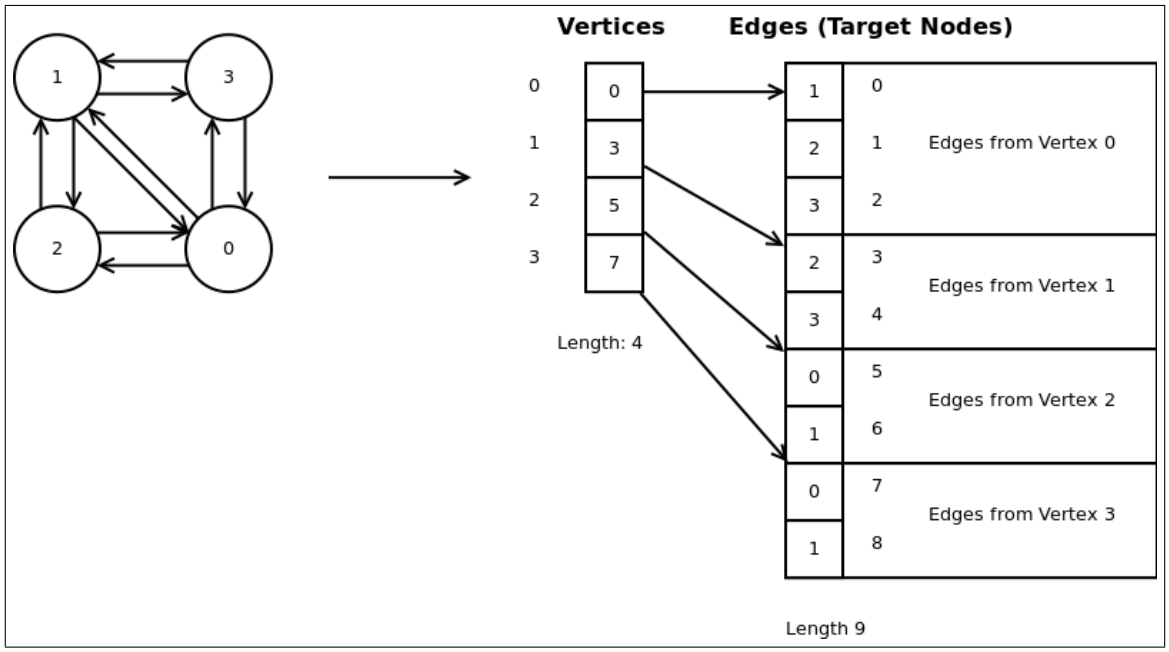

Figure 4.6: Graph structure for GPU-accelerated SSSP

The actual graph structure is contained in two arrays, one containing the vertices, the other containing the edges. The edges array contains all edges for one vertex as a cluster, represented by indices in the other array. The vertices array contains pointers into the edges array. An example for this structure can be seen in figure 4.6.
The three kernels are executed with as many threads as there are vertices in the graph, so each function is executed "for each vertex".

1. **Initialize variables**: This kernel initializes the path and cost variables on the graphics card. See listing 4.3
2. **Phase 1**: Relaxes the edges. It works by having each thread iterate through its outgoing edges and comparing the current cost of the targeted node to that of the current node plus the cost of the edge. If the relaxed cost is less than the current cost, the edge that causes the relaxation is saved for this particular node. This step might cause a data race which is corrected in a later iteration. See listing 4.4
3. **Phase 2**: This kernel checks, if the current node is to be relaxed by an edge. That edge would then be recorded and the new cost for reaching this node is recorded. It also keeps track of whether the algorithm is finished. This is the case if no relaxation took place in a particular iteration. See listing 4.5

The kernel that initializes the variables is executed once before everything else. The two phase kernels are repeated as long as there have been changes, which is evaluated by the second phase.
After executing the kernels, the cost array contains the cost to get to each node from the source, and the *lastTakenEdgeArray* contains the index of the edge that must be taken from the previous node in the path to get to this node. This actually describes a tree from the source node to each other node. In order to get the path to the target node, the algorithm takes advantage of the fact that each node has a corresponding link object, the internal id of which is taken from an additional array. This way, it can construct an array of link objects that is then returned to the routing algorithm.

```
__kernel void initializeBuffers(__global float *costArray,
 __global int *lastTakenEdgeArray,
 int sourceVertex,
 int vertexCount ) {

 // access thread id
 int tid = get_global_id(0);
 if (tid < vertexCount) {
  if (sourceVertex == tid) {
   costArray[tid] = 0.0;
   lastTakenEdgeArray[tid] = -1;
  }     else {
   costArray[tid] = FLT_MAX;
```

```
   lastTakenEdgeArray[tid] = -1;
  }
 }
}
```

Listing 4.3: Code of Dijkstra GPU variable initialization kernel

```
__kernel  void OCL_SSSP_KERNEL1(__global int *vertexArray,
 __global int *edgeArray,
 __global float *weightArray,
 __global float *costArray,
 __global int *updatingEdgeArray,
 __global int *lastTakenEdgeArray,
 int vertexCount,
 int edgeCount ) {

 // access thread id
 int tid = get_global_id(0);
 if ( tid < vertexCount ) {
  updatingEdgeArray[tid] = -1;
  int edgeStart = vertexArray[tid];
  int edgeEnd;
  if (tid + 1 < (vertexCount)) {
   edgeEnd = vertexArray[tid + 1];
  } else {
   edgeEnd = edgeCount;
  }

  for(int edge = edgeStart; edge < edgeEnd; edge++) {
   int nid = edgeArray[edge];
   if (updatingEdgeArray[nid] == -1 && costArray[nid] >
       (costArray[tid] + weightArray[edge])) {
    updatingEdgeArray[nid] = edge;
   }
  }
 }
}
```

Listing 4.4: Code of Dijkstra phase 1 kernel

```
__kernel  void OCL_SSSP_KERNEL2(__global int *reversedEdgeArray,
 __global float *weightArray,
 __global float *costArray,
 __global int *updatingEdgeArray,
 __global int *lastTakenEdgeArray,
 __global int *maskExec,
 int vertexCount) {

 int tid = get_global_id(0);
 if (tid == 0) {
```

```
  maskExec[0] = 0;
 }

 if (tid < vertexCount) {
  int updatingEdge = updatingEdgeArray[tid];
  if (updatingEdge != -1) {
   costArray[tid] = costArray[reversedEdgeArray[updatingEdge]]
                              + weightArray[updatingEdge];
   lastTakenEdgeArray[tid] = updatingEdge;
   // Loop as long as something changes.
   maskExec[0] = 1;
  }
 }
}
```

Listing 4.5: Code of Dijkstra phase 2 kernel

A detailed example for the execution steps of the algorithm can be seen in figure 4.7.

In the graph in the top left corner, red are the node costs and black numbers are IDs. The green arrow points to the source node. Its cost is initialized as zero, the cost of all other nodes are initialized as positive infinity.

Then, the algorithm is executed just like the normal SSSP by Dijkstra with the notable exception, that all nodes are considered at once. In step 2.1, this leads to the previously mentioned data race. In this version however, step 3.1 corrects this data race.

At the end, the cost and the required edges to get to any target node are stored in the table and can be copied back to the host.

