Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

GPU based IP forwarding

by

Linus Blomquist & Hampus Engström

LIU-IDA/LITH-EX-A--15/019—SE

2015-06-17
Final Thesis

GPU based IP forwarding

by

Linus Blomquist & Hampus Engström

LIU-IDA/LITH-EX-A--15/019—SE

2015-06-17

Supervisor: Usman Dastgeer & Arian Maghazeh
Examiner: Unmesh D. Bordoloi
Abstract

This thesis was about investigating if it is feasible to implement an IP-forwarding data plane on a GPU. A GPU is energy efficient compared to other more powerful processors on the market today and should in theory be efficient to use for routing purposes. An IP-forwarding data plane consist of several things where we focused on some of the concepts. We have implemented IP-forwarding lookup operations, packet header changes, prioritization between different packets and a traffic shaper to restrict the packet throughput. To test these concepts we implemented a prototype, on a Tegra platform, in CUDA and evaluated its performance. We are able to forward 28 Mpackets/second with a best case latency of 27 μS given local simulated packets. The conclusions we can draw of this thesis work is that using a GPU for IP-forwarding purposes seems like an energy efficient solution compared to other routers on the market today. In the thesis we also tried the concept of only launching the GPU kernel once and let it be infinite which shows promising results for future work.
Table of Contents

Chapter 1  Introduction ........................................................................................................... 11
  1.1  Motivation ...................................................................................................................... 11
  1.2  Purpose ......................................................................................................................... 11
  1.3  Problem statements ...................................................................................................... 12
  1.4  Limitations and assumptions ....................................................................................... 12
  1.5  Outline ........................................................................................................................... 12

Chapter 2  Background .......................................................................................................... 13
  2.1  IP forwarding ............................................................................................................... 13
  2.1.1  Routing and forwarding table .................................................................................... 14
  2.1.2  IP address prefix matching ....................................................................................... 15
  2.1.3  Data structure for forwarding table ........................................................................... 15
  2.1.4  Quality of service ...................................................................................................... 18
  2.1.5  Traffic shaper ............................................................................................................ 19
  2.2  CUDA ............................................................................................................................. 19
  2.2.1  Warps, blocks and grids ............................................................................................. 19
  2.2.2  Memory types ............................................................................................................ 20
  2.2.3  Synchronization ......................................................................................................... 22
  2.2.4  Coalescing memory access ....................................................................................... 22
  2.2.5  Streams ...................................................................................................................... 22
  2.2.6  Nvidia profiling ......................................................................................................... 22
  2.2.7  Debugging .................................................................................................................. 23
  2.3  Hardware ....................................................................................................................... 23
  2.3.1  Nvidia Jetson TK1 development board ........................................................................ 23
  2.3.2  Kepler architecture ................................................................................................... 25
  2.4  Communication with lower OSI-layer .......................................................................... 26

Chapter 3  Related work ....................................................................................................... 29

Chapter 4  Method ................................................................................................................. 31
  4.1  Evaluation ...................................................................................................................... 31
  4.1.1  Test traffic .................................................................................................................. 31
  4.1.2  Measurement ............................................................................................................. 31
  4.2  Implementation .............................................................................................................. 32
  4.2.1  Forwarding table ........................................................................................................ 32
  4.2.2  IP forwarding packet processing on the GPU ............................................................ 34
  4.2.3  Align memory access ................................................................................................. 35
  4.2.4  Coalescing ................................................................................................................ 35
  4.2.5  Kernel launch evaluation ........................................................................................... 36
  4.2.6  Infinite kernel ............................................................................................................ 37
  4.2.7  Quality of Service ..................................................................................................... 38
  4.2.8  Traffic shaping .......................................................................................................... 40
  4.2.9  Memory storage of forwarding table ......................................................................... 42
  4.2.10 Optimal number of threadgroups ............................................................................. 43
  4.2.11 Final prototype .......................................................................................................... 44

