MASTER'S THESIS | LUND UNIVERSITY 2017

Specification and Visualization of Interconnection Networks of Mobile GPUs

Björn Wictorin

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2017-32

Specification and Visualization of Interconnection Networks of Mobile GPUs

Björn Wictorin
lak12bwi@student.lu.se

December 21, 2017

Master's thesis work carried out at Arm Sweden AB.

Supervisors: Fuad Tabba, fuad.tabba@arm.com
Jörn Janneck, jwj@cs.lth.se

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

The number of components that can be fit into a single chip is increasing over time, because of improved manufacturing technology. An efficient way to connect the components is through a packet switched network, a Network on Chip. Such a network can be designed in many different ways. An effective way to explore the large design space is by simulation. In this project, we have developed a network description language to define the network of a mobile GPU. As part of this process, we have surveyed how NoCs are specified in several simulators. We have implemented support for the language in a GPU simulator. We have also implemented a script that outputs a visual representation of networks defined in the language.

The purpose of the project has been to develop tools to make Network on Chip modeling faster, easier, and less error prone. The result was evaluated qualitatively and quantitatively.

Keywords: NoC, GPU, modeling, simulation, visualization

Acknowledgements

I would like to thank Fuad Tabba, my supervisor at Arm, for all the help and knowledge he has provided me with throughout the project.

I would also like to thank the GPU modeling team at Arm, for being welcoming and supportive during the project.

Finally, I would like to thank Jörn Janneck, my supervisor at Lund University, for his support and comments on my thesis.

Contents

1 Introduction **9**
1.1 Goals 10
1.2 Contribution to the Knowledge within the Field 10
1.3 Graphics Processing Units (GPUs) 10
1.3.1 The Graphics Rendering Pipeline 10
1.3.2 GPU Hardware 11
1.4 Networks on Chips (NoCs) 12
1.4.1 Motivation 12
1.4.2 Development 13
1.4.3 Protocol Stack 13
1.4.4 Topology and Routing 13

2 NoC Simulator Survey **17**
2.1 Information Sources 17
2.2 Netjson 18
2.3 NoC Simulators 18
2.3.1 Atlas 19
2.3.2 BookSim 19
2.3.3 Darsim 20
2.3.4 Noxim 21
2.3.5 VisualNoC 21
2.3.6 Ns-2 22
2.3.7 Garnet and gem5 23
2.4 Conclusion 23

3 Approach **25**
3.1 Background Survey 25
3.1.1 GPU and NoC Theory 25
3.1.2 Network Description in NoC Simulators 25
3.1.3 Markup Languages 26

3.2 Definition of Criteria . . . 26
3.3 Network Description Language . . . 27
3.4 Software Implementation . . . 28
3.4.1 Visualization Script . . . 28
3.4.2 Extension of the Arm GPU Model . . . 29
3.5 Verification . . . 32

4 Evaluation **35**
4.1 Fulfilled Criteria . . . 35
4.2 Lines of Code . . . 35
4.3 Flexibility of Description Files . . . 36
4.4 Time to Define Network . . . 36
4.5 Comparison with Source Code . . . 37
4.6 Implementation of New Topologies . . . 40
4.7 Discussion with our Supervisor . . . 41
4.8 Visualization . . . 41

5 Discussion **45**
5.1 Custom Topologies and Abstraction . . . 45
5.2 Support for Existing Topologies . . . 46
5.3 Lines of Code . . . 46
5.4 Flexibility of Description Files . . . 47
5.5 Comparison with Source Code . . . 47
5.6 Visualization . . . 48
5.7 Design Decisions . . . 49
5.8 New Topologies . . . 49
5.8.1 Mesh and Hash Tag . . . 50
5.8.2 Line and Narrow . . . 50

6 Conclusions **51**
6.1 Future Work . . . 51
6.1.1 Flexible Switches . . . 52
6.1.2 Routing Algorithms . . . 52
6.1.3 More Configuration Parameters . . . 52
6.1.4 RTL Generation . . . 52
6.1.5 List Comprehension . . . 52

Bibliography **53**

Appendix A Network Description Language Manual **59**
A.1 Introduction to JSON . . . 59
A.2 Outermost Object . . . 59
A.3 Switches . . . 61
A.4 Bypassable Switches . . . 61
A.5 Input Priorities . . . 61
A.6 Dead End Switches . . . 61
A.7 Crossbars . . . 62

A.8 Shader Cores 62
A.9 MMUs, L2 Cache Units, Tilers, and Job Managers 63
A.10 Links 63
A.11 Shader Core Stacks 64
A.12 Limitations 65
A.12.1 Switches and Crossbars 65
A.12.2 Shader Core Stacks 65

Appendix B Example Topologies **67**
B.1 Topology 1 67
B.2 Topology 2 68

Appendix C Markup Language Tests **73**

Chapter 1
Introduction

The System on Chip (SoC) field is developing at a high pace. Processors and memory elements keep getting faster, and the speed of an SoC is increasingly limited by the interconnection network connecting its components [1]. Traditionally, components have been connected through a bus [2]. This has allowed a simple design, but the speed of a bus is limited. This is due to its electrical features, and also that only one message can be relayed by a bus at any given time [1]. One solution is to replace the bus with a network of connections. Such a network connecting the components within an SoC is referred to as a Network on Chip (NoC), or an On-Chip Network (OCN) [3]. An introduction to NoCs is presented in Section 1.4.

Graphics Processing Units (GPUs) are hardware devices developed to accelerate computer graphics. They consist of many components, such as processing cores [3]. The components are connected to each other via an NoC. The processing cores of a GPU work in parallel. Due to their parallel structure, GPUs are increasingly used for general computing [3]. Arm develops a family of GPUs for mobile devices called Mali [4]. An introduction to GPUs is presented in Section 1.3.

In accordance with Moore's law, the number of transistors that can be fit into a single chip is increasing exponentially [5]. This allows for more complex SoCs, containing more components. As the complexity of the SoCs grows, so does the difficulty of designing them. All the possible design choices make up a large design space, in which efficient design space exploration is crucial. Formally proving the correctness of a design is infeasible in the general case. Producing a design and then testing whether it works as intended would be extremely expensive. Therefore, simulation is a valuable tool during the design space exploration [6].

For simulation to be possible, a model of the hardware has to be built. The model can either be completely custom developed for the specific hardware, or developed in some available simulation framework. An example of such a framework is gem5 [7]. Arm develops a model of their Mali GPU, which we have used in our project. We will refer to this model as the Arm GPU model.

1.1 Goals

The goal of this project is to develop tools that simplify and speed up the process of defining NoCs. The developed tools target the NoC in the Arm GPU model, and include a network description language with support for topology visualization. Defining and editing the topology of the NoC should be possible without knowledge of the underlying model code. We also want to make it easier to detect errors at an early stage, using the visualization tool.

1.2 Contribution to the Knowledge within the Field

To reach the project goals, we have created a human and machine readable network description language, designed particularly for configuration of NoCs. Furthermore, we have extended the Arm GPU model with support for this language. The extension serves as a proof of concept of how the language can be used to define the NoC in a simulator. We have also implemented a tool for visualization of the defined network.

The project presents an overview of a selection of NoC simulators in general, and how they handle visualization and configuration of network topologies in particular.

1.3 Graphics Processing Units (GPUs)

A GPU is a piece of computer hardware designed primarily to accelerate computations that generate visual content [8]. GPUs evolved from VGA (Video Graphics Array) controllers, which were the units that handled the drawing to the screen in older PCs. Since 1999, when the first consumer GPU was released and the term was coined by Nvidia [9], it has developed from being a device with fixed functionality to being highly programmable [8].

1.3.1 The Graphics Rendering Pipeline

To understand the reasons behind the design of GPUs, it is useful to be familiar with the graphics rendering pipeline. We will only provide an overview of the pipeline. Our overview is based on the description of the rendering pipeline in a book by Akenine-Möller et al. [9]. The interested reader is recommended to read further [9, 8].

The graphics rendering pipeline consists of a number of steps, which transform a 3D description of a scene to a 2D representation. The steps in the graphics rendering pipeline is pictured in Figure 1.1 below. The input to the pipeline is a 3D scene. The scene is described mainly by points in space called vertices. The pipeline performs a series of mathematical operations, of which many are matrix multiplications, to produce the desired output.

The first part of the graphics rendering pipeline is the application step. Its task is to render the primitives (points, lines, and triangles) describing the scene. The application step is performed by software running on the CPU rather than the GPU.

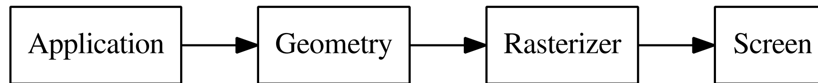

Figure 1.1: This figure shows the three major steps of the graphics rendering pipeline. For clarity, a screen has been added to the figure, even though it is not part of the pipeline.

The second step of the pipeline is the geometry step. It takes the primitives, which are produced by the application step, as input. The computations in this step can be done in parallel, and this step is performed by the GPU. The primitives are not necessarily described in the same coordinate system, and one task for the geometry step is to transform all primitives into one uniform coordinate system. Once the primitives are in that coordinate system, several transformations follow. The result is a space in which all objects are placed correctly relative to the viewing position.

After this, the appearance of objects is computed. This is called vertex shading, and examples of appearances that are computed in this step are the lighting and shadowing properties of a scene. After vertex shading follows projection, clipping, and screen mapping. Projection transforms the scene into a volume called a frustum, which corresponds to what the camera can see. Clipping decides which of the primitives that will fit inside the frustum. The screen mapping finally transforms the 3D primitives to screen coordinates. The screen coordinates contain x and y coordinates, as well as depth information called z-values.

The third step of the pipeline is called the rasterizer step. Its task is to make pixel values out of the vertices it receives as input from the geometry step. It does this by calculating which triangles will make up the surfaces connecting the vertices, and then traversing the triangles to compute which pixels will cover it. After this comes the pixel shading, during which all pixels are traversed and per-pixel computations are performed. This includes texturing, in which 2D images (textures) are applied on surfaces. Finally, the information from the triangle traversal and pixel shading is merged to calculate the color value for each pixel. If several objects are covered by the same pixel, which one is visible is resolved by consulting the z-values. Once the pixel values are computed, they are copied to the frame buffer, a memory space whose content will be displayed on screen.

1.3.2 GPU Hardware

The hardware partitioning of a GPU does not correspond directly to the steps of the graphics rendering pipeline. Traditionally, GPUs have three programmable computational units. They are called vertex shaders, geometry shaders, and pixel shaders. The vertex shader performs the vertex shading step of the pipeline, the geometry shader performs transformations of objects that can consist of several vertices, and the pixel shader performs the pixel shading step. The other steps in the pipeline are performed by fixed function hardware [9].

In modern GPUs, the vertex, geometry, and pixel shader have been replaced by unified processing units. We will, like Arm, refer to those unified units as shader cores [10]. A shader core is a computational unit that can be programmed to execute the functionality of the vertex, geometry, and pixel shader. Using unified shader cores provides several

benefits. First, only one hardware unit has to be developed, instead of three. Second, it gives flexibility. The number of cores can be varied depending on the desired trade-off between price and performance. Third, load balancing is simplified by having several identical units [8].

The calculations for the different pixels are independent to a large extent. Hence, the rendering computations can be executed in parallel. A large number of threads execute on the GPU simultaneously, several for each shader core. By adding more cores, more threads can be executed in parallel, resulting in higher performance. The large number of threads per shader core is used to hide memory access latencies. While one thread is waiting for its memory request to be ready, other threads get executed. This makes a lower cache hit rate acceptable in GPUs than in CPUs. In a GPU, an acceptable cache hit rate is 90%. The corresponding number for a CPU is 99.9%. Because of this, additional chip die area is usually used for more cores and registers in GPUs, whereas it is used for more cache memory in CPUs [8].

The number of components in a GPU is significantly higher than in a CPU. For instance, Intel's latest CPU in their Core i7 series contains 6 cores [11], whereas Nvidia's high end GPU Geforce GTX 1080 contains 3584 cores [12]. The high core count makes the design of the interconnection network very important in GPUs.

1.4 Networks on Chips (NoCs)

A Network on Chip (NoC) is a network that connects the components within a chip. Components to be connected can be, for instance, processor cores, memory units, DSP cores, or custom hardware blocks [13]. There are several aspects that make NoCs different from other types of interconnection networks. First, the physical size of the network is smaller, because its wires are usually less than a centimeter long [3]. Second, there is a strong demand to keep the energy consumption low [2]. Third, the network can be fully defined at design time, in contrast to, for example, local area networks (LANs), which support ad-hoc changes [2]. The NoC is a fundamental part of a digital system, together with logic and memory units [1].

1.4.1 Motivation

One reason for developing efficient NoCs is the increasing bandwidth demand between the components of Systems on Chips (SoCs) [13]. According to Moore's law, the number of transistors that can fit on a silicon substrate grows exponentially [5]. This enables more components to be added, and with more components the connections between them become increasingly important. When the number of transistors grows very large, it becomes difficult to sync the clocks between all parts of the chip. This can be remedied by creating a system that has several clock domains, asynchronous to each other. Within each clock domain, the components are synchronous. This kind of system is called Globally Asynchronous Locally Synchronous (GALS). Enabling such systems is one important driving force behind NoC development [2].

Many SoCs are built using intellectual property designed by different companies. This makes standardized network interfaces important. Because the components get increas-

ingly complex and costly to develop, it is important that components can be used in several products and product versions. This is made more convenient by NoCs [13]. For instance, the high end version of a GPU may contain a larger number of identical shader cores than a low end version. The same shader core design could be used in both products [8]. NoCs also simplify the verification process, because they divide SoCs into separate units [13].

1.4.2 Development

Improved manufacturing technology makes smaller transistors possible. This, in turn, gives better transistor performance (enabling higher clock frequencies) [3]. An example of a transistor size used in modern smart phones, e.g. iPhone X, is 10nm [14]. Faster transistors increase the speed of processors and memory systems, which are built by transistors. The communication systems of SoCs do not, however, benefit in the same way. Signal propagation speed in wires is still the same, which has led to computation speed being limited by wire delay rather than gate delay [1]. Connecting the components of an SoC in a way that minimizes the signal propagation time between them therefore becomes increasingly important. For instance, this can be done by making the length of the wires between the components shorter. It is, however, not possible for all components of an SoC to be physically close to each other. An NoC can offer alternative ways to speed up communication, such as allowing multiple signals to traverse the network simultaneously.