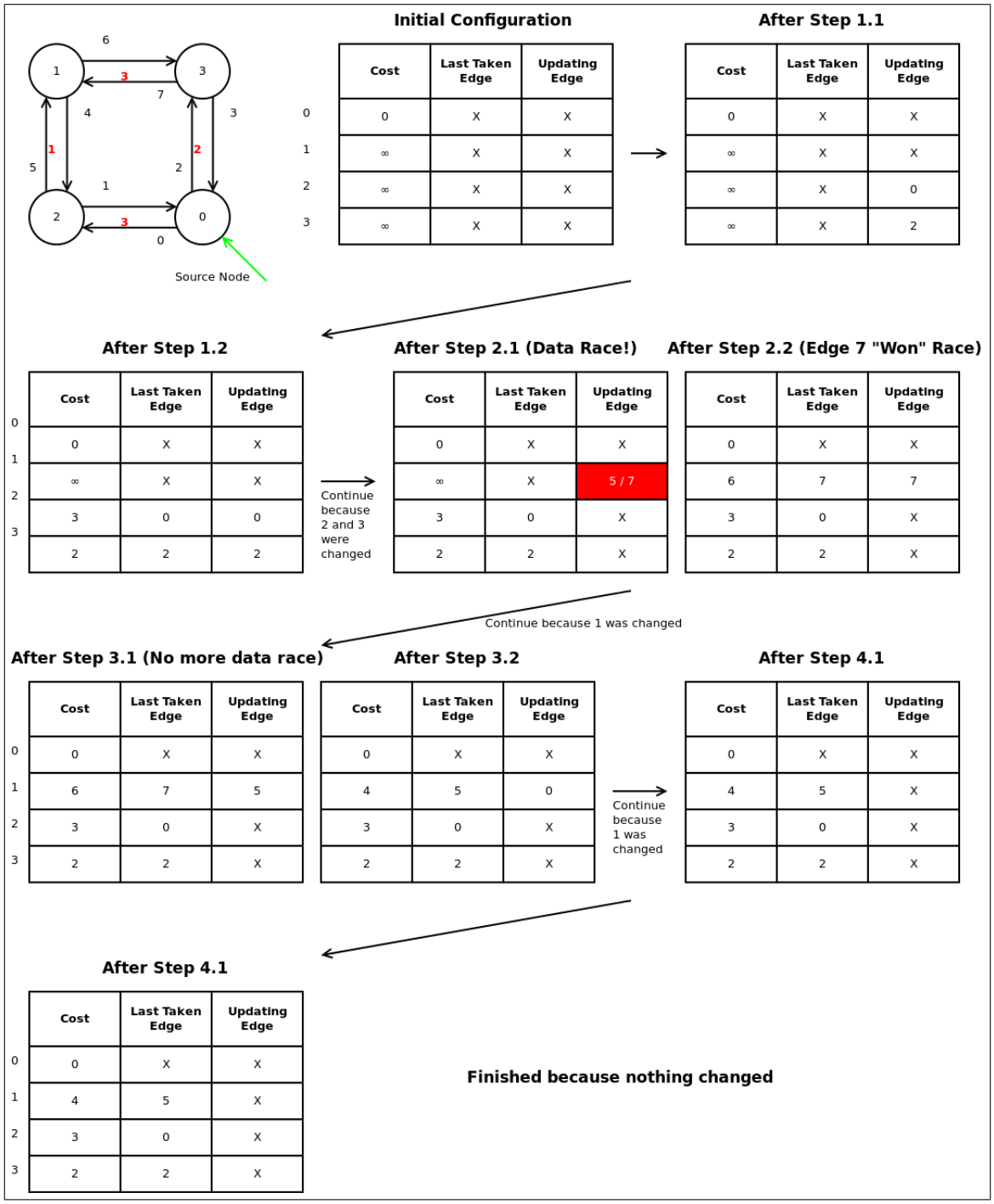

Figure 4.7: Dijkstra's algorithm on the GPU

4.6 Compute Device Manager

In order to be able to keep track of the resources allocated for the OpenCL algorithms, the Bonfire controller has a component that manages them. It is called the *Compute Device Manager* and offers an interface that allows other components to get access to the current OpenCL context and the command queue, which they can then use to create GPU buffers and enqueue operations such as kernel executions or memory operations.
It also unifies the code used for compiling kernels, which is done before running the kernel, but after running Bonfire. This approach was chosen due to the fact that, while it increases start-up time by a few seconds, it makes the controller more portable. When pre-compiling kernels, they would be compiled against a specific architecture.
Another advantage of the approach is, that methods for prioritizing kernels and for assigning them to devices can easily be added as an interface method, e.g. for assigning a lower priority to the threads that check for timeouts.

4.7 Example: Incoming Packet for Unicast Routing

In order to show how all components work together, an example is portrayed in figure 4.8, outlining only the communication between the components. It shows an incoming PACKET_IN message, an event that denotes an incoming packet at a switch, for unicast routing between two known clients and is described in this section.
First, the Bonfire controller receives a packet from a connected SDN switch. Beacon core handles all switch connections, so it receives the data first. In order to gain more information about the packet, it calls OpenFlowJ to encapsulate the packet.
When it recognizes that it is a PACKET_IN message, it looks up which components registered for handling this kind of event. In Bonfire, these components are the topology module, the client discovery module and the routing module. This is also the defined order of execution.
Beacon then calls the topology module first. It notes that there was an incoming packet on a certain switch via a certain link and sends this information to the datastore. As it is no LLDP ping answer, that is used to find links between switches, it does not consume the packet and returns CONTINUE to Beacon Core.
Next, Beacon calls the client discovery module. This component sends the datastore the information that a live client is at the corresponding switch. If the client were not already known to the system, the module would try to extract MAC address and IP addresses from the packet and send this information to the datastore as well. Also, it would consume ARP ping replies, the requests for which are regularly sent to the client. As this is no such packet, the Beacon core gets the command to CONTINUE handling the packet.
Finally, the packet arrives at the routing module. It first decides what kind of routing the packet requires: multicast, broadcast or, as it is here, unicast. As unicast packets have a destination, it check with the datastore if the destination is known to the controller. As

this is the case, it orders the shortest path algorithm to compute a route to the destination, which does so by getting the topology information from the datastore and doing the necessary computations.

When the SSSP algorithm returns a route, the routing component enters a new flow in the datastore and installs the flows on the appropriate switches. Afterwards, it return STOP to Beacon core, as it consumed the packet.