Chapter 5  Results .................................................................................................................. 45
  5.1  Throughput ..................................................................................................................... 45
  5.2  Shaper performance ....................................................................................................... 45
  5.3  Trie lookup ...................................................................................................................... 46
  5.4  GPU utilization .............................................................................................................. 46
  5.5  GPU execution count ..................................................................................................... 47
Preface

General purpose computing on graphics processing units (GPGPU) has grown dramatically in recent years and today it often offers a higher performance per both watt and cost compared to CPU processors. GPUs are massively parallel processors with hundreds or thousands of cores that can efficiently solve problems that can be parallelized in a suitable way. During the spring of 2015 we were assigned by Ericsson to build a prototype that should be able to perform IP forwarding on a GPU, a task normally done by a specialized network processing unit (NPU).

Linköping, Sweden, June 17

Linus Blomquist
Hampus Engström

Acknowledgements

We would like to thank Johan Moe and Usman Dastgeer at Ericsson for letting us do this thesis work and for giving us all the support that we needed to finish it. A special thank also to our examiner at Linköping University Unmesh D.Bordoloi and our supervisor Arian Maghazeh.

Linköping, Sweden, June17

Linus Blomquist
Hampus Engström
Chapter 1  Introduction

1.1  Motivation

In 2007 Nvidia launched CUDA (compute unified device architecture), a new programming language for their GPUs aimed to make GPGPU (General purpose computing on graphics processing units) programming easier for developers. CUDA brought GPGPU programming to a new level since it was not necessary anymore to map all data to images to be able utilize the GPU for general purpose programming (Sanders, J, Kandrot, E, 2012). The CUDA programming is done in C/C++ and most of the code looks like an ordinary program which makes it much easier to get started with. Example of areas where CUDA have successfully been adapted are medical imaging, computational fluid dynamics and environmental science (Sanders, J, Kandrot, E, 2012). What all these areas have in common is that they require processing of large amounts of data that could be served effectively by the CUDA parallel execution pattern.

IP forwarding is about routing packets between different hosts on a network. It could be performed on either a hardware router using customized hardware for routing purposes, a software router running on a regular CPU or on a programmable NPU (network processor). New standards and types of packets have been introduced to adapt to market demands since the start of internet. For that reason it is important that routers can adapt to these changes to avoid the need of replacements with newer routers. Hardware routers offers the highest performance but the least flexibility. NPUs are both expensive and require special knowledge to be programmed. The main advantage of them are that they also offer a high throughput. The software routers on the other hand are more flexible and are easier to adapt to changes in network standards but offer less throughput compared to the other alternatives (Shuai Mu, 2010) (Zhu, 2011).

As IP forwarding could process each packet individually it is interesting to find out if this can be done in parallel on a GPU. GPUs are, in comparison, cheaper than NPUs and easier to program due to their more general and known architecture. They also offer higher computational power compared to software routers.

1.2  Purpose

The purpose of this thesis is to investigate if it is feasible to implement an IP forwarding data plane on a GPU. According to Ericsson the IP forwarding data plane can be divided into three separate parts: classification, editing and traffic management. In the first part, classification, we will implement a forwarding table lookup for routing packets. In the second part, editing, we use the result from the forwarding table lookup to edit the layer 2 headers of the packets. For traffic management we will focus on Quality of Service to be able to prioritize packets differently and traffic shaping in order to be able to control the output bandwidth of our application. To test these concepts on a GPU we will implement a prototype in CUDA and evaluate its performance.
1.3 Problem statements

In the end of our thesis work we shall have built a working prototype that performs key parts of the IP forwarding data plane. When this is accomplished these questions need to be answered:

- Is it feasible to use GPUs to accelerate the IP forwarding data plane for software routing?
- How do you best utilize the GPU to accelerate the IP forwarding data plane?
- Which properties of a GPU are most important for routing applications?
- What parts of the IP forwarding data plane are not suitable for GPU computation?