Buses and dedicated point-to-point connections, which were the standard interconnections in SoCs until the late 90's, have in most cases been replaced by more sophisticated NoCs [3].

1.4.3 Protocol Stack

To provide abstraction and interoperability between different components, the NoC concept proposed by Benini and Micheli [2] is based on a network protocol stack. The stack resembles the TCP/IP stack used throughout the Internet, because their layer partitioning is similar [15]. The layers of the NoC protocol stack are visualized in Figure 1.2 below. The bottom layer in the stack is the physical layer, which consists of the physical wires. On top of the physical layer comes the data link layer, which provides error correction for losses in the physical layer. This can be implemented using error correction codes. Above the data link layer comes the network layer. The responsibility of the network layer is to route the network packets from the source to the correct destination. On top of the network layer comes the transport layer. It splits data into packets at the sending end, and reassembles the data at the receiving end. On top of the previous layers, which are all implemented in hardware, lies two software layers. First a system software layer, followed by an application software layer.

1.4.4 Topology and Routing

The topology of a network describes the pattern in which its nodes are connected. The choice of topology is important, because it is the aspect of the NoC that has the biggest impact on the implementation cost [3]. Topology is this project's primary focus.

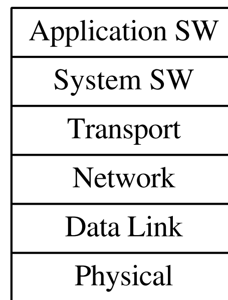

Figure 1.2: The figure shows the layers of the NoC protocol stack [2]. The four lowest layers are implemented in hardware, and the two uppermost in software.

Three common topologies used in NoCs are ring, mesh, and crossbar. They can easily be laid out on a planar silicon substrate, and they work with simple routing algorithms [3]. Routing algorithms decide which way a packet should take through the network to travel from source to destination. Below is a short description of the ring, mesh, torus, and crossbar topologies.

In a ring topology, each node is connected to two neighboring nodes. Together, all nodes and connections constitute a physical ring. To send data between two nodes that are not directly connected, data is relayed through the nodes between them. The most common routing algorithm sends the packet along the shortest path between source and destination nodes [3]. Figure 1.3 below contains an example of a ring topology.

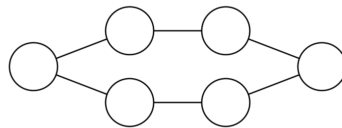

Figure 1.3: An example of a ring topology.

In a mesh topology, sometimes also referred to as a grid topology, all nodes are connected in a two-dimensional array. Figure 1.4 presents an example of a mesh topology. A routing algorithm commonly used in mesh networks is dimension-order routing, because it makes both implementation and deadlock avoidance simple [1]. First, it moves a packet in the x dimension of the array until the x coordinate of the packet and destination match. It then moves the packet in the y direction until the destination is reached. For each pair of source and destination node in a mesh topology, dimension-order routing always routes packets between them along the same route. This makes dimension-order routing in mesh topologies a deterministic routing algorithm [3].

A torus topology is similar to a mesh topology, with the addition of wrapping links that connect edge nodes on opposite sides. Because of the additional links, all nodes are connected to four other nodes. Figure 1.5 below contains an example of a torus topology.

In a crossbar topology, all peripheral nodes are connected to a centralized main switch called a crossbar. Hence, all data passes through the crossbar. Inside the crossbar, direct links between input and output ports are set up as needed. Figure 1.6 shows an example of a crossbar topology. Figure 1.7 shows an image of a crossbar.

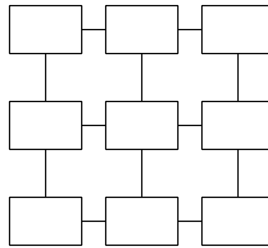

Figure 1.4: An example of a mesh topology.

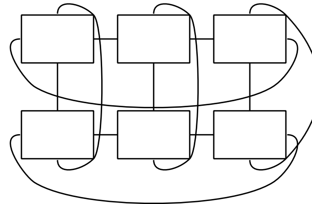

Figure 1.5: An example of a torus topology.

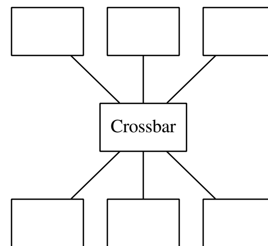

Figure 1.6: An example of a crossbar topology.

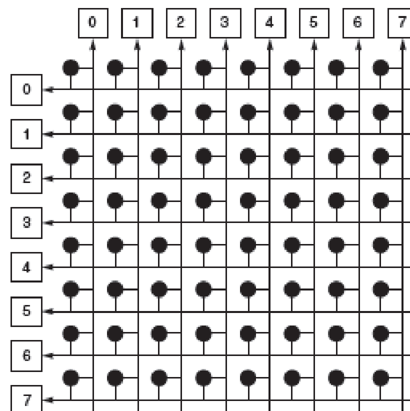

Figure 1.7: Conceptual image of a crossbar. The crossbar consists of 8 input lines and 8 output lines. The dots represent the logical switches connecting inputs to outputs. Reproduced from Hennessy et al. [3].

The choice of topology depends on the NoC requirements and budget. Each topology provides a different trade-off between speed, cost, and complexity.

Chapter 2

NoC Simulator Survey

We have surveyed the field of simulation and visualization tools for NoCs, to understand the state of the art of visualizing and specifying NoCs, and to gain inspiration on how we would do it. In Section 2.1, we will go through the papers we based our survey on. In Section 2.2, we discuss a network format called Netjson, which was a source of inspiration for our network language. In Section 2.3, we present an overview of the NoC simulators we have surveyed. Particular attention will be paid to how the simulators handle specification and visualization of networks. In Section 2.4, we present the conclusions we came to based on the survey.

2.1 Information Sources

Three early works describing NoCs are papers by Benini and Micheli [2], Kumar et al. [13], and Dally and Towles [16].

A survey of available simulators is presented by Achballah and Saoud [17]. A paper by Ali et al. discusses using Ns-2, a general network simulator, for NoC simulation [18]. Using Ns-2 for NoC simulation is also discussed in a paper by Sun et al. [19]. The simulator Darsim is discussed in a paper by Lis et al. [20]. The simulator Noxim is discussed in a paper by Catania et al. [21]. That paper also contains an overview of popular NoC simulation tools. This overview is of particular interest because it was written recently, in 2016.

The simulator VisualNoC is described in a paper by Wang et al. [22]. It is an NoC simulator that allows configuration through a graphical user interface. One part of this project is to create a visualization tool for NoCs, which makes VisualNoC particularly relevant as related work.

2.2 Netjson

Netjson is a project that has been a source of inspiration and ideas for the design of our network description language [23]. The project develops a JSON based format called Netjson, for definition of computer networks. JSON is a general markup language, readable by both humans and machines [24]. Netjson's primary goal is to provide a format that can be used to exchange network data between different applications. Netjson is designed to represent networks on layer 2 and 3 of the OSI stack [25]. One reason behind our choice to base our network description language on JSON, was that we knew Netjson successfully used JSON for a similar purpose.

Netjson allows much more detailed definition of network parameters than we need in our project. Still, we used ideas from its topology definition part. When a topology is defined in Netjson, nodes are first created as individual JSON objects. Each node is identified by an id attribute. The links connecting the nodes are defined in a separate array called links. For each such link, the ids of the source and target nodes are defined. We have used those concepts in our network description language. An example of a link defined in Netjson is showed in Listing 2.1.

Listing 2.1: Netjson code defining a link between two nodes. Reproduced from Netjson's homepage [26].

```
1  "links": [
2          {
3              "source": "172.16.40.24",
4              "target": "172.16.40.60",
5              "cost": 1.000,
6              "cost_text": "1020  bit/s",
7              "properties": {
8                  "lq": 1.000,
9                  "nlq": 0.497
10             }
11         }
12     ]
```

We considered using Netjson as it is, instead of designing our own language. This would have made it possible to use existing visualization tools. netjsongraph.js is a visualization tool for Netjson which is presented on the Netjson homepage [27, 28]. Using a standard would also have been beneficial if data was to be exchanged with third party software, even though no such exchange is present today. Netjson is, however, unnecessarily complex for our needs. It is also a limitation that it is not designed for NoCs. This made us decide to borrow some ideas from Netjson, rather than using Netjson as it is.

2.3 NoC Simulators

Possible design options for an NoC make up a very large design space, in which efficient design space exploration is crucial. In this process, simulation is a valuable tool. Important evaluation criteria for designers are power consumption, latency, throughput, and reliability [17]. There are several available NoC simulators, and below is a description of some of them. We have chosen to include the non-commercial simulators mentioned by Catania

et al. in their simulator overview (Atlas, BookSim, Darsim, and Noxim) [21]. In addition to them, we include VisualNoC, Ns-2, and Garnet. VisualNoC is included because of its visualization capabilities and its GUI for topology configuration. Ns-2 is included because it is a general computer network simulator that can be used for NoC simulations. Garnet is included because it is an NoC simulator that is part of the extensively used gem5 full system simulator. There are several NoC simulators that we do not cover in our survey, but we think the simulators we have chosen to include will give an overview of the field.

2.3.1 Atlas

Atlas is not only an NoC simulator, it is also an NoC generator. It allows the user to define the desired network parameters using a GUI, and outputs a description of the NoC in VHDL. The network can then be simulated using the simulator ModelSim, which is a part of Atlas. Atlas supports power simulations. The supported topologies are mesh and torus. Atlas is implemented in Java, and its source code is open [29, 30].

We consider only being able to chose from a predefined set of topologies to be too limited for our project. We did consider implementing a GUI in our project, but after discussions with Arm engineers we concluded that it would take too much time and effort.

2.3.2 BookSim

BookSim is an NoC simulator developed as accompanying material for Dally and Towles's book on the topic [1, 31]. It is implemented in C++, and the source code is available online [32]. Topologies can be chosen from a predefined set, and it is also possible to implement arbitrary custom topologies. Either way, all configuration is done in a text file, which is parsed by the simulator when it starts.

The definition format for custom topologies is quite minimalistic, and lets the user define which routers connect to which routers and nodes. For each connection, a latency can be defined. The two types of components are routers and nodes. Listing 2.2 shows a custom network definition in Booksim.

Listing 2.2: Custom network definition in Booksim. Reproduced from the Booksim source code repository [33].

```
1 router 0 node 0 node 1 node 2 router 1
2 router 1 node 3 node 4 node 5
3 router 2 node 6 node 7 node 8 router 1 router 0
```

By offering custom topology definition, and configuration through a parsed text file, Booksim implements two of the main features we are adding to the Arm GPU model. Booksim was therefore a source of inspiration. We did, however, consider the format for defining the custom networks to be too minimalistic, partly because it supports only two types of components. Therefore, we chose not to use the network specification format of Booksim.

2.3.3 Darsim

Darsim is an NoC simulator that is designed to benefit from running on multicore systems [20]. When running on a processor with four cores, it runs 3.5 times faster than on a single core. With eight cores, it is almost five times faster than with one. Darsim can simulate any network topology. Darsim is configured using configuration files, which are preprocessed by a separate program into an intermediate format, which then is provided as input to Darsim. To simplify the creation of the configuration files, Darsim is delivered with scripts that generate standard configuration files. Darsim is also referred to as Hornet. The open source code and a manual describing how to use Darsim is available online [34]. Listing 2.3 presents an excerpt from a Darsim configuration file.

Listing 2.3: Excerpt from a Darsim configuration file. Reproduced from the Darsim/Hornet manual [34].

```
1  [geometry]
2  height = 8
3  width = 8
4
5  [routing]
6  node = weighted
7  queue = set
8  one queue per flow = false
9  one flow per queue = false
10
11 [node]
12 queue size = 8
13
14 [bandwidth]
15 cpu = 16/1
16 net = 16
17 north = 1/1
18 east = 1/1
19 south = 1/1
20 west = 1/1
21
22 [queues]
23 cpu = 0 1
24 net = 8 9
25 north = 16 18
26 east = 28 30
27 south = 20 22
28 west = 24 26
29
30 [core]
31 default = injector
32
33 [flows]
34 # flow 00 -> 01 using xy routing
35 0x000100@->0x00 = 0,1
36 0x000100@0x00->0x00 = 0x01@1:24,26
37 0x000100@0x00->0x01 = 0x01@1:8,9
38 # flow 00 -> 02 using xy routing
39 0x000200@->0x00 = 0,1
40 0x000200@0x00->0x00 = 0x01@1:24,26
```

```
41          0x000200@0x00->0x01 = 0x02@1:24,26
42          0x000200@0x01->0x02 = 0x02@1:8,9
43          # flow 00 -> 03 using xy routing
44          ... ...
```

2.3.4 Noxim

Noxim is an NoC simulator that supports arbitrary network topologies [21]. It is developed in SystemC, and configured using files written in the markup language YAML. Noxim supports simulation of wireless NoCs. Listing 2.4 presents YAML code defining the network showed in Figure 2.1. Its source code is open and available online [35]. Large changes were made to Noxim in 2015. The paper describing Noxim does not mention visualization capabilities.

Noxim provides another example of a network simulator in which the network can be configured using a markup language. The network description file is read when the simulation starts, which lets the user redefine the NoC without editing the source code or recompiling. Our extension of the Arm GPU model is, in this particular aspect, similar to Noxim.