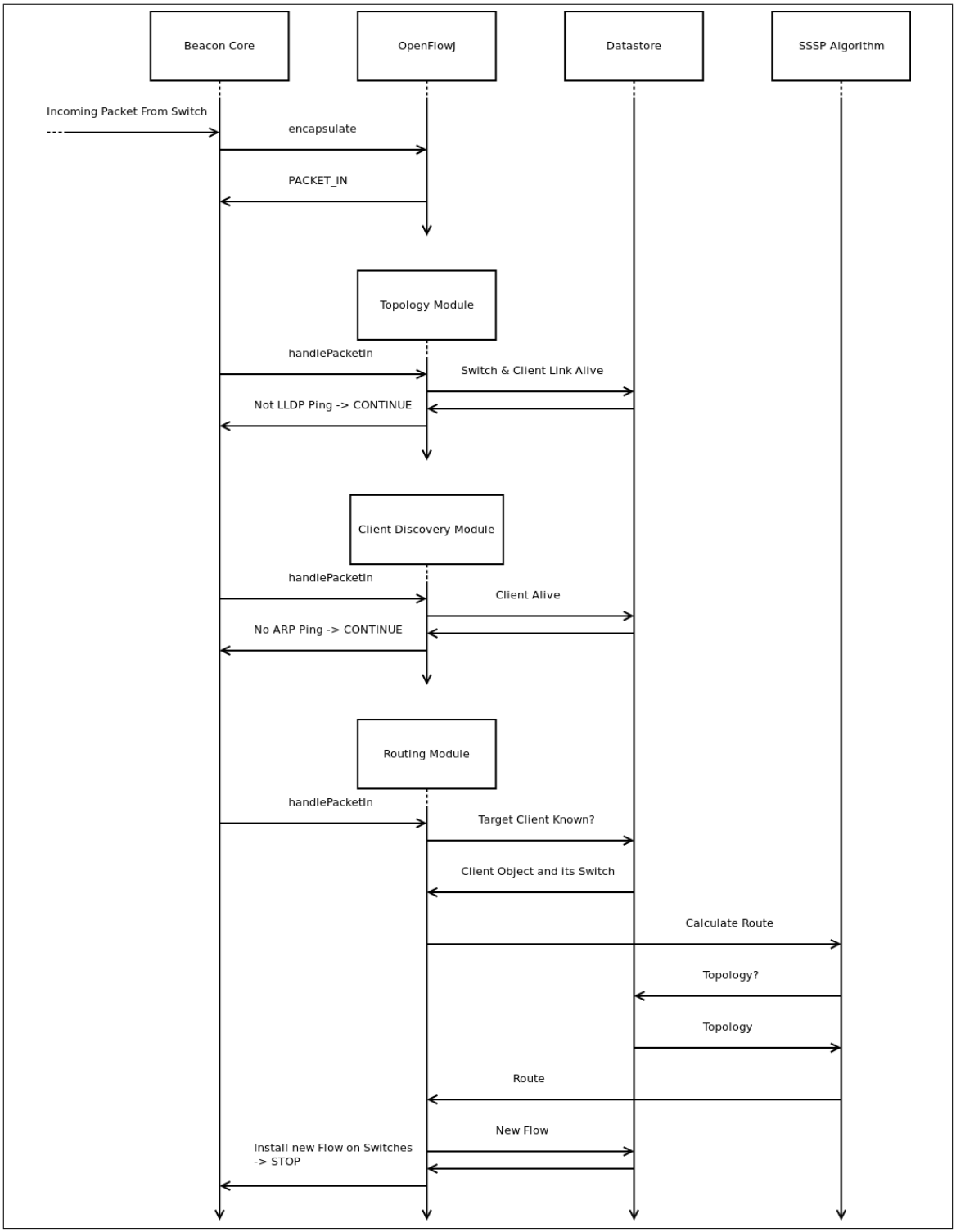

Figure 4.8: Example how Bonfire components processes a packet in event

4.8 Challenges

There were several challenges when writing the code for this thesis, some of which are listed in the following sections.

4.8.1 Controller Frameworks

There are already several OpenFlow controllers written in multiple languages available: *Nox Classic* [48] is written in C, with plugin interfaces available in C and Python, its successors, *Nox* [48] and *Pox* [49] are written in C++ and Python respectively and only support other components in its own languages. *Beacon* [44] and an industry-backed controller called *Floodlight* [50] are written in Java and its plug-in capabilities are created by implementing OSGi standard interfaces, with some support of the Spring framework. During the development of this controller, the Linux Foundation published their own SDN controller called *OpenDaylight* [41]. It too, is written in Java, already has a multitude of plugins available and is even SDN API-agnostic.

The choice to use Beacon as a framework was only made after seriously considering all alternatives. Nox Classic is no longer supported by its developers, that gave up the project for Nox and Pox. The Nox controller lacks documentation and is therefore hard to work with, while the Pox controller is written in Python, which is easy to write code for, but is also comparatively slow. As OpenDaylight was, at the time of this decision, not yet released, the choice was between Beacon and Floodlight, which are both well-documented and supported, but Beacon was deemed better suited due to its very modular structure.

4.8.2 GPU Support Library

As previously mentioned, for CUDA, there is a library that helps programmers with creating code that uses standard data structures, which would have been ideal for accelerating the data store. It is called Thrust [28] and is an implementation of the Standard Template Library for C++. Unfortunately though, this thesis uses the Beacon framework in Java with OpenCL.

A similar implementation exists for Java and OpenCL, though. It is called Aparapi [51], was created by AMD and is available under a non-standard open source license. It supports a more comfortable development of kernel code, as it can be written in native Java. Additionally, the library also eases the pain of selecting the correct device and managing the allocated resources.

Unfortunately though, Aparapi also takes over memory management from the programmer, which means that the library automatically copies kernel arguments to the compute device and copies the result back later. This makes it impossible to keep data on the device, which means a loss of control over an important part of the controller.

4.8.3 Defective Dijkstra Algorithm

When implementing the single-source shortest path algorithm, the choice to use Dijkstra was made because it is a well-known algorithm that has a good performance and is yet basic. An additional point in the favor of the algorithm is, that there already exists an OpenCL implementation of it. It can be found in the book "OpenCL Programming Guide" [9] on page 411ff. The parallel implementation of the algorithm itself was written for CUDA by Harish et al. [8].
The parallelized version of the algorithm (listing 4.6) itself though, has an error: It contains a data race, that may occur when two or more threads try to correct the cost for going to the same node. Even though it is protected by an "if"-statement before writing to the variable, threads in single work group execute the statement at the same time, each still evaluating to true as the value is not yet changed by another thread. In the next step, all threads calculate the new cost and then try to write it *at the same time*. If the node relaxing the other node is then not relaxed any more, the wrong cost may propagate and yield a wrong result. This issue is confirmed by a bug report on the book's project page, but was never fixed [52]. A possible execution that may lead to this bug is depicted in figure 4.9
One possible solution for fixing this bug is executing the kernel for all vertices until nothing changes any more, because each iteration eliminates at least one thread from the data race as it always changes the value to a lower one than in the previous iteration.
In addition, the algorithm does not keep track of the links that are taken to reach each node, which is necessary for computing a route, though. Simply adding this assignment in the if-statement would, however, create another data race, the outcome of which does not necessarily correspond to that of the previous cost assignment.
The solution to this problem is, at least for an undirected graph, saving the edge that would be taken to relax the path cost in phase one and later, in phase 2 actually applying the change to the data set. This solution is added to executing the kernel for each vertex in every iteration, which leads to a correctly functioning algorithm that outputs the path. It is already implemented in this thesis and can be found in section 4.5.1.

```
for(int edge = edgeStart; edge < edgeEnd; edge++) {
 int nid = edgeArray[edge];
 if (updatingCostArray[nid] > (costArray[tid] +
                               weightArray[edge])) {
  updatingCostArray[nid] = (costArray[tid] +
                            weightArray[edge]);
 }
}
```

Listing 4.6: Position of data race in original parallel implementation of Dijkstra's algorithm