1.4 Limitations and assumptions

Here we state different limitations for our work on this thesis:

- We will limit our thesis to focus on the IP forwarding data plane to be able to maximize the performance of this part. We will also only handle IPv4-packets in our prototype.
- Our IP forwarding prototype will be implemented in CUDA.
- The forwarding table is going to contain a few hundreds of entries and shall never contain more than 10000 entries.
- We assume that all incoming packets to the IP forwarding data plane are correct and corrupt packets are dropped by lower levels.
- The forwarding table does not have to support dynamic updates.
- Our prototype does not need to be able to handle terminating packets.

1.5 Outline

The report starts by giving a background to relevant areas that are connected to this master thesis. These parts will introduce basic knowledge about how IP forwarding works and present earlier research on implementing parts of forwarding data plane on a GPU. This is followed by a brief introduction of CUDA and a short specification of the platform on which we are going to implement our IP forwarding data plane software. There is also a discussion about how to retrieve packets to our application.

In Chapter 4 we are describing our practical work in the project and show how we went from a first primitive prototype to more advanced versions. Here we will explain key features without going into code details.

Finally, in Chapters 5, 6, and 7 we will present our results and discuss these results. There are several tradeoffs in the implementation that are up to the end user to decide according to what best suit their needs. We will explain these features in order to make it possible for an end user to make these decisions. We will also give some suggestions about future work in the area.
Chapter 2  Background

Here we will present and explain general concepts and theory needed for our thesis. We start by presenting the basics of IP networking, in Chapter 2.1, to give a background for the reader. Details regarding data structures and lookup algorithms needed to perform the actual IP forwarding will be described. We then introduce the Quality of Service and shaping which is also done in the IP forwarding data plane.

Previous work on implementing IP forwarding on GPUs, used to retrieve ideas from, will be described in Chapter 3. Further in Chapter 2.2 we explain CUDA and its most relevant concepts that we will be dealing with in our thesis. In Chapter 2.3 we will give a short presentation of the hardware platform we used for our prototype. Lastly, in Chapter 2.4, we will give a short description of possible techniques of retrieving packets to our prototype.

The information in this chapter is used to motivate the design choices we made when developing our prototype and are referred to later in Chapter 4.

2.1  IP forwarding

Figure 1, payload encapsulation.

In the OSI-network model the network layer is the third layer, stacked between the transport and the link layer. Our work is mostly done inside this layer 3 but we are also affected by the lower layer 2 (we assume it is Ethernet based). Figure 1 illustrates how the payload of a packet coming from layer 4 is encapsulated into an IP packet at the third layer, and then to an Ethernet frame at layer 2.

The internet protocol defines a set of rules for how packets traverse an IP network and how they are routed. In short, the steps taken to transport a payload from a source to a destination can be described in the following way:

At the source the payload is encapsulated inside an IP packet by appending an IP header to it. The IP header contains vital information such as destination and source IP address so that the network can figure out where the packet shall be delivered. In the header there is also other information such as length, protocol numbers and checksums. By querying its routing table the source then determines which
of the nodes that it is directly connected to is most suitable to be the next-hop for the packet (Weidong, 2007, p. 2). This will be described in more detail in Chapter 2.1.1. The complete IP packet is then encapsulated again, now into an Ethernet frame by appending an Ethernet header to it. The Ethernet header contains source and destination MAC-addresses which correspond to physical Ethernet interfaces of the source and next-hop node (Weidong, 2007). Using this information, the NIC can then handle the physical transmission of the frame to the next-hop node.

When the next-hop node receives the packet it strips off the Ethernet header. The node then examines the IP header to determine the next hop. When this is performed, the node encapsulates the packet in a new Ethernet frame and sends it to the next node. This is repeated by each node on the path until the destination node receives the packet. The destination node recognizes that the destination IP address of the packet is the same as its own and therefore determines not to forward the packet and sends it up to the transport layer instead.

Worth noticing is that IP is a connectionless and best effort protocol which means that it treats every packet individually and does not promise any level of performance. Packets can be dropped, delayed and reordered and a set of packets with the same destination and source can even take different routes through the network. This however can affect the performance and dropped or reordered packets can severely reduce the throughput because of how it affects the transport protocol used. Neither TCP nor UDP, which are the most commonly used packet protocols on the internet today, are able to tackle packet reordering perfectly. It depends on the usage of these protocols how large impact packet reordering will have but it is clear that the effect of packet reordering could not be negligible (Przybylski, 2005).