Listing 2.4: Definition of the network shown in Figure 2.1 in YAML for Noxim. Reproduced from Catania et al. [36].

```
1  mesh_dim_x: 4
2  mesh_dim_y: 4
3
4  Hubs:
5      0:
6          rxChannels: [0,2]
7          txChannels: [0,2]
8          attachedNodes: [0,1,4,5]
9      1:
10         rxChannels: [0,1]
11         txChannels: [0,1]
12         attachedNodes: [2,3,6,7]
13     2:
14         rxChannels: [2,3]
15         txChannels: [2,3]
16         attachedNodes: [8,9,12,13]
17     3:
18         rxChannels: [1,3]
19         txChannels: [1,3]
20         attachedNodes: [10,11,14,15]
```

2.3.5 VisualNoC

VisualNoC is an NoC simulator which focuses on network visualization [22]. The user can define the network topology using either an XML configuration file, or a GUI. 2D mesh and 2D torus topologies are supported, as well as irregular topologies. VisualNoC provides visualization of the topology, together with visualization of the packets as they flow

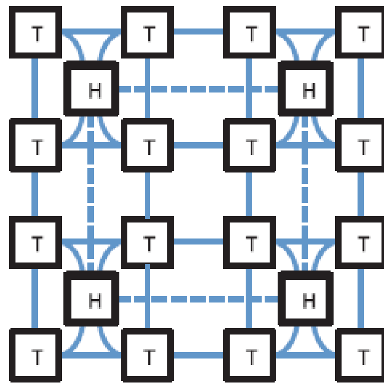

Figure 2.1: Network topology defined by the YAML code in Listing 2.4. Reproduced from Catania et al. [36].

through the network. This simulator is interesting because it supports both visualization and irregular topologies. Figure 2.2 presents a screenshot from the VisualNoC GUI.

VisualNoC served as an example of an available network simulator that lets the user define the topology in a markup language.

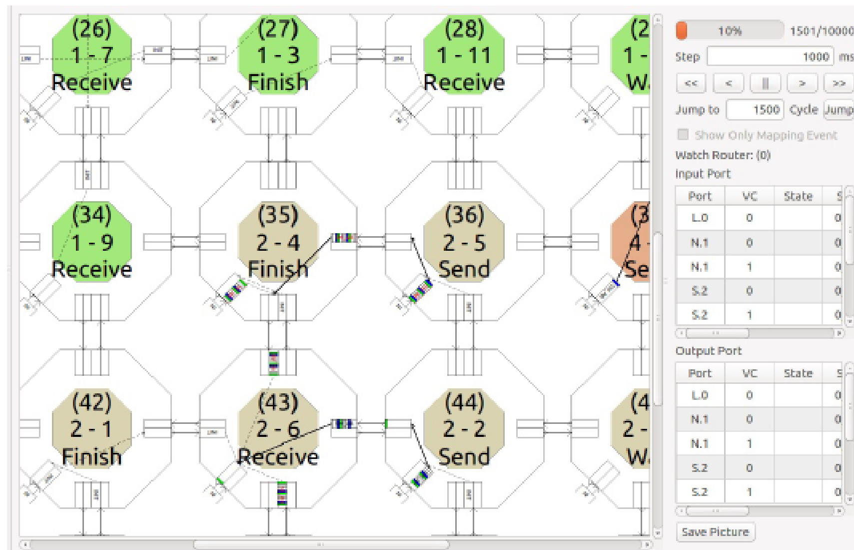

Figure 2.2: Screenshot from the VisualNoC GUI. Reproduced from Wang et al. [22].

2.3.6 Ns-2

Ns-2 is developed for simulations of networks of computers rather than NoCs. It uses the TCP/IP protocol stack in its simulations. It has, however, often been used to simulate NoCs [17]. Several works describe using Ns-2 for NoC simulation [18, 19, 37]. By configuring the network parameters to be similar to those in NoCs, Ali et al. conclude that it is feasible

to simulate NoCs on a behavioral level using Ns-2 [18]. However, there is no support for simulation of power consumption. Sun et al. state that the NoC simulations in Ns-2 would be more accurate if NoC specific routing algorithms and protocols were implemented [19].

Ns-2 is an open source project [18]. The back-end of Ns-2 is implemented in C++. The programming language OTcl is used to configure simulation parameters such as the network topology [19]. The simulation can be visualized using the tool Network AniMator (NAM) [18].

We chose not to use the Ns-2 approach for our network description, because it requires hard coding topologies in OTcl. Moving away from hard coded topologies is one of the purposes of this project.

2.3.7 Garnet and gem5

gem5 is a full system computer architecture simulator [7]. It is developed by a community of both academic and industrial partners, including Arm. The gem5 simulator is available under a BSD license, which makes it usable for many different parties. All configuration is done in Python, and the simulation is implemented in C++. gem5 is flexible, and offers several modes for CPU and memory system simulation.

There are two available memory system simulators in gem5, called Simple and Garnet [38, 39]. In both, the topology is set up using Python programming, and can be configured to model arbitrary topologies. Garnet is the more detailed and accurate model of the two. In addition to latency and bandwidth configuration, which both support, Garnet also simulates the routers in detail. A paper by Binkert et al. states specifically that Garnet is suitable for NoC simulation [7]. Garnet can be used in full system simulations, but can also be used for network only simulations. The power consumption simulator Orion is included as part of Garnet, which adds energy estimations to the simulations [40].

We chose not to use the Python approach of gem5, because in it the topologies are hard coded. Hard coding the topologies was already possible in the Arm GPU model before this project, and we are trying to move away from that solution.

2.4 Conclusion

This survey found three ways to define networks in NoC simulators. The network can either be hard coded, defined using a GUI, or defined in a text file using a markup language. One purpose of this project was to move away from hard coded topologies, to speed up explorations. This meant that the two remaining options were using either a GUI, or markup language description files. Markup language files is an approach used by several of the surveyed simulators (BookSim, Noxim, and VisualNoC). Therefore, we knew it was a feasible approach. We also expected implementing support for parsing of configuration files to be easier than building a GUI. This made us decide to implement a markup language based solution for this project.

There is no standard approach to topology visualization among the surveyed simulators. We considered the solutions in VisualNoC and Ns-2, which animates the flow of packets through the network, to be too advanced and complicated for our use case. We

wanted a simplistic solution that only visualizes the topology, but did not find any such solution in our survey.

Chapter 3
Approach

In this section, we go through the different tasks the project was divided into. They include a background survey, criteria definition, a design phase, software implementation, and verification.

3.1 Background Survey

As the first part of the project, we did a background survey. The purpose of the survey was to acquire the background knowledge needed for the project. The background survey was divided into three parts, which we describe below.

3.1.1 GPU and NoC Theory

In the first part of the background survey, we read about GPUs and NoCs. When reading about GPUs, we found the books by Akenine-Möller et al. [9], and Patterson and Hennesy [8], particularly useful. When reading about NoCs, we found that a book by Dally and Towles [1], and a paper by Benini and Micheli [2], gave good insights into the field. We wrote Section 1.3 and 1.4, to provide the reader with overviews of GPUs and NoCs. In those sections, more references to works about GPUs and NoCs can be found. A basic understanding of GPUs is useful to understand the motives behind the design of the GPU interconnection network, and important to be able to place this project in its context. Knowledge about NoCs, and in particular what characterizes them, was important for us to have during the design of the network format.

3.1.2 Network Description in NoC Simulators

In the second part of the background survey, we studied how others have solved problems similar to ours. To find out how other NoC simulators allow their users to define NoC

parameters such as topology, we read several research papers. Two papers, one by Achballah and Saoud [17], and one by Catania et al. [21], were particularly useful, because they presented overviews of available simulators. The results from this part of the survey are presented in Chapter 2 above.

In this part of the project, we also looked at how the interconnection network is currently configured and implemented in the Arm GPU model. Several topologies are currently hard coded in the model source code. By passing a command line flag to the model, one of those topologies can be chosen.

3.1.3 Markup Languages

In the third part of the background survey, we surveyed different markup languages. We did this to see if any of them were suitable for the NoC specification in our project. We looked at the following five markup languages: XML [41], JSON [24], YAML [42], DOT [43], and GML [44]. The first three (XML, JSON, and YAML), are general markup languages. They are all both human and machine readable. Because they are general, they are flexible, and adding concepts such as bundles of components is straightforward. They do, however, not directly support visualization. The last two, DOT and GML, are graph description languages. The advantage of those is that they can be visualized using existing software. Their disadvantage is that they are less flexible.

To evaluate how well suited the different markup languages were for NoC definition, we implemented the same grid based NoC containing 24 switches in each language. We considered how flexible and extensible each option was, as well as whether they could be directly visualized or not. Appendix C shows the implementations.

3.2 Definition of Criteria

At an early stage in the project, we defined a set of distinct criteria the network description language and the implementation should meet. This was done through discussions with both our supervisors, to align the criteria with the demands from the Arm GPU model and the university. The criteria were important to have when we made the major design decisions, and can be seen as the software requirements of the project. Furthermore, they were used during the evaluation phase to decide whether our implementation was successful. The defined criteria were that our solution should do the following:

- Support visualization
- Make possible network configuration without any knowledge of the internal implementation of the Arm GPU model
- Support implementation of all topologies possible to implement in the Arm GPU model
- Support four-port switches
- Support crossbars

- Support delays defined per link
- Support prioritizing of input signals to a switch

The features mentioned in the last four criteria (four-port switches, crossbars, delays per link, and input signal prioritizing), are all present in the Arm GPU model, and hence necessary to support in order to support all topologies of the model. Four-port switches are commonly used in two-dimensional topologies.

3.3 Network Description Language

Appendix A presents a manual describing our network description language. Appendix B presents two examples of topology descriptions. Below follows a brief overview of the network description language.

The network description language we designed is based on the general markup language JSON. The components of the network are categorized in arrays, one for each category. In each such array, each component of that category, for instance each switch or each shader core, is defined as a JSON object. The information stored for each component varies depending on the type of component. For most of the types, the only mandatory information is an id string. The id strings are used to identify the components when connecting them. They must therefore be unique. Listing 3.1 shows an example of a switch array, in which two switches are defined.

Listing 3.1: Definition of two switches.

```
1     "switches": [
2         {"id": "s04","x": 0,"y": 4},
3         {"id": "s14","x": 1,"y": 4}
4     ]
```

In a separate array, each connection is specified as a JSON object. For every connection, it is mandatory to define a source node and a target node. Source port, target port, and delay can be defined per connection as optional attributes. Listing 3.2 shows an example of a connection array, in which a connection between the two switches in Listing 3.1 is defined.

Listing 3.2: Definition of a connection.

```
1 "links": [
2         {"source_node": "s04","target_node": "s14","
             source_port": "e","target_port": "w"}
3 ]
```

In addition to the possibility to add each component individually, it is possible to define so called shader core stacks. A shader core stack is a linear array of switches, with one shader core connected to each of them. Figure 3.1 shows an example of a shader core stack. Shader core stacks appear frequently in the hard coded topologies in the Arm GPU model, and therefore we wanted to provide a convenient and compact way to define them. Shader core stacks are defined in an optional array in the root JSON object. In this array, each stack is represented by a JSON object. The object contains information about id, via which switch the stack connects to the rest of the network, and in which direction the

stack points. Listing 3.3 shows an example of a shader core stack definition. The length of each stack is not defined here. Instead, it is calculated based on the number of shader cores used in the simulation run. If shader core stacks are defined, it is not possible to add shader cores to the network individually.

Listing 3.3: Definition of a shader core stack.

```
1 "core_stacks": [
2         {"id": "stack0", "base": "s04", "direction": "s"}
3 ]
```

For more details about the network description language, refer to the manual in Appendix A.

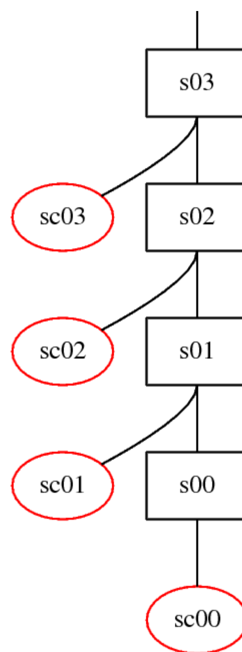

Figure 3.1: A shader core stack of length four, pointing south. The boxes represent switches and the ellipses represent shader cores.

3.4 Software Implementation

This section goes through the software we have implemented. The software consists of two parts: A visualization script and an extension of the Arm GPU model.

3.4.1 Visualization Script

The visualization script is implemented in Python. It parses a network description file written in our network description language. This is done using the standard JSON parsing library in Python. Based on the parsed information, the script builds an internal representation of the network using classes representing switches, crossbars, and other components. The script analyses the network, and removes switches that are only connected to one other

switch and nothing else. Such switches are referred to as dead ends, because no information passes through them. The script also detects switches that can be bypassed. A switch can be bypassed if it is connected to two other switches on opposite sides, and nothing else. The switch must also be explicitly marked as possible to bypass in the configuration file.

Once the script is done with the pruning of dead end switches and detection of switches to bypass, it traverses the network and creates a representation of it in the graph description language DOT. The DOT representation is written to a file, which can then be used by the program Dot [45] to generate an image of the network. Figure 3.2 presents a graph describing the visualization workflow. In Appendix B, visualizations of two network topologies can be seen, together with the network description files defining them.

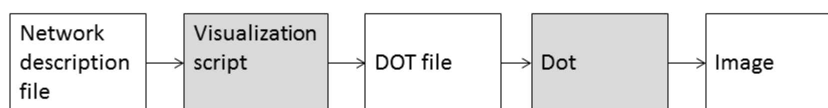

Figure 3.2: The workflow for generation of topology images. The shaded boxes represent programs, and the white boxes represent files.

3.4.2 Extension of the Arm GPU Model

In its current state, the available topologies of the GPU interconnection network are hard coded in the model. There are several topologies available, and which one to use is chosen when simulation starts.

In the model, the setup of the network is done in a class called GPUNetwork. This class creates instances of all the objects representing the components of the network, and connects them according to the chosen topology.

In this project, we have implemented additional functionality. This functionality parses a configuration file written in our network description format, and sets up the network accordingly. This has been added as an option in the GPUNetwork class. This means that the user can choose whether to use one of the existing hard coded topologies, or to build the network according to his specification in a network configuration file.

To add this functionality, we implemented several classes to handle the JSON parsing, the network setup, and the network analysis. The names of those classes, and the relationship between them, can be seen in Figure 3.3. We also added code in the GPUNetwork class.