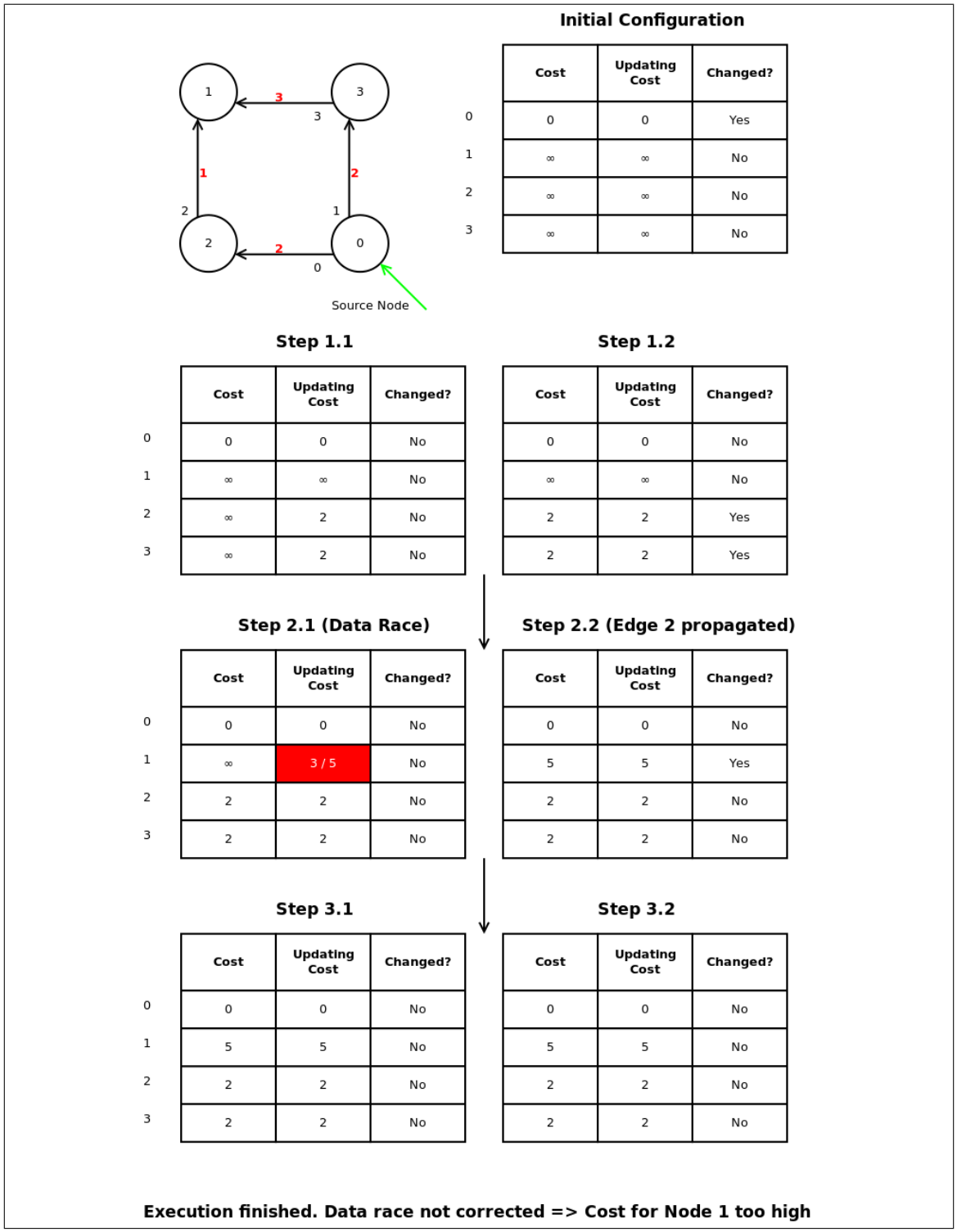

Figure 4.9: Execution of defective SSSP algorithm from book

Chapter 5

Evaluation

In order to be able to properly evaluate a program as complex as an SDN controller, that comprises of several components, it is important to test the single components, as well as the performance of the program as a whole. As the main focus of this thesis is on comparing a CPU-based controller with a GPU-based one, only the components that actually use the GPU are compared to its non-accelerated counterparts, i.e. only the routing algorithm and the datastore are tested as single components. In addition to that, the performance of the program as a whole is evaluated.
In order to also examine, whether transfer rates between the host memory and the graphics card's memory limit the execution performance of the controller, all tests are executed on both a device with a dedicated graphics card, as well as a device with a graphics chip on the same chip as the CPU. Such devices have the advantage, that their memory is already a dedicated part of the host's main memory, so data transfers between the two memories should be up to or even as fast as copying memory areas in RAM.

5.1 Testbed

There are two devices on which the tests are performed. The first device has a *dedicated graphics card* and the second device has an *integrated graphics processor* (IGP) on the CPU. The exact specifications are as follows:

1. Custom-built desktop PC with Intel Desktop Board DQ965GF mainboard, Intel Core2Duo E6300 with 2 cores at 1.86GHz, 8GB of DDR2-RAM at 667MHz. It has an AMD Radeon HD 7970 graphics card with 3GB of GDDR5-RAM, yielding a data rate of up to 273.6GB/s between graphics RAM and GPU. Note, that even though the card supports being connected to the mainboard via PCIe 3.0 16x, the mainboard only supports PCIe 1.0 16x (up to 15754MB/s), which caps the maximum transfer rate between host and device at around 4000MB/s.

2. Schenker S413 notebook[1], with Intel Core i7-4750HQ processor with 4+4 HT Cores at 2.0-3.2GHz, 16GB DDR3-RAM and integrated Intel Iris Pro 5200 IGP on the CPU.

For the component evaluation, the desktop computer executes a test suite, specifically designed to test the component performance in an environment that excludes the other controller components as good as possible. The exact dimensions of the test suites are described in the corresponding section.
For the datastore component, all tests are also performed on the laptop PC with the IGP in order to find out whether shared memory between GPU and CPU is beneficial.
The evaluation of the complete controller as a whole will take place in Saarland University's own OpenFlow testbed.
The testbed consists of six virtualized machines in the university's data center that run Ubuntu 10.04 with the OpenFlow switch reference implementation, as well as several TP-Link TL-WR1043ND access points with OpenWRT and the corresponding OpenFlow switch software. The network between these devices is virtualized within the university's normal network. Additionally, two remote locations are connected to the network, one at the *Helsinki Institute for Information Technology*, the other at the *Smart Factory* in Kaiserslauten. The connection graph can be seen in figure 5.1. Connections to the controllers are established in an out-of-band network, first to a FlowVisor instance that enables testing of multiple controllers within the same network and without restarting the switch software instances.

5.2 Single-Source Shortest Path Component

For measuring the performance of the sequential and parallel versions of Dijkstra's algorithm, randomized graphs were created and the time for the algorithm to run once was measured.
The tests were executed for different number of vertices and edges on both the CPU and the GPU. The numbers of edges range from 4 to 64, which might already be high for networks in which mostly servers are, but which is still considered.
The values for the number of switches range from 16 to 65536, which is also a high estimate for the number of switches in a data center.
The exact results can be found in table 5.1. Please note that both axes are in logarithmic scale, to be precise base 10 for y-axis and base 2 for x-axis. Also note, that testing was aborted for the sequential version after the execution time became larger than 15 minutes.
The results of the measurements suggest, that while the GPU is by up to a factor 10 slower for smaller work sizes, it thereafter becomes faster than the CPU version by orders of

[1]The OEM for this is Clevo, its model number is W740SU, also available as Sager NP2740 and System76 Galago UltraPro

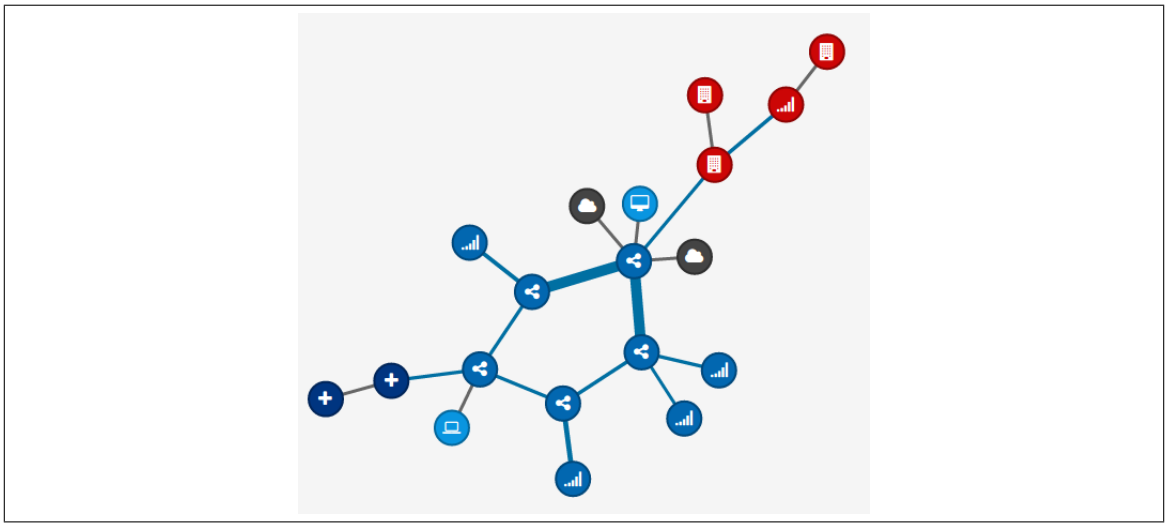

Figure 5.1: Topology of OpenFlow Network at Saarland University. Visualization via OFViz by Andreas Schmidt [11]

magnitude. This is probably due to the comparatively large overhead of copying the data over to the device and executing the kernels on it.

Unfortunately though, for a large number of links and a large number of edges, the time it takes to compute a path through the network becomes unfeasibly high (>100ms) to do it on demand. In this case, it might be beneficial to proactively precompute and cache the routes.