There are some additional adjustments and verifications in the IP header that needs to be performed when doing IP forwarding. The TTL (time to live) field needs to be decremented before the packet is processed and if it reaches zero the packet should be dropped. This is to prevent packets from circulating in the network for eternity if there is a misconfiguration somewhere creating a routing loop. There is also a packet length field that should be compared to the actual length of the packet and a checksum to detect transmission errors. When the TTL field is changed the checksum also needs to be updated accordingly. Luckily this operation is cheap, one unit of decrement in the TTL requires one unit of increment in the checksum (Weidong, 2007).

### 2.1.1 Routing and forwarding table

The routing table stores the information needed to forward packets to the correct destination. The table could be static and entered manually by a technician when the system is configured which is what we assume in this thesis work. It could also be dynamically updated by a routing protocol that autonomously learns the best routes and adapts to changes in the network (Sangoma, 2013). Each routing entry contains a number of fields that varies slightly between implementations but a few are present in most:

- A network prefix saying which packets depending on their destination IP address shall use this route. For example all packets with the destination IP address starting with 0001 shall use this route with the exception if there exist a more specific route. More about this in Chapter 2.1.2.
- A subnet mask saying how many bits in the network prefix need to be checked in order to match this route.
- A gateway IP address to which packets matching this route should be sent.
- An interface index saying on which interface the destination gateway resides.

This routing table is also often called a routing information base (RIB).
From the RIB a forwarding table, called forwarding information base (FIB), is generated which contains much of the same information but compressed and limited to the most essential parts needed to perform fast lookup operations (Weidong, 2007).

### 2.1.2 IP address prefix matching

An IP address consists of a prefix followed by a host id. The prefix symbolizes the network and the host id the specific host in that network. Previously there was IP classes A, B and C with prefix length 10, 16, and 24 bits respectively but they are now outdated due to the introduction of classless intra-domain routing (CIDR). This change was introduced 1993 and now the prefix could be of arbitrary length which gives a higher flexibility in the number of hosts that a network could contain. For example a network of 500 hosts could have a prefix of 9 bits which support 512 hosts instead of a prefix of 16 bits that supports 65536 hosts.

Before CIDR was introduced there were no overlap in entries in the forwarding tables which made the routing easier. For example if one network has the prefix of 10001000 and another has the prefix of 100010001 they both have the same first 8 bits. Further the second network have another prefix bit which makes it more specific. After CIDR a forwarding lookup therefore have to check each IP address bit by bit instead of steps of 10, 16 and 24. The process of finding the most specific route in the forwarding table is called longest prefix matching (Weidong, 2007). It is used to choose among several matching rules in the forwarding table.

### 2.1.3 Data structure for forwarding table

Since one lookup in the forwarding table is performed for every single packet traversing a router the time complexity of that lookup is directly influencing the throughput. In our case the memory size also has a great impact since it determines which type of memory we are able to use.

#### 2.1.3.1 Array

The most naive way to represent the forwarding table would be a simple unsorted array storing key-data pairs. The time complexity for a linear search would be O(n) in the worst case, where n is the total number of entries to the forwarding table. This also means that the search time would increase proportionally to the number of entries in the forwarding table making it a bad candidate for larger tables. The space complexity would be O(n) since each element would simply be added as an extra entry to the end of the array. The main benefit for an array representation would be its simplicity and low memory usage.

#### 2.1.3.2 Bitwise Trie

A better solution is to store the forwarding table as a Trie data structure. In this chapter we will describe this data structure in detail and explain different ways to optimize it. A Trie tree is similar to a binary tree but the key does not need to be stored in each node. Instead the edges in the Trie correspond to the key implicitly. The key could consist of any type of symbol, if it is bits the Trie is called a bitwise Trie and every node has up to two children. A lookup in the Trie is performed by traversing the tree, always choosing the next node according to the next element in the search key. The data can either be stored directly in the nodes or as a pointer to the actual location.
An example of a bitwise Trie can be seen in Figure 2 where keys of 2-bits length are used. The keys in this Trie are 00, 01 and 11 which corresponds to the nodes A, B and C respectively.