When the execution flow of the model reaches the GPUNetwork class, and the user has chosen to parse a network configuration file, the system creates data structures to store an intermediate representation of the network. Before it proceeds, it calls a function in GPUNetworkParser. Pointers to the aforementioned data structures are passed as function parameters to GPUNetworkParser, which takes care of populating the data structures according to the information in the description file. Once the execution flow returns to

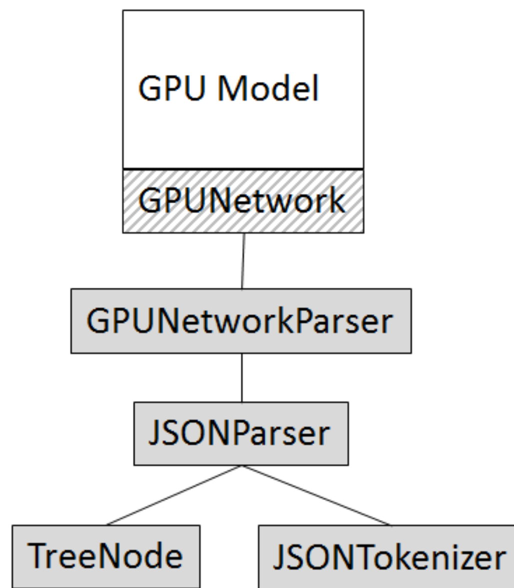

Figure 3.3: Relationship between the original GPU model, and the classes we have added to it. The shaded classes were added by us. The striped class was edited by us, but already existed before this project.

GPUNetwork, it iterates over the data structures and sets up the actual network by creating the simulation objects.

Once this is done, the simulation continues in the exact same way it would if a hard coded topology was used. Because the extension only affects the setup phase of the simulations, which consumes only a small fraction of the total simulation time, the addition of the extension has a very small impact on the total simulation time. For one of the networks, we measured the time increment to be less than one percent when using a simple graphics benchmark. Figure 3.4 shows the stages the network description information goes through when a parsed network is created.

The GPUNetworkParser class populates the data structures created in GPUNetwork with an intermediate representation of the network. The first step in this process is to invoke a function in the JSONParser class, which parses the JSON file and returns a pointer to the root of the JSON tree structure. During the JSON parsing, the JSONParser class calls JSONTokenizer, which preprocesses the JSON data. JSONParser then uses the preprocessed data to build a tree structure consisting of TreeNode nodes.

Once the JSON parsing is done, GPU Network Parser iterates over the tree, parsing the content of each JSON array into the data structures created in GPU Network. Some meta information, such as the maximum and minimum number of supported shader cores and L2 cache units, is also parsed. If shader core stacks are present in the topology, their length is calculated. The calculation is based on the information about the shader core stack configuration, and the number of shader cores the current simulation will use.

After all other network information has been parsed, the creation of the shader core

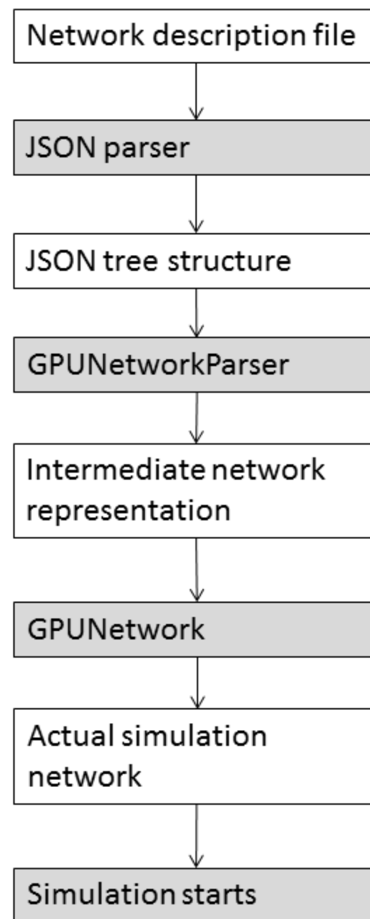

Figure 3.4: The stages the network description information goes through when a network is created. The white boxes represent the different stages of the network information, and the gray represent the parts of the GPU model that process the network data.

stacks take place. When GPU Network Parser iterates over the JSON array specifying the stacks, it adds switches, shader cores, and connections to the intermediate network representation. This means that the GPU Network class is not aware of the shader core stack concept; from its perspective a stack is just a set of switches and shader cores.

Once all information from the configuration file has been extracted from the JSON file, GPUNetworkParser performs some optimizations. Just like the visualization script, it removes all switches that are only connected to one other switch. No data passes through them, so they are considered to be dead ends. Even though removing them does not impact the results of the simulation, the removal speeds it up. The speed up occurs because the removal simplifies the simulation. This step is repeated until no dead end switches are detected, because removing a dead end switch might turn the switch next to it into a dead end.

After the pruning of unused switches, GPU Network Parser detects all switches marked as candidates for bypassing in the configuration file, that are also connected to nothing but two other switches on opposite sides. Such switches are marked as definitely possible

to bypass. This marking is used by GPU Network when the actual simulation objects representing the switches are created. Switches marked to be bypassed never get created, and the switches on its sides get connected directly to each other instead.

The extension also allows the user to choose to use fewer shader cores and/or L2 cache units than defined in the description file. If the user chooses to use n cores, only the first n cores in the description file will be included. This, together with the removal of dead end switches, and the shader core stack concept, makes the description files flexible. The same description file can be used for several numbers of shader cores and/or cache units, reducing the number of description files to be written and maintained. We got this idea from the hard coded topologies in the model, for which similar flexibility was already present.

3.5 Verification

Throughout the project, we worked actively with verification of the implemented software. For the Python visualization script, the verification consisted of us testing to visualize different topologies, and then manually inspecting the produced visualizations.

For the extension of the GPU model, we have performed three types of verification during the implementation. Those three types of verification are unit tests, log file comparison tests, and clock cycle count comparison tests.

In each of the unit tests, two networks are built. One is built using a hard coded topology, and one using our extension of the GPU model. Every data structure in each network is then compared to its counterpart in the other network, and the test passes if no differences are detected. No simulation is actually run, which makes the unit tests relatively fast. To verify that we did not introduce any memory leaks in our code, we ran the unit tests under Valgrind [46].

The log file comparison tests are similar to the unit tests in that they also build two networks each. One is built using a hard coded topology, and one using our model extension. What sets the log comparison tests apart from the unit tests, is that full simulations are run. One simulation is carried out for each of the two networks, and the resulting simulation log files are then compared. The log files contain performance data for every component in the GPU. This means that even very subtle mismatches in the configurations would be detected by the comparison. The tests are successful if there are no differences between the log files. The log file comparison tests are implemented as Bash scripts. Those tests involve running two full simulations, and therefore take significantly more time to run than the unit tests. To make those tests as fast as possible, the graphics benchmarks used in the simulations are simple.

The clock cycle count comparison tests are similar to the log file comparison tests, because they also compare the results of full simulations. Instead of comparing the entire log files, only the number of used clock cycles is compared. Those tests do, however, use more varied and complex graphics benchmarks.

For both the log file and the clock cycle count comparison tests, it is likely that even small differences in the configurations result in a changed clock cycle count. The high clock cycle count in the clock cycle count comparison tests makes it more likely that even a small mismatch would result in a changed clock cycle count. Therefore, we considered

it enough to only compare the clock cycle count when more complex benchmarks were used.

Our extensions to the model, as well as the visualization script, were run through the tests of Arm's continuous integration system. Those tests detected errors and bad coding practices introduced in our code, which we corrected. Hence, those tests helped improve the code quality.

It was useful to have this combination of verification tests, with varying speed and level of detail. Thanks to the speed of the unit tests, we could run them often during development. Thanks to the level of detail in the other tests, we could make sure the generated topologies were identical.

Chapter 4
Evaluation

To determine whether we had reached the project goals, we evaluated our work quantitatively as well as qualitatively. For the quantitative evaluation, we used a set of metrics such as fulfilled criteria, lines of code used to define the NoCs, and the flexibility of the description files. For the qualitative evaluation, we compared the complexity of defining the networks in the hard coded way and in our network description language. We also discussed our solution with our supervisor at Arm, who is an experienced modeling engineer. The purpose of the evaluation was to investigate whether the project reached the stated goal of making the definition of the NoC easier, faster, and less error prone. Below follows a walk through of the methods used for the evaluation, together with the acquired results.

4.1 Fulfilled Criteria

As part of the quantitative evaluation, we checked how many of the criteria defined in Section 3.2 that were reached in our implementation. All criteria were reached by our implementation.

4.2 Lines of Code

The number of lines of code needed to define the interconnection network was used as a quantitative measurement. For each hard coded topology in the model source code, we implemented an identical topology in our network specification language. By using the unit tests and log file comparison tests, we could assert that the networks were indeed identical. The number of lines of code/description language used to specify the two networks was then compared.

In Table 4.1, the results of the line count comparison is presented. On average, the number of lines needed to define the previously hard coded topologies decreased by 48%. Topology 10 stands out, because it requires more than three times as much code when defined in a description file. The reason behind this is discussed in Section 5.3.

Table 4.1: Number of lines of code used to define the ten hard coded topologies of the model.

Topology	Hard coded	Description file	Ratio
Topology 1	41	47	1.15
Topology 2	124	55	0.44
Topology 3	124	56	0.45
Topology 4	124	55	0.44
Topology 5	124	55	0.44
Topology 6	124	55	0.44
Topology 7	124	56	0.45
Topology 8	210	80	0.38
Topology 9	210	80	0.38
Topology 10	31	108	3.48
Average	123.6	64.7	0.52

4.3 Flexibility of Description Files

In the Arm GPU model, it is possible to select how many shader cores and L2 cache units to add to the topology. The hard coded topologies include logic that defines which components should be included when the defined number is lower than the maximum number supported by the topology. For the description language based topologies, the desired number of components are added, starting from the top of the respective component array.

We wanted to see how much flexibility our solution allows in terms of support for different numbers of shader cores and L2 cache units in a single file. To measure this, we checked for each hard coded topology how many description files were needed to cover all supported numbers of shader cores and caches. Ideally, it should be enough with one description file per topology.

Table 4.2 below displays the number of description files needed to represent all supported numbers of shader cores and L2 cache units for each hard coded topology. The topology that stands out is Topology 9, and the reasons behind this are discussed in Section 5.4.

4.4 Time to Define Network

The time it takes to define networks using our network specification language was used as a quantitative measurement. Through discussions with our supervisor at Arm, we got an estimate of how much time was spent implementing the different hard coded topologies.

Table 4.2: Number of configuration files needed to represent all supported shader core and L2 cache numbers for each topology.

	Number of needed configuration files
Topology 1	3
Topology 2	1
Topology 3	1
Topology 4	1
Topology 5	1
Topology 6	1
Topology 7	1
Topology 8	1
Topology 9	7
Topology 10	1

According to him, writing the code for a new topology previously took several hours, excluding the test and debug time [47].

We implemented four topologies in our network description language, that were previously not present in the Arm GPU model. We did this to get practical experience of what it is like using the language, as well as an impression of how much time it takes to implement new topologies using it. Writing each description file was largely a matter of copying and pasting configuration elements between files. For the first two topologies, we did not write down the exact time it took us. We do, however, estimate that it took us less than an hour per file. For the last two topologies we did write down the time it took us; 18 and 16 minutes respectively.

Implementing a new topology does, however, include a large amount of testing. Testing is primarily done by running test simulations, and comparing the resulting metrics with metrics from previous simulation runs. This process has not been changed in this project, and still takes the same amount of time as it did before. Running those simulations takes several hours, even on a powerful computer cluster.

4.5 Comparison with Source Code

To evaluate how much our network description language simplifies the network configuration, we compared structures implemented in source code and the description language qualitatively.

To give an idea of how the network definition has changed from the hard coded to the description language solution, we present pseudo code together with description language describing the same structures. The pseudo code is similar to the source code in the model. For legal reasons we cannot present the actual source code in the report.

In the first example, shown in Listing 4.1 and 4.2, we show how a linear array of switches is set up, and how an MMU gets connected to it.

Listing 4.1: Setup of switches and MMU (in pseudo code)

```
1 // List to store switches in
```

```
2  switch_list
3
4  // Calculate number of switches
5  num_switches = max(num_cores, num_l2_caches + 2)
6
7  // Create and connect switches
8  for (i = 0; i < num_switches; i++)
9      switch_p = new Switch(switch_id(i, 0))
10     if (i > 0)
11         switch_p.connect(switch_list.last(), WEST)
12     switch_list.add(switch_p)
13
14 // Connect MMU
15 mmu_switch = switch_list.first()
16 mmu_switch.connect(MMU, WEST)
```

Listing 4.2: Setup of switches and MMU (in description language)

```
1  "switches": [
2      {"id": "s00","label": "switch00","x": 0,"y": 0},
3      {"id": "s10","label": "switch10","x": 1,"y": 0},
4      {"id": "s20","label": "switch20","x": 2,"y": 0},
5      {"id": "s30","label": "switch30","x": 3,"y": 0}
6  ],
7  "mmu": [
8      {"id": "mmu","label": "mmu"}
9  ],
10 "links": [
11     {"source_node": "s00","target_node": "s10","source_port
          ": "e","target_port": "w"},
12     {"source_node": "s10","target_node": "s20","source_port
          ": "e","target_port": "w"},
13     {"source_node": "s20","target_node": "s30","source_port
          ": "e","target_port": "w"},
14     {"source_node": "s00","target_node": "mmu","source_port
          ": "w"}
15 ]
```

In the second example, shown in Listing 4.3 and 4.4, we show how L2 cache units are connected to certain switches. Please note that the pseudo code and the description language code only describe the same structure if there are two switches to which L2 cache units are connected. In cases like this, the transition to description language code improves the readability of the network definition, at the cost of reduced flexibility.