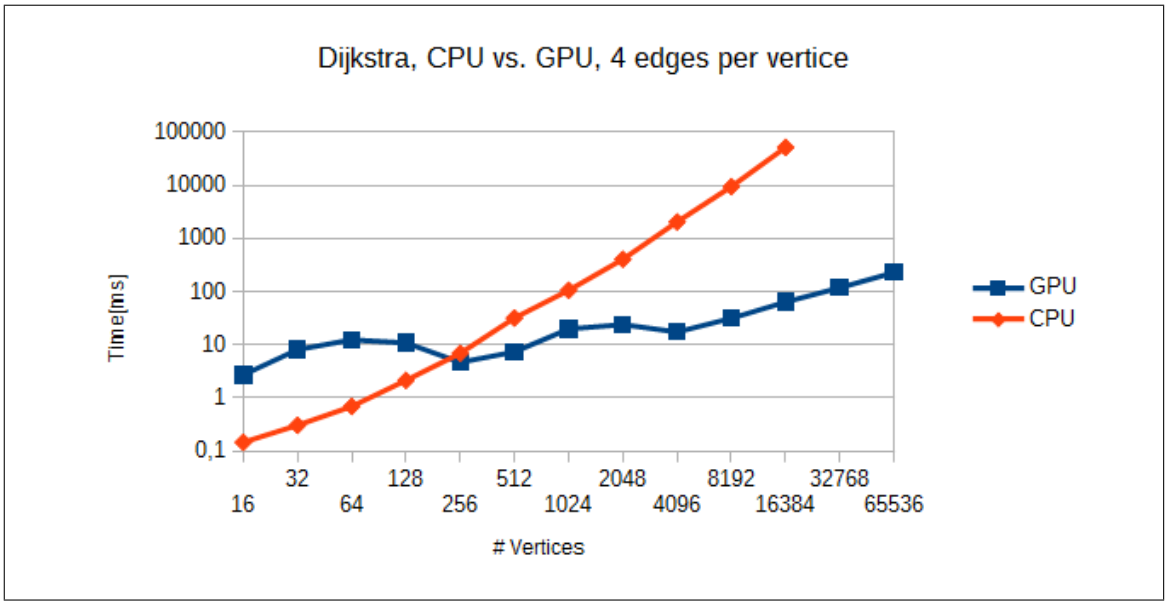

Figure 5.2: Graph of measurements with Dijkstra's algorithm with 4 edges

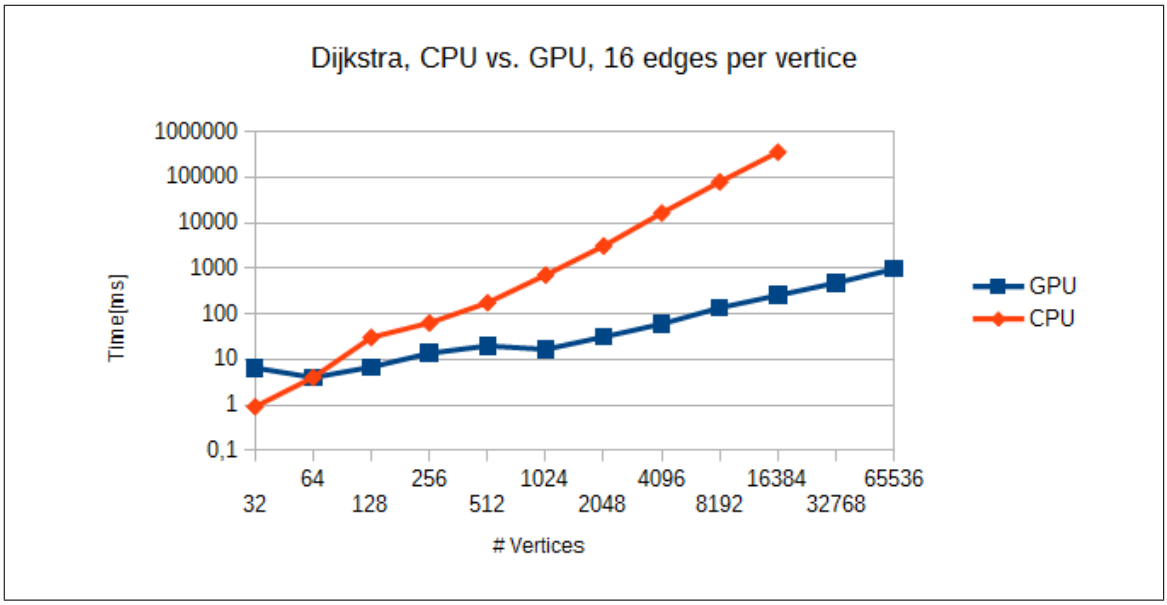

Figure 5.3: Graph of measurements with Dijkstra's algorithm with 16 edges

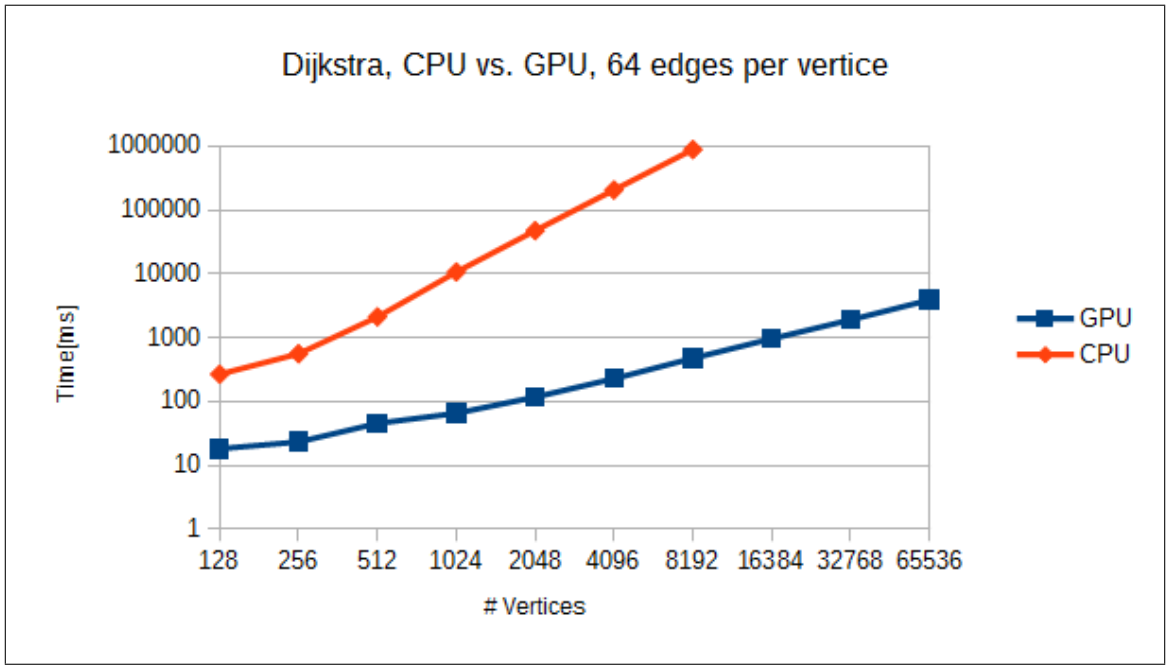

Figure 5.4: Graph of measurements with Dijkstra's algorithm with 64 edges

Edges	**Vertices**	**GPU time[ms]**	**CPU time[ms]**
4	16	2.720722	0.147301
4	32	8.116339	0.307567
4	64	12.447079	0.696023
4	128	10.854201	2.152634
4	256	4.688315	6.922384
4	512	7.362987	31.920989
4	1024	19.911914	106.156378
4	2048	24.050073	406.16374
4	4096	17.421693	2050.916004
4	8192	31.506228	9438.031317
4	16384	64.02539	51671.586355
4	32768	118.90646	
4	65536	229.665606	
16	32	6.54237	0.92478
16	64	4.029229	4.049552
16	128	6.849724	30.515229
16	256	13.67246	63.517857
16	512	19.965281	176.385238
16	1024	16.697578	718.25544
16	2048	31.374299	3107.809856
16	4096	60.217826	16366.980274
16	8192	136.676911	79226.751853
16	16384	255.682678	355409.747078
16	32768	481.151379	
16	65536	968.176994	
64	128	18.30241	269.340934
64	256	23.459185	564.059288
64	512	45.727757	2136.242324
64	1024	65.837699	10786.462169
64	2048	118.094014	48004.879122
64	4096	229.355129	208217.557121
64	8192	478.597393	895714.350254
64	16384	967.424327	
64	32768	1916.072547	
64	65536	3939.734469	

Table 5.1: Measurements of time in ms for Sequential vs Parallel SSSP Performance

5.3 Datastore Component

In order to test the performance of the datastore, it is filled with dummy switches and links between those switches. The test then tries to simulate realistic behavior by accessing data in a pattern that is designed to resemble realistic behavior when checking for switch and link timeouts or accessing link data for route calculations.