**Representation**

The way each node is represented in the Trie have a large impact on the amount of memory the Trie will require. Every node needs to store information so that they are able to refer to their children which is normally done by using pointers. However this requires a lot of memory since each pointer on a 64-bit system takes eight bytes and each node in a bitwise Trie has two children. By storing all the nodes in an array it is possible to replace the pointers with indexes instead which requires less memory (Joanis, 2009). The size needed for this index would be $\log_2(n)$ bits where $n$ is the size of the array.

Another way to decrease the child references within a node is by only storing a reference to the first of its two children (Nilsson, S, Karlsson, G, 1998). This is possible if all the sibling nodes always are stored in consecutive memory and if every node has either none or two children. These requirements can be solved by introducing dummy nodes and using path compression, which are going to be discussed later.

Finally each node also needs to contain a data field. This field would also benefit of being represented as an index to an array rather than storing the actual data in order to reduce each node need of memory.

**Time and space complexity**

In order to find the data corresponding to a key a Trie lookup will require traversing as many nodes as the key length, which is also equal to the depth of the Trie. Since the search time is constant compared to the number of elements in the Trie its time complexity is $O(1)$ which is much better than for the linear search. There are two major ways to reduce the depth of the Trie and that is by using path and level compression.

In the worst case, where all key entries differ as much as possible which would require the largest tree structure, the Trie would need $2n - 1$ nodes where $n$ are the number of entries to the table (Nilsson, S, Karlsson, G, 1998).
Path compression

Path compression is used to merge all nodes in the Trie that only has one child with their child node. Since there is no choice of path when traversing these nodes they can easily, without affecting the rest of the Trie structure, be merged together. To be able to achieve this a skip value has to be introduced (Nilsson, S, Karlsson, G, 1998). This skip value is stored in every node and represents the number of bits in the key the edge to their parent represent. Normally, without path compression, this value is one which means that the edge represent one bit in the search key. With path compression this value is between 1 and 32.

If we consider the first Trie example again, as can be seen in Figure 3, we can see that one branch, the branch to the right from the root, only contains one leaf. In this case all nodes starting with the first bit equal to one, could be compressed to one node. In this example the compressed node will contain the skip value of two representing the two bits in the search key that lead to the data C. All other nodes will have the skip value of one.

By using path compression the Trie demands far less memory than before. For a full tree it would demand the same amount of memory as a non-optimized Trie but less memory in all other cases. The reason for that is because a full tree consist of all possible nodes which means that all skip values are always going to be one, leading to no possible optimizations.

Figure 3, Trie path compression.
Level compression is used to decrease the depth of the tree by increasing the number of branches from the nodes (Nilsson, S, Karlsson, G, 1998). By applying this technique to the Trie the total time complexity would decrease as the number of traversed nodes, in a forwarding lookup, would decrease. The intent of this optimization is to recursively compress the densely populated parts of the tree. In more detail all $x$ highest complete levels of the Trie are replaced by nodes of degree $2^x$. In Figure 4 you can see an example of such a compression.

### 2.1.4 Quality of service

Besides doing IP forwarding, the IP forwarding data plane can also control quality of service (QoS) for the packets. Depending on the type of a packet it can be assigned to different priority levels (Weidong, 2007, p. 5) in order to meet its demand. For instance VOIP (voice over IP) packets could be prioritized to have a low consistent latency in order to keep the call quality up and remove jitter.