Listing 4.3: Connection of L2 caches to switches (in pseudo code)

```
1  // Calculate where to place first L2 cache.
2  l2_column_offset = (num_l2_caches > num_columns) ? 0 : 1
3
4  // Calculate number of L2 caches per middle switch.
5  num_l2_per_switch = (num_l2_caches <= num_columns / 2) ? 1
       : 2
6  num_connected_l2s = 0
7
8  // Connect L2 caches to switches.
```

```
9  for (j = 0; j < num_l2_per_switch; j++)
10     for (i = l2_column_offset;
11     i <= num_l2_caches / num_l2_per_switch &&
          num_connected_l2s < num_l2_caches; i++)
12         l2_id = num_connected_l2s++
13         direction = NORTH + j
14         switch_list[i].connect(l2_list[l2_id], direction)
15         if (num_connected_l2s == num_l2_caches)
16             break
```

Listing 4.4: Connection of L2 caches to switches (in description language)

```
1  "l2_caches": [
2      {"id": "l2_0","label": "l2_0"},
3      {"id": "l2_1","label": "l2_1"},
4      {"id": "l2_2","label": "l2_2"},
5      {"id": "l2_3","label": "l2_3"}
6  ],
7  "links": [
8      {"source_node": "s14","target_node": "l2_0","
          source_port": "n"},
9      {"source_node": "s14","target_node": "l2_2","
          source_port": "e"},
10     {"source_node": "s24","target_node": "l2_1","
          source_port": "n"},
11     {"source_node": "s24","target_node": "l2_3","
          source_port": "e"}
12 ]
```

In the third example, shown in Listing 4.5 and 4.6, we show how shader core stacks are created.

Listing 4.5: Creation of shader core stacks (in pseudo code)

```
1  // List to store pointers to the switches of the previous
      row.
2  previous_row_switches
3
4  // Fill list with the switches of the top row.
5  for (i = 0; i < num_columns; i++)
6      previous_row_switches.add(switch_list[i])
7
8  // Create and connect shader cores and switches one row at
      a time.
9  for (j = 0; j < num_rows; j++)
10     for (i = 0; i < num_columns; i++)
11         if (j * num_columns + i >= num_cores)
12             break
13         core_delay = 4
14         if (j == 0)
15             down_delay = 3
16
17         // Create and connect switch.
18         switch_list.add(new Switch(switch_id(i, top_row - j
               - 1)))
```

```
19          previous_row_switches[i].connect(switch_list.last()
               , SOUTH, down_delay)
20          previous_row[i] = switch_list.last()
21
22          // Create and connect shader cores.
23          core_id = shader_cores_list.size()
24          switch_list.last().connect(new Core(core_id), WEST,
                core_delay)
```

Listing 4.6: Creation of shader core stacks (in description language)

```
1  "core_stacks": [
2      {"id": "stack0", "base": "s04", "direction": "s"},
3      {"id": "stack1", "base": "s14", "direction": "s"},
4      {"id": "stack2", "base": "s24", "direction": "s"},
5      {"id": "stack3", "base": "s34", "direction": "s"}
6  ],
7  "core_stack_config": {
8      "switch_delay": 0,
9      "core_delay": 4,
10     "root_delay": 3,
11     "balanced_stacks": "true",
12     "max_length": 4
13 }
```

4.6 Implementation of New Topologies

As mentioned in Section 4.4, we implemented four topologies, which were not previously available in the Arm GPU model, using our network description language. The purpose of this was to test how convenient and time consuming it is to use the language. We call the implemented topologies mesh, hash tag, narrow, and line.

Even though performance was not the primary focus when implementing the topologies, we still wanted to evaluate how well they performed. Therefore, we ran simulations with them. Our supervisor at Arm helped us choose four metrics from the generated simulation logs that would give an idea of how well the topologies performed [47]. Those metrics were the total number of clock cycles, the L2 cache message stall time, the L1 cache read latency, and the L1 cache write latency. The metrics were measured in clock cycles. The total clock cycle count gives an idea of the overall performance. The three other metrics relate closely to the memory system, which is highly affected by the network topology. Visualizations of the topologies are presented in Section 4.8.

Table 4.3 shows the results of the simulations, and has been included for completeness. For each topology, ten simulations using different graphics benchmarks were run. For mesh and hash tag, the results are mean values based on all ten simulation runs. For the line topology, three out of ten simulations timed out due to network congestion. Therefore, the values for the line topology are mean values of the seven simulations that completed. For the narrow topology, one simulation timed out. The values for the narrow topology are therefore the means from nine simulations. In all simulations, 16 shader cores and four

L2 cache units were used. The results are normalized, relative to the default topology used for 16 cores.

Table 4.3: Simulation results for the implemented topologies. GPU active refers to the total clock cycle count. All results in the table are normalized against the default topology. For the topologies marked with stars, only the average from the successful simulation runs are shown.

Topology	Default	Mesh	Hash tag	Line*	Narrow*
GPU active	1	0.96	0.95	0.98	1.01
L2 message stall	1	0.56	0.88	2.96	1.15
L1 read latency	1	0.90	0.90	0.99	1.00
L1 write latency	1	0.77	0.77	0.90	0.97

4.7 Discussion with our Supervisor

To qualitatively evaluate the implemented solution, we discussed both the new and the previous solution with our supervisor at Arm, an engineer working with the simulation of the GPU interconnection network. The comments from the discussion are presented in the evaluation and discussion sections they are related to.

4.8 Visualization

The quality of the network visualizations was evaluated qualitatively. The result is based on our personal opinions. During the project we implemented and visualized several topologies, and based on those visualizations we fine tuned the visualization script to produce clear visualizations. We discuss the visualizations in Section 5.6.

In this section, we present four visualized topologies. One more visualization can be seen in Appendix B. The visualizations are included to give the reader an idea of what the produced visualizations look like. Figure 4.1 shows a mesh topology. Figure 4.2 shows a topology we have chosen to call a hash tag topology. We chose that name because the layout of the switches resembles a hash tag. Figure 4.3 shows the line topology. Figure 4.4 shows the narrow topology.

Please note that the topologies in the Arm GPU model do not necessarily correspond to the actual topologies in the Arm GPU hardware.

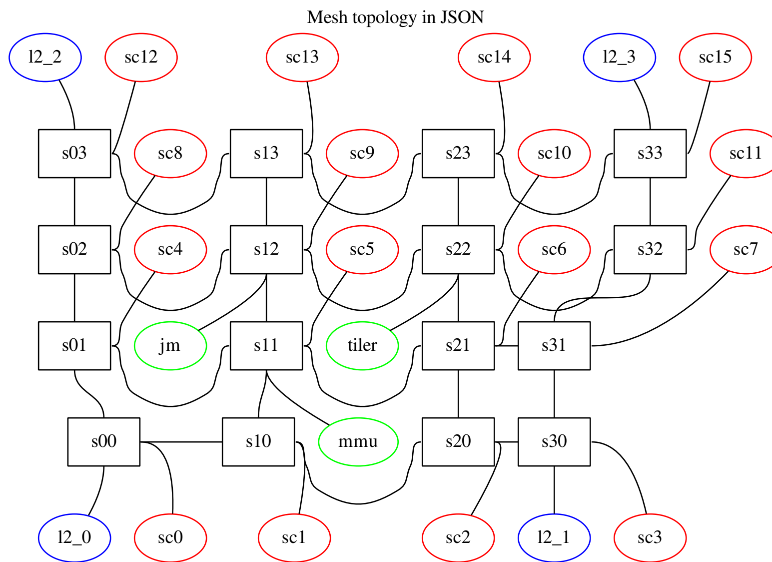

Figure 4.1: Visualization of a mesh network topology, produced by the visualization script in combination with Dot.

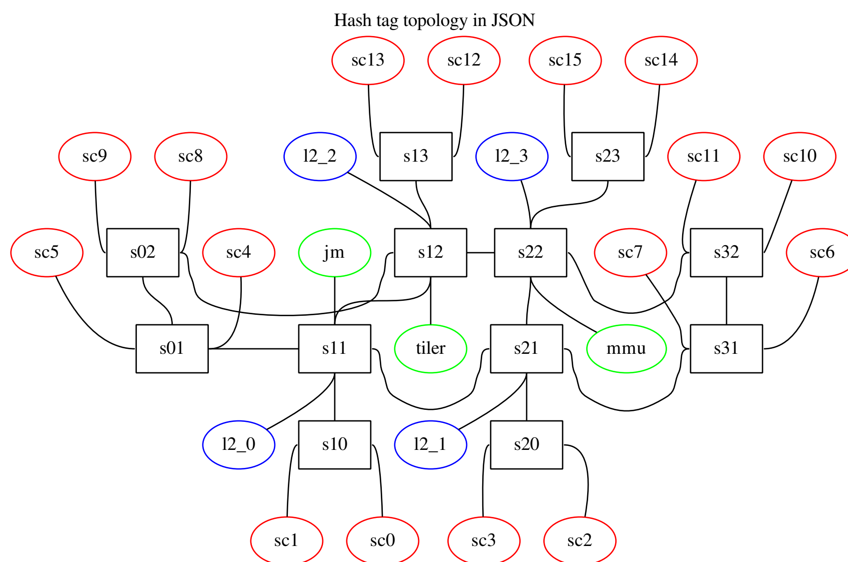

Figure 4.2: Visualization of a network topology we call the hash tag topology, produced by the visualization script in combination with Dot.

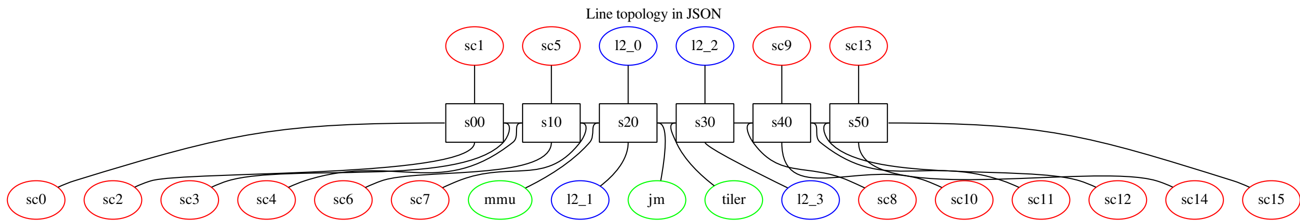

Figure 4.3: Visualization of a network topology we call the line topology, produced by the visualization script in combination with Dot.

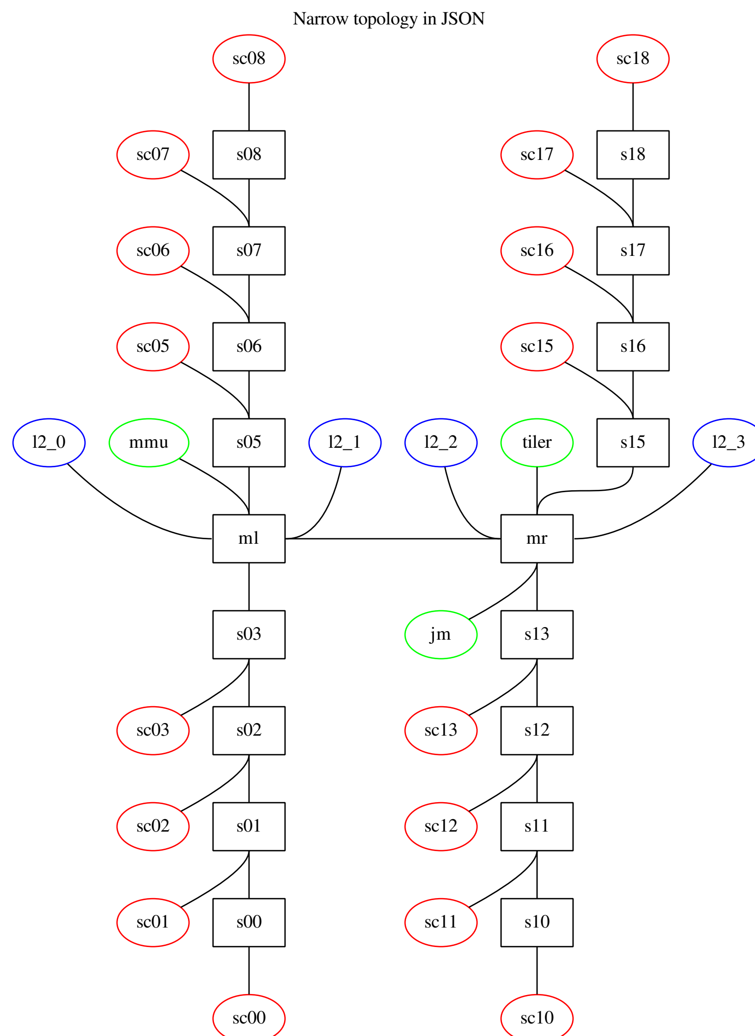

Figure 4.4: Visualization of a network topology we call the narrow topology, produced by the visualization script in combination with Dot.

Chapter 5
Discussion

In this section, we discuss the impact of the results.

5.1 Custom Topologies and Abstraction

Our implementation supports any custom topologies possible to implement in the Arm GPU model. This is important, because it allows users to explore the design space freely. A limit on possible topologies would have forced users to sometimes edit the source code. It would then not have been enough only to know about the network description language. This would have partly defeated the purpose of providing a level of abstraction, which lets users define and explore topologies without knowledge about the underlying source code.

There are, however, limitations to which topologies can actually be implemented. Only four-port switches are supported, and each such switch can only connect to maximally four other switches. The switches must also be connected in such a way that their coordinates match. For instance, the switch connected on the north side of a switch with the coordinates (x=1, y=1) must have the coordinates (x=1, y=2). A network can only contain one crossbar. If a network contains a crossbar, it cannot contain any switches. These limitations are consequences of the design of the Arm GPU model, in which they are present. It is beyond the scope of this project to change this. If those restrictions were lifted from the model, it would only require small changes to make our extension support the new possibilities.

The user can not choose which routing algorithm to use during the simulation. The routing algorithm is currently hard coded inside the Arm GPU model. We do, however, think it would be a useful improvement of the network description language if it could be used to define, or at least choose from a list of, routing algorithms.

We believe the added abstraction of the network setup will lower the threshold for new users. Defining a network topology now requires no knowledge of neither C++, nor of the model internals. Looking at example topologies, such as those in Appendix B, is probably enough for a new user to get started. This will be faster than reading and understanding

the source code. For more details, users can turn to the manual presented in Appendix A.

In addition to lowering the threshold for new users, we believe the abstraction will speed up exploration of new ideas by the experienced engineers. Making changes to a topology, such as changing which switch a specific component is connected to, requires very small changes in the description file. The new tests can then be run directly, without recompilation. The location of the edited network description file is passed as a command line argument when the simulation is started.