As previously mentioned, this test is run with a CPU-based version of the datastore, as well as with a GPU-based version on both a dedicated graphics card, as well as on an internal GPU.

In addition to measuring the total execution time of the test suite, the time it takes to add the data is also output. This is done because if most of the time were spent on the initialization of the test data, then one might argue that once the controller is fully booted up, it becomes significantly faster.

Similarly to the test suite for the SSSP algorithm, this one tests values from 4 to 4096 for the number of switches with 4, 16 and 64 links to other switches per switch.

The detailed results can be seen in table 5.2, as well as figures 5.5, 5.6 and 5.7. There are two things that can be learned by this experiment: Using the graphics card, the dedicated as well as the integrated ones, the measurements are significantly worse, most of the times by at least an order of magnitude. The other take-away is that, by looking at the raw data, it becomes apparent that even though the time used for initialization is at an average of around 80% for the GPU version, the remaining queries are still slower than the approximately 70% of the time the CPU version needs.

As the graphs, however, do show a tendency towards the CPU version becoming significantly slower with larger argument sizes, the GPU version will eventually overtake the CPU version. Then, however, the time a single query takes is so large that executing it at all on one controller is not feasible and the user should think about other alternatives for taking load off of her SDN controller.

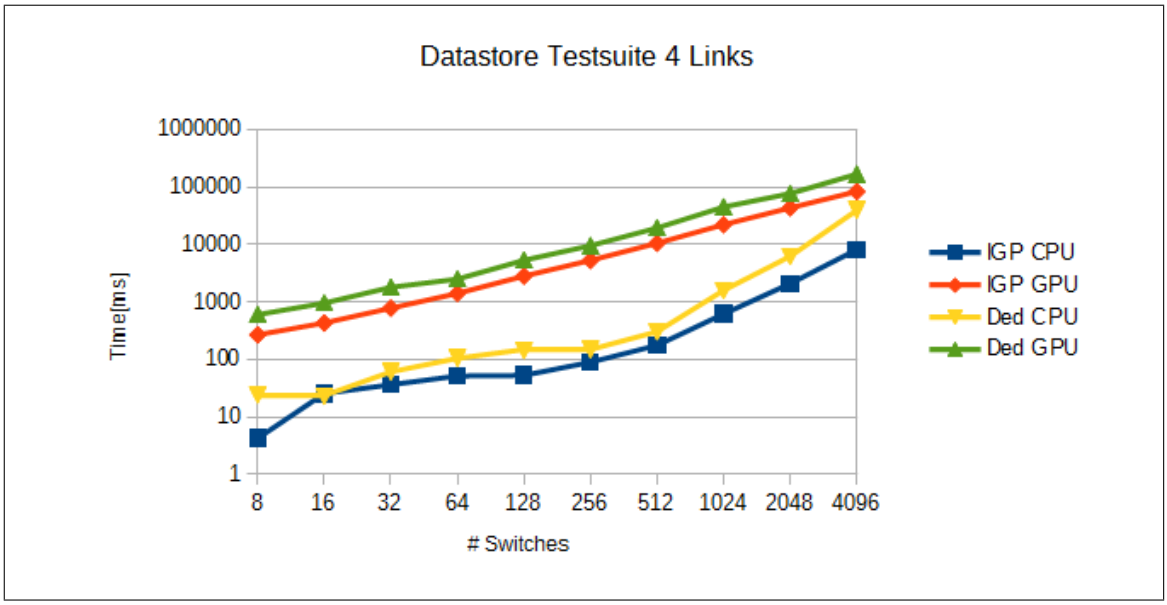

Figure 5.5: Graph of measurements with datastore test suite with 4 edges

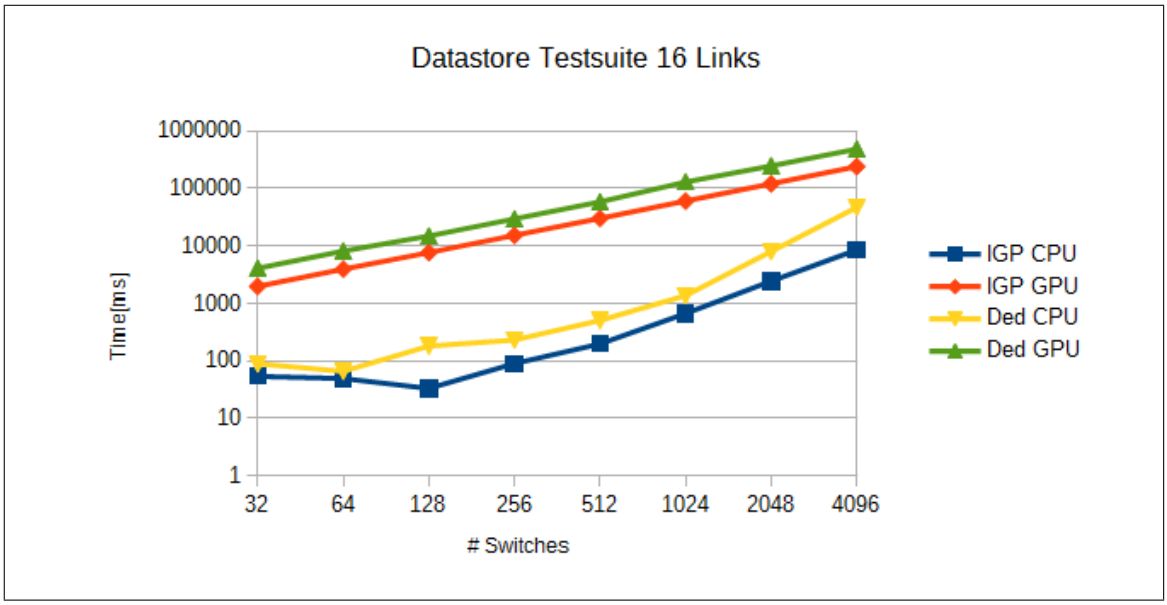

Figure 5.6: Graph of measurements with datastore test suite with 16 edges

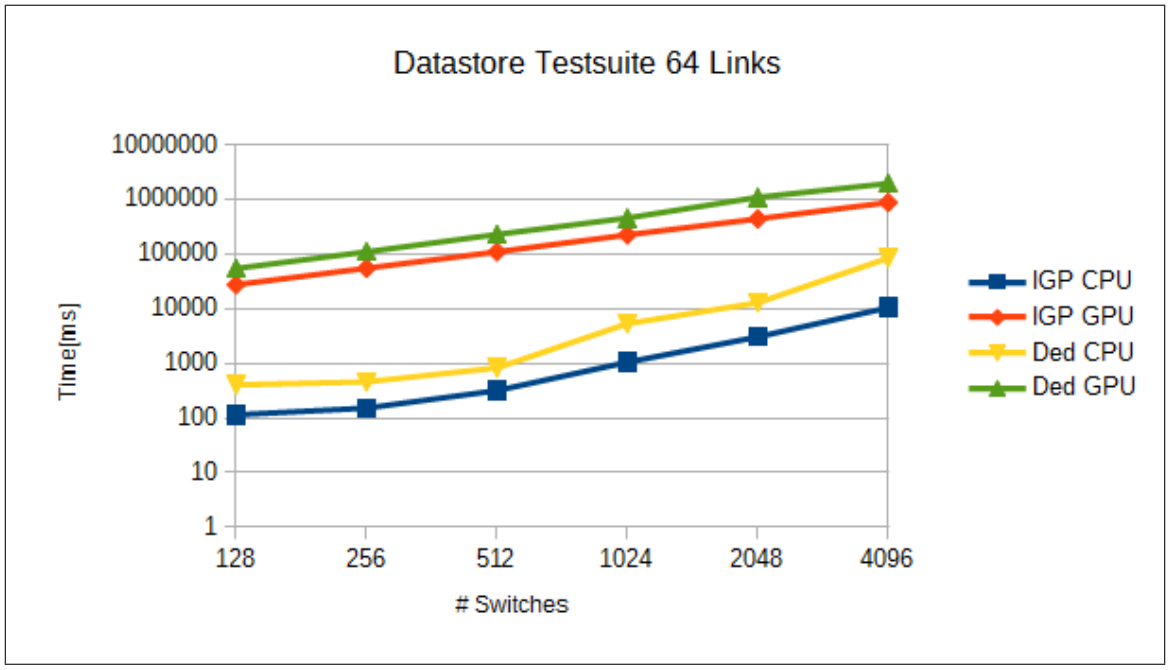

Figure 5.7: Graph of measurements with datastore test suite with 64 edges

5.4 Controller Performance

The performance of the whole controller is measured by using the University's OpenFlow testbed.
As for the controller software, Bonfire will run with a CPU-only component set, with a GPU-accelerated SSSP algorithm and a CPU-only datastore, with a CPU-only SSSP-algorithm and with both components being executed on the GPU. The test suites are executed on the desktop PC with the dedicated graphics card.
Initially, it was planned to also use a software called Mininet [53], but the software's high initialization time paired with unsuited time measurement features lead to the decision, not to use it.
With this test, there is a predefined set of parameters: There are seven switches with seven switch links and two clients. One of the clients sends a ping to the other. The measured time is the round-trip time of said ping, taken 5 times and then averaged.
The results can be observed in table 5.3. They suggest that the completely CPU-based version is the optimal configuration for this network, which is confirmed by the measurements the previous chapters.

Links	Switches	IGP CPU	IGP GPU	Ded CPU	Ded GPU
4	8	4.241985	267.908608	24.240501	601.65122
4	16	25.441133	430.532234	22.988228	952.799319
4	32	36.279596	780.411328	60.689302	1803.438312
4	64	51.425217	1411.184143	104.393706	2498.103723
4	128	52.845199	2821.120221	143.816794	5327.036314
4	256	89.397295	5302.480838	147.746548	9561.472191
4	512	175.707271	10536.433037	303.147636	19437.525015
4	1024	618.468698	22166.423759	1593.345754	45023.232828
4	2048	2088.865123	43375.18218	6223.704942	76933.360253
4	4096	8166.223104	83656.446421	39478.906357	167571.965175
16	32	53.943414	1990.501977	89.497844	4098.532012
16	64	49.405835	3965.187086	66.192202	8196.490292
16	128	33.330836	7700.586981	181.188084	14972.535046
16	256	90.078805	15357.1812	231.356252	29725.52721
16	512	198.240187	30363.765873	507.335711	58645.08828
16	1024	668.728985	60682.772799	1375.503454	130398.352744
16	2048	2458.417666	121286.506423	8048.792359	248395.736563
16	4096	8582.488802	241606.226675	47220.476063	487513.125543
64	128	114.660867	27624.881968	405.374958	54508.798425
64	256	152.131051	54949.178289	459.716762	110275.556254
64	512	318.379301	109681.043366	825.968695	228548.547426
64	1024	1052.066698	224340.960456	5334.869638	455997.799189
64	2048	3050.883294	439964.614373	12901.429634	1090605.143344
64	4096	10488.359944	876235.412723	84560.378933	1949834.819128

Table 5.2: Measurements of time in ms for Sequential vs Parallel and IGP vs Dedicated GPU Datastore Performance