Quality of service could be realized by using priority queuing that sorts packets of different priorities into separate queues (Babiarz, 2006). The packets in the higher priority queues should then be prioritized over the packets in the lower priority queues in the routing. However, care must be taken to prevent low priority packets from suffering from starvation. To decide which priority queue a packet should be placed in, the DSCP (Differentiated Services Code Point) field in the packet header is used.
2.1.5 Traffic shaper

In order to utilize the network efficiently or to meet agreements between different networks the outgoing traffic could be passed through a traffic shaper. The purpose of traffic shapers is to control the flow of packets according to a traffic profile which could specify desired properties such as average throughput and maximum burst length. If the shaper receives more incoming packets than it is able to buffer, it drops the packets according to its dropping policy (Blake, 1998, p. 16).

A traffic shaper can be implemented by a token bucket that generate tokens at a specified rate (P.Tang, 1999). A packet can only be transferred if there are enough tokens to match the packet. Each token corresponds to a number of bytes which means that larger packets requires more tokens. After sending a packet the available tokens is decreased by the number of tokens used by that packet.

2.2 CUDA

CUDA was released in the beginning of 2007 (Sanders, J, Kandrot, E, 2012, p. 8) and is a programming language for Nvidia GPUs. The aim of this language is to make GPGPU programming easier. It removes the need for having to map all data into images as programmers would have had to do before when programming GPUs.

The main benefit with CUDA compared to other languages is that it is generally much easier to get started with. The most common workflow could be described in a simplified way as follows:

1. CPU code allocates space and uploads data to the GPU. The CPU then start a GPU kernel with a specified number of threads.
2. The GPU runs the kernel and in the meantime the CPU either do something else or waits for the kernel to finish.
3. When all threads on the GPU are done processing the CPU can synchronize with the GPU and copy the result back to the CPU memory. The GPU kernel then terminates.

The following parts of this chapter are going to describe important concepts once working with CUDA. Helpful tools are also described in the end of this chapter.

2.2.1 Warps, blocks and grids

Threads on the GPU, that you launch in order to execute parallel algorithms, are divided into blocks and grids. There are one or several threads in each block and one or several blocks in each grid. In order to fully utilize the GPU you need a good balance between these two settings.

Further threads are grouped into warps, that have a size of 32 threads on all current CUDA enabled Nvidia GPUs (Harris, 2015a) (Nvidia, 2015b), where each thread in the same warp is executed in parallel on the same lines of the code. For example if there is an if-else statement all threads, within the same warp, are forced to follow the same branch. This could lead to the need of executing several branches in every thread when threads want to execute divergent parts of the code. Each thread is then only committing the operations within the branch they wanted to follow. For this reason it is desirable to keep the branching factor as low as possible for GPU functions.

Occupancy is a key concept when determining a proper grid and block size (Nvidia, 2015b). The definition of occupancy is the ratio between the actual numbers of active threads on a multiprocessor compared to the theoretical maximum number of active threads on the multiprocessor. There could be several active blocks on the same multiprocessor simultaneously if there are hardware support for it. For
example if a multiprocessor have support for 768 threads, as in compute capability version 1.1, there could be three blocks of 256 threads each active at the same time.

The number of threads per block have to be a multiple of the warp size since a block is able to process a multiple of warp number of threads simultaneously. If a block is not filled with working threads that means that hardware resources are wasted.

Another constraint when you are choosing grid and block size is that the threads in one block have limited memory resources in form of registers and shared memory. If threads are using a lot of memory resources there will be a fewer number of threads that will fit into that block before register spilling start to occur (Nvidia, 2015b).

The number of blocks used should be at least as high as the number of multiprocessors that are available in order for each multiprocessor to at least have one current block to process (Nvidia, 2015b).

### 2.2.2 Memory types

There exist several types of memory that could be used by the GPU in CUDA. Some of them are used implicitly like cache memories and registers and some have to be explicitly stated by the programmer. In order to achieve high performance in a CUDA application it is a must to have a deep knowledge of the different memory types.

If an input parameter to an instruction resides in an off-ship memory each instruction that includes accessing that parameter takes 200-400 clock cycles to complete, for compute capability version 3 and higher, that could be compare to 11 clock cycles if there were no access to the off-ship memory (Nvidia, 2015h). If a lower compute capability version is used the difference is even higher. These numbers shows the importance of good trade-offs in memory usage. These are hidden by ILP (instruction level parallelism) if there are enough warps that are ready to execute while the first warp is waiting for its results.