We consider reaching the four last criteria defined in Section 3.2 to be part of making definition of all topologies supported by the model possible.

5.2 Support for Existing Topologies

Implementing support for all hard coded topologies was a requirement to make a future migration to our solution possible. Had there been topologies the model extension could not support, only a partial migration would have been possible. Furthermore, it would not have been enough only to know the network description language.

One can argue that supporting all previously implemented topologies follows by implication from supporting all topologies that are possible to implement in the Arm GPU model. We did, however, not settle on making it possible to define all previously hard coded topologies. We went further, by adding features that made defining those topologies compact and efficient. A feature that recurs frequently in the hard coded topologies is shader core stacks. By adding the stack concept to our network description language, we significantly reduced the lines of code needed to define the networks.

5.3 Lines of Code

The number of lines of code does not, in itself, say much about the quality of the implementation. The 48% average reduction does, however, indicate that defining a network topology now requires less code.

Many of the hard coded topologies are defined in the same blocks of code. This is because the differences between some of them are subtle. Topology 2 to 7 in Table 4.1 are defined in the same block. The same applies to topology 8 and 9. Inside those blocks, there are logic statements that decide which parameters to use. In Table 4.1, this logic is included in the line count for each hard coded topology. Had these logic statements been removed, the line count for the hard coded topologies would have been smaller. This would, in turn, have reduced the difference.

Topology 10 stands out in Table 4.1, because it takes 3.48 times more code to define it when using our network description language. This is partly because Topology 10 contains 32 shader cores, organized in a very simple structure. In the hard coded version, the addition of the shader cores is implemented using a loop. In our network description language, it is instead implemented as a list of 32 switches. The same applies to the eight L2 cache units in the topology. The network definition in our network description language is simpler, but longer. One way to enable shorter definitions of simple networks could be to add some sort of list comprehension to the description language. We discuss this idea

further in Section 6.1.5.

5.4 Flexibility of Description Files

Each hard coded topology supports several different numbers of shader cores and L2 cache units. Ideally, we also wanted to support all different numbers of cores and L2 cache units for each topology in a single configuration file. This would be beneficial, because it keeps the number of configuration files small. Hence, fewer files needs maintenance and testing. When something is changed in a topology, it only needs to be changed in one place.

As can be seen in Table 4.2, it is enough with one configuration file for most of the topologies. What all those topologies have in common, is that additional shader cores are added in a simple, regular, and easy to describe order. In topology 2-8, shader core stacks are used, to which additional cores are added to in a consistent order.

The topology that stands out in Table 4.2 is topology 9. Topology 9 is, like most other, built by shader core stacks. There are mainly two aspects that set this topology apart from the other. First, shader core stacks can not be used for some core counts. This is because of an optimization which replaces core stacks with only one shader core, with shader cores directly connected to the rest of the topology. Because our description language lacks support for this optimization, the stacks needs to be replaced by manually defined cores and switches in the files intended for those core counts. Second, the order in which the stacks get longer when shader cores are added is not consistent.

Those results show that our network description language can support several different core counts for the same topology, as long as the order in which additional cores are added is consistent. Adding cores in an unstructured order is possible, but requires separate files for different core numbers.

Bypassable switches and removal of dead end switches are two features that were added to increase the flexibility of the description files. Those features did not, however, impact the numbers in Table 4.2. Despite this, we believe the addition of those features adds flexibility, though not for those particular topologies.

According to our supervisor at Arm, making non-trivial modifications to existing topologies was as cumbersome as writing new topologies from scratch when they were hard coded. Furthermore, reusing code from one topology in another was hard [47]. With our network description language, we believe making changes is straight forward. Extracts from description files is often possible to reuse in other description files.

5.5 Comparison with Source Code

Based on the comparisons with source code, we believe the introduction of our language has made network definition more accessible. The comparisons showed in Listing 4.1 to 4.6 should give the reader an impression of how the configuration has changed.

In Listing 4.1, we can see that each switch has to be created (line 9) before being connected (line 11). This corresponds to the declaration of the switches (line 2-5) and the links connecting them (line 11-14) in Listing 4.2. Both steps, creating and connecting the switches, are present in both ways to define networks. We believe the way it is done in the

description language is better, though, because it provides a clear separation between, and location for, the creation and the connection.

As can be seen in Listing 4.3 and 4.5, setting up networks the hard coded way often included using nested for loops, together with logical statements. Though not necessarily very complex, we find this approach to be more difficult than the corresponding solutions presented in Listing 4.4 and 4.6. We also expect our solution to be less error prone, especially when used together with the visualization script.

The simplification of the setup of shader core stacks is one of the features we think will be most useful. Out of the ten hard coded topologies, eight use shader core stacks. This makes this feature likely to be frequently used in the future. As can be seen by comparing Listing 4.5 with Listing 4.6, creating shader core stacks is compact in our description language.

We believe the new way to define networks is easier to learn, because the content of the configuration is simpler than before. In many cases, it should be possible to start from an existing description file and edit it. Defining a network no longer requires any programming language constructs, such as loops or conditional statements. This makes it possible for users without programming knowledge to define networks, even though we find it unlikely that anyone without programming experience would use the model.

5.6 Visualization

Before this project, there was no support for topology visualization in the Arm GPU model. We think the addition of visualization capabilities is useful in several ways, which we describe below.

We believe the quality of the visualizations is high enough for them to be useful. According to our supervisor at Arm, it has happened that well after a topology was modeled, it was discovered that components were not connected in the place the Arm engineers thought [47]. We think this kind of mistakes will be possible to detect at an early stage using the visualization tool.

We also think the visualizations are good enough to simplify discussions about the network topology. From a discussions with our supervisor at Arm, we learned that confirming with the hardware engineers that the design of a hard coded topology is correct, previously required a modeling engineer to manually draw a diagram, or explain by walking through the code together with the hardware engineer [47]. In the future, we think the visualizations will provide a useful overview of the network, which can be used for topology confirmation.

The visualizations would, according to us, be clearer if the switch-to-switch connections were drawn as straight lines. This could make it clearer which those connections are, which in turn would give a better overview of the topology. We tried to achieve this by assigning large weights in DOT to the connections between the switches, but did not manage to produce the desired results.

5.7 Design Decisions

One topic we discussed early on was whether to create a program that generated source code that could be pasted into the model, or include a markup language parser. We chose to include a parser, to allow network definition at runtime rather than compile time. Parsing also allows modifications of the NoC without access to the source code of the model.

We had to decide which markup language to use for the network description. As part in this decision process, we implemented the same topology in the five markup languages we considered using. Those implemented topologies can be seen in Appendix C. For several reasons, we chose JSON. First, it is a general markup language. Hence, it can be used to represent arbitrary data structures. This, for instance, made the addition of shader core stacks straight forward. Second, of the three general markup languages we considered, we found JSON to be the most concise and clear. This is, however, our subjective opinion. We found it easy to read, yet structured in a clear way. We believed the clear structure would simplify the parser implementation. Third, JSON is a widely known and used format. Our intention was that people already familiar with JSON should be able to learn our language quickly. We could not find a JSON parser with a license allowing usage in the Arm GPU model, so we implemented our own.

As we mentioned in 2.2, Netjson is a network description format based on JSON. Because we knew Netjson successfully used JSON for network description, we knew using JSON would be feasible. That also influenced our decision to use JSON.

We also had to decide which visualization software to use. We chose the software Dot, whose DOT format we considered using for the network description in Section 3.1.3. Dot creates an image of a graph given an input file in DOT format. It takes care of much of the graph layout itself, but still gives the user the option to decide which nodes should appear on the same level in the image. The user can also define in which direction the links should leave and enter the nodes. Our conclusion was that Dot provides us with the functionality we need, with a simple configuration system.

We chose JSON for the network description, but DOT for the visualization. The reason behind our choice to use JSON, instead of for instance DOT, for the network definition was that we wanted to be able to add custom concepts such as shader core stacks. By using a general markup language such as JSON we got a large amount of freedom on how to design the language specifically for NoC specification. We also thought JSON would be easier to learn and use. This meant that a program that produces a DOT version of a JSON network description had to be implemented. We chose to implement this in Python, because it contains an easy to use library for JSON parsing, and is well suited for text manipulation. The generation of DOT files is not time critical, so the overhead of using an interpreted language such as Python is not a problem.

The Arm GPU model is implemented in C++. Therefore, we used C++ for our extension of the Arm GPU model too.

5.8 New Topologies

In this section, we discuss the performance of the four implemented topologies. As mentioned, performance was not the primary goal when designing those topologies. We do,

however, still find it interesting to evaluate and discuss their performance.

5.8.1 Mesh and Hash Tag

The results for the mesh topology, which are shown in Table 4.3, show a GPU clock cycle count reduction of four percent. The memory speed related metrics show reductions of between 10% and 44%. For the hash tag topology, the GPU clock cycle count reduction is five percent. The memory speed improvements are between 10% and 23%.

Though the improvements of the selected metrics seem promising for mesh and hash tag, more analysis would be required before any conclusions could be drawn about whether they are actually better than the default topology. The simulations do not, for instance, take area or power consumption into consideration. Those metrics are decisive when designing SoCs, and would have to be investigated if Arm wanted to proceed with either topology. Furthermore, the simulations were run on a limited set of graphics benchmarks. This content is probably not representative for all use cases. To learn more about the performance of the topologies, it would be interesting to carry out simulations using a larger set of graphical content.

5.8.2 Line and Narrow

One simulation run using the narrow topology, and three simulation runs using the line topology, resulted in time out due to network congestion. Network congestion results in unacceptable performance in the GPU NoC. Therefore, those topologies can be discarded.

Even when not considering the starved simulation runs, the line and narrow topologies do not seem to provide good performance. The total clock cycle count and the L1 read latency are both almost unaltered compared to the default topology. The metric standing out is the L2 cache message stall, which is 196% higher for the line topology than for the default topology. For the narrow topology, it is 15% higher. We believe the long message stalls are consequences of the reduced number of ways the data can take when traversing those networks.

The L1 write latency is lower than for the default topology, ten percent lower for line and three percent lower for narrow. This does not, however, weigh up for the other results for the narrow and line topologies.

Chapter 6
Conclusions

We have implemented a network description format which simplifies the definition of NoCs. By building our description language upon JSON, we made the syntax of our language easy to learn and accessible for many users. Using Dot provides a solution for visualizing the network topologies. When many connections are defined, the generated image can give a cluttered impression. The quality of the visualizations does, however, let the user inspect the defined network to see that the correct components are connected to each other. This should make the exploration of new topologies less error prone.

We also conclude that the introduction of an abstraction level makes it easier and faster to define new networks. This is partly because the definition becomes more compact, but also because subtle changes can be done with very small changes in the description files. The possibility to change the topology without recompiling the model further speeds up the process.

In addition to making the network definition easier and faster, our network description language makes NoC exploration more accessible and easier to learn. All the user needs to know is the network description language. Knowledge of the underlying source code is not required.

Because the project was carried out within the GPU modeling team at Arm, the resulting description language is designed to suit topologies possible to implement in their GPU model. Some of the restrictions of the language would not have been introduced had it not been for model compatibility. We still think it was valuable to implement the language targeting a specific simulator, because it made us aware of what aspects are important to model and define in an NoC.

6.1 Future Work

In this section, we will point out possible improvements for our network description language.

6.1.1 Flexible Switches

In its current state, the network description language only supports two types of switches; four-port switches and crossbars. The four-port switches can be connected to up to four other four-port switches. A possible improvement would be to add more connections to the four-port switches, such as connections in the north-east, south-east, south-west, or north-west direction. Each topology can, for now, only contain one crossbar. Allowing several crossbars to build a network together would be an improvement. This would allow modeling more general networks. It would require changes in the GPU model, such as implementations of more complex routing algorithms, though.

6.1.2 Routing Algorithms

The description language currently only lets the user define the topology of the network, but there are other aspects which also defines an NoC. One such is the choice of routing algorithm. It would be interesting to add selection and/or definition of routing algorithms to the language. As a first step, one could enable the selection of routing algorithms from a predefined set. Going further, one could enable specification of routing algorithms.

6.1.3 More Configuration Parameters

One change which might make the description language more useful during simulations could be allowing more simulation configuration parameters to be defined in it. For instance, one could add the definition of the sizes of the L2 cache units. This is not necessarily an improvement over the existing solution, but investigating it could be interesting.

6.1.4 RTL Generation

Currently, support for the language is only implemented in the Arm GPU simulator. Future work could be to implement software that generates a hardware description in RTL of the specified network. The output could be in VHDL or Verilog. Inspiration for this could be acquired by studying the Atlas project mentioned in Section 2.3, which support RTL generation as well as simulation.

6.1.5 List Comprehension

As can be seen by looking at the example topologies shown in Appendix B, the description files often include sets of almost identical lines. For instance, each switch is defined on an individual line, unless shader core stacks are used. The variation between the switches often follow a simple pattern, such as stepwise increment of a coordinate. It might be possible to make the network definition more compact by introducing concepts similar to list comprehension. This could, for instance, make it possible to define on a single line that you want five switches with a fixed x coordinate and y coordinates spanning from zero to four. One risk with this approach is that network definition becomes more complex and more like programming. Investigating this option could still be interesting, though.

Bibliography

[1] W. J. Dally and B. Towles. *Principles and practices of interconnection networks*. Morgan Kaufmann Publishers, Amsterdam ; San Francisco, 2004.

[2] L. Benini and G. De Micheli. Networks on chips: a new SoC paradigm. *Computer*, 35(1):70–78, January 2002.

[3] J. L. Hennessy, D. A. Patterson, and K. Asanović. *Computer architecture: a quantitative approach*. Morgan Kaufmann/Elsevier, Waltham, MA, 5th ed edition, 2012. OCLC: ocn755102367.

[4] Arm Limited. Mali gpu - arm, 2017. https://www.arm.com/products/graphics-and-multimedia/mali-gpu, Accessed: 2017-09-11.

[5] Moore's law. *Columbia Electronic Encyclopedia, 6th Edition*, page 1, 2017.