	SSSP CPU	SSSP GPU
Datastore CPU	18.64ms	18.22ms
Datastore GPU	134ms	84.12ms

Table 5.3: Measurements of time in ms for pinging another host on the network for different component combinations

Chapter 6

Conclusion

This thesis tackles the performance problem that a single controller in a large Software Defined Networking Environment poses. For this, the idea of outsourcing some computationally expensive algorithms to the GPU is presented. After explaining the concept of SDN, the basics of GPU computing are introduced and illustrated by several examples. Afterwards, the components of an SDN controller are examined in detail, their inner workings explained and searched for algorithms that could be accelerated by a GPU.

As a result, the GPU-aided SDN controller Bonfire is introduced. It has two GPU-accelerated components: The datastore, which is the central module of the controller, is used to store all information about the network that the controller has. In addition to that, a GPU-accelerated single-source shortest path algorithm that is based on Dijkstra's algorithm is used to compute the ideal route for a given packet type through the network.

After that, the Bonfire controller is evaluated by first testing the individual components on the CPU and the GPU. For the datastore, there are also comparisons between dedicated graphics cards and GPUs that are integrated with the CPU and share its memory. As it turns out, both GPU-based solutions do not come close to the performance of accessing it via the CPU, which is presumed to be due to the comparatively large overhead that GPU computing entails. The GPU-based SSSP algorithm however, can accelerate computations in realistic networks. A result, that the experimental evaluation is a realistic testbed network confirms.

While the GPU-based datastore is not an ideal solution for the problem, the parallel SSSP algorithm can be integrated in production environments and larger networks may profit event more from it, as the algorithm's runtime grows significantly slower than its sequential counterpart.

Additionally, outsourcing algorithms to the GPU helps take load off of the CPU, of which the remainder of the controller may benefit.

Future Work

There are several steps that may be taken in the future:

- **SSSP Breakeven Heuristic**: As can be seen very nicely in e.g. figure 5.3, there is a certain point in the input values, at which the GPU-based version of the algorithm becomes faster than the CPU-based one, which is due to the overhead that GPU computing entails. Such a point is called a breakeven point. If one were to develop a heuristic that checks where this point lies, it could dynamically, at run-time, decide which version of the algorithm to use. This would combine the low overhead of the CPU-based algorithm for small network with the high efficiency of the GPU-accelerated version.

- **Integration in Major SDN Frameworks**: Because the design of the GPU-based datastore is so fundamentally different from the way Beacon's own components are implemented, most parts of Beacon were not used in the implementation of this controller. With the new datastore proven inefficient but the new SSSP algorithm shown useful, this step could be reversed. The GPU-accelerated version of Dijkstra's algorithm could be integrated into the regular Beacon framework and even others such as OpenDaylight. Some engineering work would have to be done to make it fit the given interfaces, but it should be possible.

- **More Intelligent Caching**: Currently, the SSSP algorithm caches most data in Java data structures that help initialize the algorithm faster. Yet, due to the way, that it requires its arguments, the view of these *relative* data structures has to be "materialized" which means that the items must be given concrete, *absolute identifiers*. This view is cached too, unless the network changes. Unfortunately, this happens every time the link metrics are refreshed, which means that this cache is invalidated very often. This may be mitigated by not invalidating the whole cache, but rather only the edge weights, which would accelerate subsequent executions of the algorithm even more.

- **GPU Implementation of Multicast Tree Algorithm** The current implementation focuses on the single-source shortest path algorithm that is used for unicast routing. When doing multicast or broadcast routing however, the problem structure changes and different algorithms have to be used to tackle them. For some of these, it may even be feasible to implement them on the GPU.