[6] M. Vachharajani, N. Vachharajani, and D. I. August. The Liberty Structural Specification Language: A High-level Modeling Language for Component Reuse. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 195–206, New York, NY, USA, 2004. ACM.

[7] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[8] D. A. Patterson and J. L. Hennessy. *Computer organization and design: the hardware/software interface*. Morgan Kaufmann Publishers, an imprint of Elsevier, Cambridge, Massachusetts, risc-v edition edition, 2018. OCLC: ocn993666159.

[9] T. Möller, E. Haines, and N. Hoffman. *Real-time rendering*. A.K. Peters, Wellesley, Mass, 3rd ed edition, 2008. OCLC: ocn213765720.

[10] P. Harris. The mali gpu: An abstract machine, part 3 - the midgard shader core. https://community.arm.com/graphics/b/blog/posts/the-mali-gpu-an-abstract-machine-part-3—the-midgard-shader-core, Accessed: 2017-11-29.

[11] Intel Corporation. 8th generation intel® core™ i7 processors. https://ark.intel.com/products/series/122593/8th-Generation-Intel-Core-i7-Processors, Accessed: 2017-10-23.

[12] Nvidia. https://www.nvidia.se/graphics-cards/geforce/pascal/compare-specs/, Accessed: 2017-10-23.

[13] S. Kumar, A. Jantsch, J. P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani. A network on chip architecture and design methodology. In *Proceedings IEEE Computer Society Annual Symposium on VLSI. New Paradigms for VLSI Systems Design. ISVLSI 2002*, pages 105–112, 2002.

[14] I. Cutress. Apple 2017: The iphone x (ten) announced. https://www.anandtech.com/show/11835/apple-2017-the-iphone-x-ten, Accessed: 2017-11-08.

[15] Internet Engineering Task Force. Requirements for internet hosts – communication layers. https://tools.ietf.org/html/rfc1122, Accessed: 2017-10-16.

[16] W. J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 684–689, 2001.

[17] A. B. Achballah and S. B. Saoud. A Survey of Network-On-Chip Tools. *International Journal of Advanced Computer Science and Applications*, 4(9), 2013.

[18] M. Ali, M. Welzl, A. Adnan, and F. Nadeem. Using the NS-2 Network Simulator for Evaluating Network on Chips (NoC). In *2006 International Conference on Emerging Technologies*, pages 506–512, November 2006.

[19] Y. Sun, S. Kumar, and A. Jantsch. Simulation and evaluation for a network on chip architecture using ns-2. In *Proceedings of the IEEE NorChip conference*, 2002.

[20] M. Lis, K. S. Shim, M. H. Cho, P. Ren, O. Khan, and S. Devadas. Darsim: A Parallel Cycle-Level NoC Simulator. *MIT web domain*, 2010.

[21] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti. Cycle-Accurate Network on Chip Simulation with Noxim. *ACM Trans. Model. Comput. Simul.*, 27(1):4:1–4:25, August 2016.

[22] J. Wang, Y. Huang, M. Ebrahimi, L. Huang, Q. Li, A. Jantsch, and G. Li. VisualNoC: A Visualization and Evaluation Environment for Simulation and Mapping. In *Proceedings of the Third ACM International Workshop on Many-core Embedded Systems*, MES '16, pages 18–25, New York, NY, USA, 2016. ACM.

[23] F. Capoano and L. Kaplan. Netjson: data interchange format for networks, 2015. http://netjson.org/rfc.html, Accessed: 2017-09-11.

[24] Ecma. The json data interchange format, 2013. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf, Accessed: 2017-09-11.

[25] Microsoft. The osi model's seven layers defined and functions explained. https://support.microsoft.com/en-us/help/103884/the-osi-model-s-seven-layers-defined-and-functions-explained, Accessed: 2017-11-15.

[26] Netjson. http://netjson.org/, Accessed: 2017-12-06.

[27] Netjson network topology visualizers. http://netjson.org/docs/implementations.html#network-topology-visualizers, Accessed: 2017-12-19.

[28] https://github.com/netjson/netjsongraph.js, Accessed: 2017-12-19.

[29] J. Lang. Atlas. https://corfu.pucrs.br/redmine/projects/atlas/wiki, Accessed: 2017-09-14.

[30] A. Mello, A. Amory, N. Calazans, and F. Moraes. Atlas a noc generation and evaluation framework. 2011. Poster presented at the DATE11 conference in Grenoble.

[31] N. Jiang, J. Balfour, D. U. Becker, B. Towles, W. J. Dally, G. Michelogiannakis, and J. Kim. A detailed and flexible cycle-accurate Network-on-Chip simulator. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 86–96, April 2013.

[32] https://github.com/booksim/booksim2, Accessed: 2017-09-13.

[33] https://github.com/booksim/booksim2/blob/master/src/examples/anynet/anynet_file, Accessed: 2017-12-06.

[34] Hornet. http://csg.csail.mit.edu/hornet/, Accessed: 2017-12-14.

[35] https://github.com/davidepatti/noxim, Accessed: 2017-09-14.

[36] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti. Noxim: An open, extensible and cycle-accurate network on chip simulator. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 162–163, July 2015.

[37] F. Fummi, G. Perbellini, P. Gallo, M. Poncino, S. Martini, and F. Ricciato. A timing-accurate modeling and simulation environment for networked embedded systems. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*, pages 42–47, June 2003.

[38] N. Agarwal, T. Krishna, L. Peh, and N. Jha. Garnet: A detailed on-chip network model inside a full-system simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 33–42. IEEE, 2009.

[39] Simple. http://gem5.org/Simple, Accessed: 2017-09-15.

[40] H. Wang, X. Zhu, L. Peh, and S. Malik. Orion: a power-performance simulator for interconnection networks. *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings., Microarchitecture, 2002.*

(MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on, Microarchitecture, page 294, 2002.

[41] W3C. Extensible markup language (xml) 1.0 (fifth edition). https://www.w3.org/TR/xml/, Accessed: 2017-12-07.

[42] O. Ben-Kiki, C. Evans, and I. döt Net. Yaml ain't markup language (yaml™) version 1.2. http://www.yaml.org/spec/1.2/spec.html, Accessed: 2017-10-05.

[43] The dot language. https://graphviz.gitlab.io/_pages/doc/info/lang.html, Accessed: 2017-12-07.

[44] M. Himsolt. Graphlet: design and implementation of a graph editor. *Software: Practice and Experience*, 30(11):1303–1324, September 2000.

[45] http://www.graphviz.org/, Accessed: 2017-10-05.

[46] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, 2005.

[47] F. Tabba, Arm Sweden AB. Personal communication.

[48] Weighted round robin scheduling. http://www.brocade.com/content/html/en/configuration-guide/nos-700-l2guide/GUID-B75BD370-B251-45A3-B3FD-87F54E35DC6E.html, Accessed: 2017-12-19.

Appendices

Appendix A
Network Description Language Manual

A central part of the project was to design a network description language that could be used to define an NoC. In this manual, we will go through this language.

This manual starts by an introduction to JSON, followed by sections explaining the different data structures constituting the description file. For each data structure, a table is supplied with the details of the structure. At the end of the manual, there is a section which covers the limitations of the language. This part, among other things, explains which combinations of attributes are not allowed.

All objects in each network are identified using their id string attribute. The ids of all objects in the topology must therefore be unique.

A.1 Introduction to JSON

The network description language is based on the JSON markup language format. To understand this manual, the reader needs to be familiar with the concepts of JSON arrays and JSON objects.

A JSON array is a comma separated list of elements. A JSON array is enclosed by square brackets, and its content is accessed by index. Here is an example of a JSON array: ["a", "b", "c"].

A JSON object is a collection of key-value pairs, in which the keys are used to access the values. A JSON object is enclosed by curly brackets. Here is an example of a JSON object: {"a": "banana", "b": "orange", "c": "apple"}.

A.2 Outermost Object

At the outermost layer, the content of each network description file is wrapped in a JSON object. The content of the outermost JSON object is described in Table A.1. Some at-

tributes must be present in a valid topology, while other are optional. In Appendix B, two example description files are presented. In the following sections, we will go through the attributes of the description file in detail.

Table A.1: Content of the outermost object of a description file.

Key	Content type	Description	Mandatory
label	String	A label describing the topology.	No
max_cores	Integer	The maximum number of cores supported by the topology.	No
min_cores	Integer	The minimum number of cores supported by the topology.	No
max_l2_caches	Integer	The maximum number of L2 cache units supported by the topology.	No
min_l2_caches	Integer	The minimum number of L2 cache units supported by the topology.	No
switches	Array of switch objects	Defines the switches in the topology.	Yes
crossbars	Array of crossbar objects	Defines the crossbars in the topology.	Yes
shader_cores	Array of core objects	Defines the cores in the topology.	Yes
l2_caches	Array of cache objects	Defines the L2 cache units in the topology.	Yes
mmu	Array of memory management unit objects	Defines the memory management units in the topology.	Yes
tiler	Array of tiler unit objects	Defines the tilers in the topology.	Yes
job_manager	Array of job manager objects	Defines the job managers in the topology.	Yes
links	Array of link objects	Defines the links connecting the components in the topology.	Yes
core_stacks	Array of core stack objects	Defines the core stacks of the topology.	No
core_stack_config	Object with meta information about the core stacks	Defines attributes for the core stacks.	No

A.3 Switches

The switches of a topology are defined in the switch array introduced in Table A.1. In the array, each switch is represented as an object containing information about id, x coordinate, and y coordinate. The coordinates are used to decide the layout of the network. The origin of the coordinate system is in the lower left corner. The y coordinate can, instead of being explicitly set using an integer, be defined as "core_switch_lower" or "core_switch_upper". If it is defined as either of those, it will be assigned dynamically. "core_switch_lower" will set the y coordinate to the length of the longest south facing core stack. "core_switch_upper" will set it to the longest south pointing core stack plus one. Core stacks are described in Section A.11. Table A.2 below describes the structure of a switch object.

A.4 Bypassable Switches

Switches can be defined to be possible to bypass. If they are, that means that if the switch is connected to only two other switches on opposite sides, and nothing else, it will be omitted from the topology. The two switches on the opposite sides of the omitted switch then get connected directly to each other.

A.5 Input Priorities

For each switch, it is possible to assign a higher priority to data that traverses the switch via specific input and output ports. If no priorities are explicitly defined, the switch will arbitrate between the ports using round robin. If priorities are defined, weighted round robin is used instead. The bandwidth of the switch is then divided between the ports proportional to their assigned weights, but the order in which the ports are serviced is still predefined [48]. Each switch object can optionally contain an array called weights, in which the switch's priority weights are defined. Each weight is defined as a weight object. The details of switch weight objects are defined in Table A.3. If the user, for instance, want traffic in the north-south direction to be prioritized twice as high as traffic in any other direction, the following entry should be added to the switch object: "weights":{"n-s":2}.

A.6 Dead End Switches

If a switch is only connected to one other switch, and nothing else, it is considered to be a dead end switch. No data passes through dead end switches, and the they are therefore omitted during the generation of the network.

Table A.2: Content of a switch object.

Key	Content type	Description	Mandatory
id	String	Identity of the switch, used to reference it. Must be unique.	Yes
x	Integer	X coordinate.	Yes
y	Integer or "core_switch_upper" or "core_switch_lower"	Y coordinate. If it is an integer the coordinate is set explicitly. If it is defined as one of the custom strings, it is set dynamically.	Yes
bypassable	Boolean	Indicates whether the switch may be bypassed.	No, defaults to false.
weights	Array of weight objects	Contains weights prioritizing data at specific ports.	No

Table A.3: Content of a switch weight object.

Key	Content type	Description	Mandatory
Source and target direction, separated by a - . For instance, "n-s" or "e-w".	Integer	Weight assigned to the internal connection between the specified ports.	Yes

A.7 Crossbars

The crossbars are defined in the crossbar array. Currently, the language only supports one crossbar per network. If a crossbar is defined, the network must not contain any switches. Each crossbar is represented as an object with the only attribute id. Table A.4 describes the structure of a crossbar object.

Table A.4: Content of a crossbar object.

Key	Content type	Description	Mandatory
id	String	Identity of the crossbar, used to reference it. Must be unique.	Yes

A.8 Shader Cores

The shader cores are defined in the shader core array. Each shader core is defined as an object with the attributes id and core number. The core number is used to identify the core internally in the GPU model. Providing numbers is optional, but must be provided

for either all or none of the cores. If no numbers are provided, the cores get assigned their index number in the JSON array as their core number.

The order in which the cores are added matters. If the number of cores to be added, n, is smaller than the number of specified cores, only the first n cores in the core array will be added.

Table A.5 describes the structure of a shader core object.

Table A.5: Content of a shader core object.

Key	Content type	Description	Mandatory
id	String	Identity of the shader core, used to reference it. Must be unique.	Yes
number	Integer	Core number, used to reference the core inside the GPU model.	No

A.9 MMUs, L2 Cache Units, Tilers, and Job Managers

The MMUs, L2 cache units, tilers, and job managers are defined in their respective arrays, which are listed in Table A.1. The objects for all of those components contain only one attribute, their id.

Table A.6 describes the content of MMU, L2 cache unit, tiler and job manager objects.

Table A.6: Content of MMU, L2 cache, tiler, and job manager objects.

Key	Content type	Description	Mandatory
id	String	Identity of the unit, used to reference it. Must be unique.	Yes

A.10 Links

The links connecting the components in the topology are defined as objects in the link array. Each such object contains information about source and target node, identified by their ids. If any of the nodes is a switch, the port of the switch connection is defined.

For each link, it is also possible to define a delay. The delay is used during simulations to model the time it takes for a signal to traverse the link.

Table A.7 describes the content of a link object.

Table A.7: Content of a link object.

Key	Content type	Description	Mandatory
source_node	String	The id of the source component.	Yes
target_node	String	The id of the target component.	Yes
source_port	String ("n", "e", "s", or "w")	The port at which the link connects to the source node, if it is a switch.	Only when source node is a switch.
target_port	String ("n", "e", "s", or "w")	The port at which the link connects to the target node, if it is a switch.	Only when target node is a switch.
delay	Integer	The link delay.	No, defaults to zero.