Bibliography

Papers

[1] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, Guru Parulkar, *Can the Production Network Be the Test-bed?*, `http://www.deutsche-telekom-laboratories.de/~robert/flowvisor-osdi10.pdf` ODSI, 2010

[2] Kamran Karimi, Neil G. Dickson, Firas Hamze, *A Performance Comparison of CUDA and OpenCL*, `http://arxiv.org/ftp/arxiv/papers/1005/1005.2581.pdf`, 2010

[3] Nate Foster, Rob Harrison, Michael J. Freedman, Jennifer Rexford, and David Walker, *Frenetic: A High-Level Langauge for OpenFlow Networks. Technical report*, `http://ecommons.library.cornell.edu/bitstream/1813/19310/4/frenetic-tr.pdf`, Cornell University, December 2010

[4] Flynn, M. J., *Some Computer Organizations and Their Effectiveness*, `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5009071`, September 1972

[5] De-Nian Yang, Wanjiun Liao *On Multicast Routing Using Rectilinear Steiner Trees for LEO Satellite Networks*, `ieeexplore.ieee.org/iel5/25/4563511/04384138.pdf`, downloaded in June 2014

[6] G. Maier, *NAT usage in Residential Broadband Networks*, `www.icir.org/gregor/papers/pam11-nat.pdf`, 2011

[7] D.J. Abadi et al. *Column-Stores vs. Row-Stores*, `http://db.csail.mit.edu/projects/cstore/abadi-sigmod08.pdf`, SIGMOD'08

[8] Pawan Harish, P.J. Narayanan *Accelerating Large Graph Algorithms on the GPU Using CUDA*, `http://ldc.usb.ve/~vtheok/cursos/ci6323/pdf/lecturas/Accelerating%20Large%20Graph%20Algorithms%20on%20the%20GPU%20Using%20CUDA.pdf`, 2007

Books

[9] Aaftab Munshi, Benedict Gaster, Timothy G. Mattson , James Fung, Dan Ginsburg *OpenCL Programming Guide*, https://code.google.com/p/opencl-book-samples/, Addison-Wesley, First printing, July 2011, ISBN13: 978-0-321-74964-2

Theses

[10] R. Yanggratoke, *GPU Network Processing*, http://www.diva-portal.org/smash/get/diva2:561258/FULLTEXT01.pdf, Stockholm, June 2010

[11] Andreas Schmidt, *Interactive Visualization of Software Defined Networks*, http://www.openflow.uni-saarland.de/fileadmin/downloads/publications/theses/IVOSDN_Andreas_Schmidt.pdf, December 2013

Standards

[12] IEEE 802.1AX, *Link Aggregation*, https://development.standards.ieee.org/get-file/P802.1AXbq.pdf?t=73972900003, September 2011

[13] IEEE 802.1D, *Spanning Tree Protocol*, http://standards.ieee.org/getieee802/download/802.1D-2004.pdf, 1990

[14] Network Working Group *Introduction and Applicability Statements for Internet Standard Management Framework*, http://tools.ietf.org/html/rfc3410, December 2002

[15] Internet Engineering Task Force *Network Configuration Protocol (NETCONF)*, http://www.ietf.org/rfc/rfc6241.txt, June 2011

[16] Intel *x86 Reference Manual*, http://download.intel.com/design/processor/manuals/253667.pdf

[17] PCI SIG *PCIe Base Specification*, http://www.pcisig.com/specifications/pciexpress/, downloaded in June 2014

[18] IEEE *IEEE 802.1Q*, http://standards.ieee.org/getieee802/download/802.1Q-2005.pdf, downloaded in June 2014

[19] Network Working Group *IP Network Address Translator (NAT) Terminology and Considerations*, http://tools.ietf.org/html/rfc2663, August 1999

[20] Network Working Group *An Ethernet Address Resolution Protocol*, http://tools.ietf.org/html/rfc826, November 1982

[21] IEEE *802.1AB*, http://standards.ieee.org/getieee802/download/802.1AB-2009.pdf, downloaded in June 2014

Websites

[22] Open Networking Foundation, *ONF Overview*, https://www.opennetworking.org/about/onf-overview, downloaded in October 2013

[23] Microsoft, *High Level Shading Language for DirectX*, http://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx, October 2013

[24] The Khronos Group, *The OpenGL Shading Language Specification*, http://www.opengl.org/registry/doc/GLSLangSpec.4.20.8.clean.pdf, September 2011

[25] AMD, *AMD Radeon HD 7970 GHz Edition Specifications*, http://www.amd.com/us/products/desktop/graphics/7000/7970ghz/Pages/radeon-7970GHz.aspx#3, downloaded in October 2013

[26] Cisco, *Multi-Layer Switching, Introduction*, http://www.cisco.com/en/US/tech/tk389/tk815/tk850/tsd_technology_support_sub-protocol_home.html, downloaded in October 2013

[27] The Khronos Group *OpenCL*, http://www.khronos.org/opencl/, downloaded in October 2013

[28] *Thrust*, http://thrust.github.io, downloaded in October 2013

[29] NVIDIA *Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs*, https://developer.nvidia.com/content/precision-performance-floating-point-and-ieee-754-compliance-nvidia-gpus, downloaded in October 2013

[30] Cisco *EtherChannel - Introduction*, http://www.cisco.com/en/US/tech/tk389/tk213/tsd_technology_support_protocol_home.html, downloaded in October 2013

[31] ZTE *LACP Configuration on ZXR10 8902 Switch*, http://zte.by/magazine/Maintenance%20Experience%20Issue219(Data%20Products).pdf, June 2010

[32] Cisco *Cisco Network Assistant*, http://www.cisco.com/web/DE/pdfs/products/CiscoNetworkAssistantDataSheet_Deutsch_Final.pdf, downloaded in June 2014

[33] Open Networking Foundation *OpenFlow White Papers*, https://www.opennetworking.org/sdn-resources/sdn-library/whitepapers, downloaded in June 2014

[34] Open Networking Foundation *OpenFlow Legal Information*, http://archive.openflow.org/wp/legal/, 2011

[35] Open Networking Foundation *Member Listing*, https://www.opennetworking.org/membership/member-listing, downloaded in June 2014

[36] Open Networking Foundation *ONF Specifications ¿ OpenFlow*, https://www.opennetworking.org/sdn-resources/onf-specifications/openflow, downloaded in June 2014

[37] Intel *Intel Xeon Phi Coprocessor 5110P*, http://ark.intel.com/de/products/71992/Intel-Xeon-Phi-Coprocessor-5110P-8GB-1_053-GHz-60-core, Q4 2012

[38] NVIDIA *Dynamic Parallelism in CUDA*, http://developer.download.nvidia.com/assets/cuda/docs/TechBrief_Dynamic_Parallelism_in_CUDA_v2.pdf, 2012

[39] Edsger W. Dijkstra, *A note on two problems in connexion with graphs*, http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf, 1959

[40] Rashid Bin Muhammad *Steiner Tree Problem's Heuristic with Minimum Spanning Tree Problem*, http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/CG-Applets/SteinerTree/msteinercli.htm, downloaded in June 2014

[41] Linux Foundation *OpenDaylight: Technical Overview*, http://www.opendaylight.org/project/technical-overview, downloaded in June 2014

[42] RIPE NCC *IPv4 Exhaustion*, http://www.ripe.net/internet-coordination/ipv4-exhaustion, downloaded in June 2014

[43] Google *IPv6 Statistics*, http://www.google.com/intl/en/ipv6/statistics.html, downloaded in June 2014

[44] David Erickson *The Beacon OpenFlow Controller*, http://yuba.stanford.edu/~derickso/docs/hotsdn15-erickson.pdf, HotSDN'2013

[45] David Erickson *OpenFlowJ*, https://bitbucket.org/openflowj/openflowj, September 2013

[46] *Java bindings for OpenCL*, , downloaded in June 2014

[47] Lars Vogel *Dijkstra's shortest path algorithm in Java - Tutorial*, http://www.vogella.com/tutorials/JavaAlgorithmsDijkstra/article.html, November 2009

[48] Nicira *Nox Versions*, http://www.noxrepo.org/nox/versions-downloads/, downloaded in June 2014

[49] Nicira *About POX*, http://www.noxrepo.org/pox/about-pox/, downloaded in June 2014

[50] Big Switch Networks *Floodlight*, http://www.projectfloodlight.org/floodlight/, downloaded in June 2014

[51] AMD *Aparapi*, http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-libraries/aparapi/, downloaded in June 2014

[52] Google Code Member *OpenCL Programming Guide Issue 68*, https://code.google.com/p/opencl-book-samples/issues/detail?id=68, February 2012

[53] Mininet Team *Mininet - An instant Virtual Network on your Laptop (or other PC)*, http://mininet.org/, downloaded in June 2014