A.11 Shader Core Stacks

Shader core stacks are linear arrays of switches, with one shader core connected to each switch. If shader core stacks are present in a network, there must not be any shader cores that are defined individually. The configuration of the shader core stacks is divided into two parts.

The attributes that are configured per stack are defined in shader core objects inside the core stack array. Each such object contains information about the direction the stack faces, via which switch the stack is connected to the network, and stack id. In the current implementation, the only supported directions are north and south. The content of each shader core stack object is described in Table A.8.

The network description file can optionally contain a core stack configuration object, in which parameters that apply to all shader core stacks are set. Those parameters include switch delay, core delay, root delay, whether the length of the stacks should be kept as balanced as possible, max length, and the direction in which switches connect to cores. Switch delay defines the delays for the connections between the switches in the stack. Core delay defines the delays for the connections between the switches and the shader cores. Root delay defines the delay of the connection that connects the stack to the rest of the network. The direction in which the cores connect to the switches can be chosen to be north, east, south, west, or the same direction the stack itself is pointing. The content of each core stack configuration object is described in Table A.9.

Table A.8: Content of a core stack object.

Key	Content type	Description	Mandatory
id	String	Identity of the shader core stack, used to reference it. Must be unique.	Yes
base	String	Id of the switch that the stack is connected to.	Yes
direction	String	The direction in which the stack points ("n" or "s").	Yes

Table A.9: Content of the core stack configuration object.

Key	Content type	Description	Mandatory
switch_delay	Integer	Delay on links between switches.	Yes
core_delay	Integer	Delay on links between switches and shader cores.	Yes
root_delay	Integer	Delay on link between each stack and the rest of the network.	No, defaults to zero.
balanced_stacks	Boolean	Determines whether the length of the stacks should be kept as equal as possible.	Yes
max_length	Integer	Max length of a stack.	Yes
core_direction	String	The direction in which the shader cores are connected to the switches ("n", "e", "s", "w", or "stack_direction").	No, defaults to w.

A.12 Limitations

In this section, we will present important limitations the user must be aware of when defining a network.

A.12.1 Switches and Crossbars

If a crossbar is defined in the network, the network must not contain any switches. A network can not have more than one crossbar. This means that a network either contains one crossbar, or an arbitrary number of switches.

A.12.2 Shader Core Stacks

If a network contains any shader core stacks, all shader cores must be defined by the stacks. No shader cores can be added to the network separately in the "shader_cores" array.

Appendix B
Example Topologies

In this appendix, we present two example topologies. For each topology, we include the definition of the topology in the JSON network description language, as well as the visualization produced by the visualization script. Please note that the presented topologies are not necessarily the topologies used in the actual Arm GPU hardware.

B.1 Topology 1

The topology defined in Listing B.1 illustrates the use of shader core stacks. In the description file, 16 shader cores and four L2 cache units are defined. When we ran the visualization script, we chose to include only 14 cores and three caches. Hence, only 14 cores and three caches can be seen in Figure B.1. Because "balanced_stacks" is set to true, the length of the stacks is kept as equal as possible.

Listing B.1: Specification of Topology 1

```
1  {
2      "label": "Topology 1 in JSON",
3      "min_cores": 1,
4      "max_cores": 16,
5      "switches": [
6          {"id": "left", "x": 0,"y": "core_switch_lower"},
7          {"id": "right","x": 1,"y": "core_switch_lower"}
8      ],
9      "crossbars": [
10     ],
11     "core_stacks": [
12         {"id": "stack0", "base": "left", "direction": "n"},
13         {"id": "stack1", "base": "left", "direction": "s"},
14         {"id": "stack2", "base": "right", "direction": "n
               "},
15         {"id": "stack3", "base": "right", "direction": "s"}
```

```
16      ],
17      "core_stack_config": {
18          "switch_delay": 0,
19          "core_delay": 0,
20          "root_delay": 3,
21          "balanced_stacks": "true",
22          "max_length": 4,
23          "core_direction": "stack_direction"
24      },
25      "shader_cores": [
26      ],
27      "l2_caches": [
28          {"id": "l2_0"},
29          {"id": "l2_1"},
30          {"id": "l2_2"},
31          {"id": "l2_3"}
32      ],
33      "mmu": [
34          {"id": "mmu","label": "mmu"}
35      ],
36      "tiler": [
37          {"id": "tiler","label": "tiler"}
38      ],
39      "job_manager": [
40          {"id": "jm", "label:": "job manager"}
41      ],
42      "links": [
43          {"source_node": "left","target_node": "right","
               source_port": "e","target_port": "w"},
44
45          {"source_node": "left","target_node": "mmu","
               source_port": "n"},
46          {"source_node": "right","target_node": "jm","
               source_port": "s"},
47          {"source_node": "right","target_node": "tiler","
               source_port": "n"},
48
49          {"source_node": "left","target_node": "l2_0","
               source_port": "w"},
50          {"source_node": "left","target_node": "l2_1","
               source_port": "e"},
51          {"source_node": "right","target_node": "l2_2","
               source_port": "w"},
52          {"source_node": "right","target_node": "l2_3","
               source_port": "e"}
53      ]
54  }
```

B.2 Topology 2

The topology defined in Listing B.2 illustrates bypassing of bypassable switches, and removal of dead end switches. In the configuration file, it can be seen that five switches are defined. Because switch "s31" is only connected to one other switch and nothing else, it

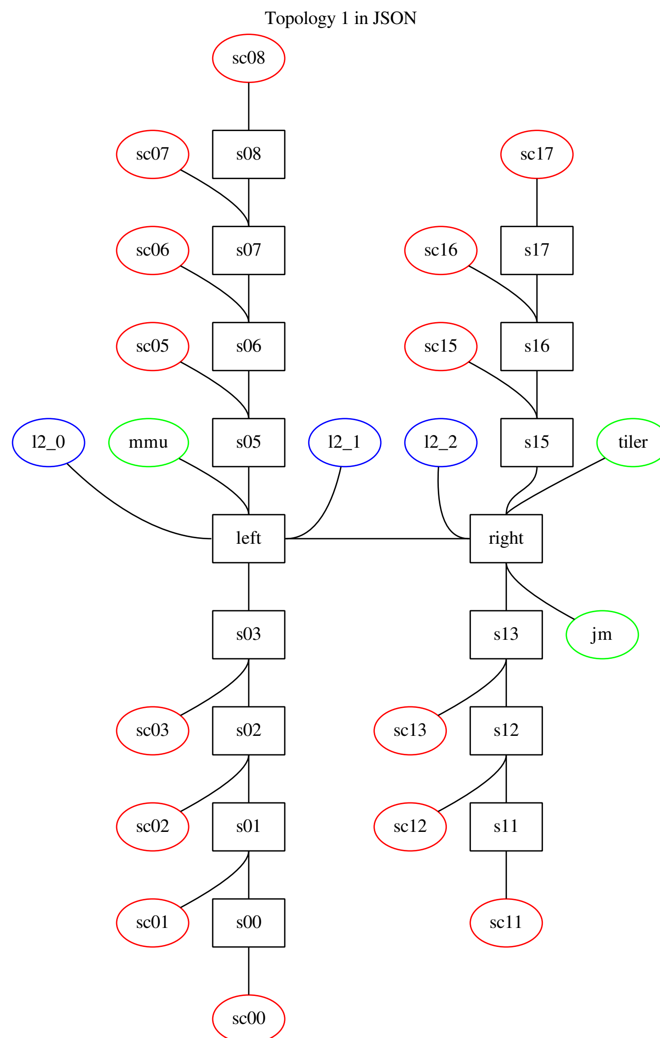

Figure B.1: Visualization of Topology 1, produced by the visualization script in combination with Dot. When the script was run, the shader core count was limited to 14 and the cache unit count to three.

is a dead end switch. Hence, it is omitted from the generated topology in Figure B.2.

The switch "s20" is marked as bypassable. Because it is only connected to two other switches on opposite sides, and nothing else, it is bypassed. In the visualization, this is indicated by drawing the switch with a dashed border.

When the figure was generated, we chose to limit the number of shader cores to two and the number of L2 cache units to one. This resulted in the shader core "sc2" and the L2 cache unit "l2_1" being omitted from the network.

Listing B.2: Specification of Topology 2

```
1 {
2     "label": "Topology 2",
3     "switches": [
4         {"id": "s00","x": 0,"y": 0},
5         {"id": "s10","x": 1,"y": 0},
```

```
6           {"id": "s20","x": 2,"y": 0,"bypassable": "true"},
7           {"id": "s30","x": 3,"y": 0},
8           {"id": "s31","x": 3,"y": 1}
9       ],
10      "crossbars": [
11      ],
12      "shader_cores": [
13          {"id": "sc0"},
14          {"id": "sc1"},
15          {"id": "sc2"}
16      ],
17      "l2_caches": [
18          {"id": "l2_0"},
19          {"id": "l2_1"}
20      ],
21      "mmu": [
22          {"id": "mmu"}
23      ],
24      "tiler": [
25          {"id": "tiler"}
26      ],
27      "job_manager": [
28          {"id": "jm"}
29      ],
30      "links": [
31          {"source_node": "s00","target_node": "s10","
               source_port": "e","target_port": "w"},
32          {"source_node": "s10","target_node": "s20","
               source_port": "e","target_port": "w"},
33          {"source_node": "s20","target_node": "s30","
               source_port": "e","target_port": "w"},
34          {"source_node": "s30","target_node": "s31","
               source_port": "n","target_port": "s"},
35          {"source_node": "s00","target_node": "mmu","
               source_port": "w"},
36          {"source_node": "s00","target_node": "sc0","
               source_port": "s"},
37          {"source_node": "s10","target_node": "sc1","
               source_port": "s"},
38          {"source_node": "s20","target_node": "sc2","
               source_port": "s"},
39          {"source_node": "s10","target_node": "l2_0","
               source_port": "n"},
40          {"source_node": "s20","target_node": "l2_1","
               source_port": "n"},
41          {"source_node": "s30","target_node": "tiler","
               source_port": "s"},
42          {"source_node": "s30","target_node": "jm","
               source_port": "e"}
43      ]
44  }
```

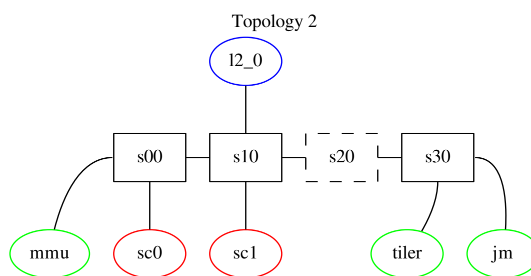

Figure B.2: Visualization of Topology 2, produced by the visualization script in combination with Dot.

Appendix C
Markup Language Tests

In this online appendix, we will show the same topology defined using five different markup languages. The languages are JSON, XML, YAML, DOT, and GML. Those topology definitions were created and used to support our decision on which markup language to use in our implementation. To avoid printing long markup language listings, we chose to make the topology files available online at http://users.student.lth.se/lak12bwi/exjobb.

EXAMENSARBETE Specification and visualisation of the interconnection network of a mobile GPU
STUDENT Björn Wictorin
HANDLEDARE Jörn Janneck (LTH), Fuad Tabba (Arm)
EXAMINATOR Flavius Gruian (LTH)

Effektivare design av nätverk inom mikrochip

POPULÄRVETENSKAPLIG SAMMANFATTNING **Björn Wictorin**

I takt med att antalet delsystem inom ett mikrochip blir allt fler, blir det nätverk som kopplar dem samman allt viktigare. I detta projekt har vi skapat ett språk och ett verktyg för att göra utforskandet av idéer på nya nätverksstrukturer enklare, snabbare och effektivare.

Tekniken som används för att tillverka mikrochip utvecklas mycket snabbt. Enligt en tumregel som ofta används inom industrin, kallad Moores lag, fördubblas antalet komponenter per chip vartannat år. Eftersom designen av ett chip därmed blir mer komplicerad, väljer man ofta att dela upp det i flera delsystem. Dessa delsystem utvecklas var för sig, ofta av olika företag, och kopplas sedan samman via ett nätverk på chippet. Utformningen av detta nätverk är mycket viktig, eftersom nätverket har stor inverkan på chippets prestanda.

Inom hårdvaruindustrin är det vanligt att man utvecklar mjukvarumodeller som simulerar den hårdvara man designar. Simulering gör det enklare att utforska nya idéer, eftersom det möjliggör snabbare och billigare prövning av nya eller förändrade koncept. Baserat på resultaten från simuleringarna kan man sedan gå vidare, och göra en hårdvaruimplementation av de alternativ som visade sig vara bäst. Dessutom kan den mjukvara som ska levereras med hårdvaran, t.ex. drivrutiner och kompilatorer, börja testas på mjukvarumodellen innan hårdvaran är klar. Vi har utfört vårt projekt på en modelleringsavdelning på företaget Arm.

Arm tillverkar världens mest sålda grafikprocessor, kallad Mali. En grafikprocessor är en enhet som genererar de bilder som visas på en dators eller mobiltelefons skärm. En grafikprocessor består av flera delsystem, och är därför beroende av ett effektivt nätverk som sammankopplar dem. I vårt projekt har vi utvecklat ett språk som kan användas för att specificera ett sådant nätverk.

I vårt projekt har vi dessutom utvecklat ett verktyg som består av två delar. Den första delen är ett tillägg till Arms mjukvarumodell av deras grafikprocessor, som möjliggör specificering av nätverkets struktur i vårt språk. Tidigare krävde förändringar av nätverksstrukturen ändringar i mjukvarumodellens källkod, men nu räcker det att göra ändringar i en separat textfil. Detta går snabbare, och kräver ingen kunskap om hur modellen är uppbyggd. Under projektet jämförde vi hur många rader som behövde skrivas för att definiera olika nätverk. I genomsnitt krävdes 48% färre rader när vårt tillägg användes.

Den andra delen av verktyget är ett program som översätter en nätverksspecifikation från vårt språk till grafbeskrivningsspråket DOT. Den översatta filen kan sedan användas för att visualisera nätverket med hjälp av tredjepartsprogramvara. Syftet med visualiseringarna är att underlätta upptäckter av eventuella misstag, där användaren råkat specificera nätverket felaktigt.