#### 2.2.2.1 Global memory

Global memory (Sanders, J, Kandrot, E, 2012) is the main memory for the GPU and can vary in size between a few megabytes to several gigabytes for high end graphics cards. The memory typically resides on fast GDDR chips on dedicated graphics cards or in a reserved part of the CPUs main memory for integrated GPUs.

In order to use global memory the programmer needs to explicitly allocate and free it with calls to `cudaMalloc` and `cudaFree`. Data can then, by the CPU code, be copied between main memory and global memory with calls to `cudaMemcpy`. Obviously this introduces a lot of copying back and forth between GPU and the CPU which takes time. Besides this the global memory is relatively slow compared to most other memories the GPU can use so it should be utilized as little as possible.

Unified memory (Harris, 2015b) is a way of handling global memory that was introduced in CUDA 6 and is used to hide the complexity of explicitly copying memory between the CPU and GPU. It is easy to set-up as it only needs to be allocated with a call to `cudaMallocManaged`. After that both the CPU and the GPU can use the memory as it was residing in main or global memory respectively. The data is copied between the GPU memory and the CPU memory back and forth on demand and is performed without the programmer’s involvement. The downside of this usage of memory is of course that the CUDA driver and runtime system never can do as good predictions as the programmer of where the memory will be used next. Therefore explicit copies may still be necessary in order to get the highest performance.
Each read from global memory is on either 32, 64 or 128 bytes depending on the compute capability version of the GPU. This means that memory transfers will always consist of these specified number of bytes and if only parts of it will be used the rest will be wasted memory transfers. Moreover the transfers will always be aligned to the cache line size so in the worst case the read of a misaligned word will result in the transfer of two cache lines (Nvidia, 2015b). For a cache size of 32 bytes a 16 bit misaligned read could in the worst case result in a degradation of 32 times if only two bytes were actually needed.

How accesses to global memory are going to be cached depends on the compute capability version. For architectures that builds on a compute capability lower than 3, data accesses from global memory are going to be saved both in the L1 and L2 cache memory. Regarding architectures that builds on a compute capability version higher than 3, the result are going to be saved in the L2 cache only. If the architecture supports a compute capability version of at least 3.5 the default behavior is still to only cache data in the L2 cache but where the programmer can alter this behavior to cache in L1 cache as well. The L1 cache consists of cache lines of 128 bytes and the L2 cache consist of cache lines of 32 bytes.

2.2.2.2 Pinned memory

Pinned memory (Sanders, J, Kandrot, E, 2012) is memory allocated on the CPUs main memory. It is fixed in its location on the physical memory chip and will never be swapped out to disk by the operating system. Too much usage of the pinned memory would result in that the CPU runs out of main memory. Since the memory could not be swapped out of the main memory it is safe to use pinned memory directly from the GPU over the PCI Express bus without CPU involvement. Using pinned memory from the GPU is also referred to as zero-copy memory and can for applications that only uses a memory location once give a great performance increase. If the same memory location is accessed several times it introduces the need of transferring the data over the PCI express bus several times which is avoided when using global memory on a dedicated GPU (Sanders, J, Kandrot, E, 2012, p. 223). To use zero-copy memory the memory needs to be allocated by the CPU with a call to cudaMallocHost.

For integrated GPUs the performance of zero-copy memory is the same as for global memory since they both reside on the same physical memory. Using global memory will only introduce an extra memory transfer between locations on the same memory. For that reason it is always better to use zero-copy memory on integrated GPUs (Sanders, J, Kandrot, E, 2012, p. 223).

2.2.2.3 Local memory

Local memory is a private memory for one single thread that are deleted once the thread terminates. It resides in the same location as global memory and is used when the compiler determines that the local variables will not fit in the registers. Because of this it is important not to overuse local variables since the performance of local memory is only a small fraction of the registers (Nvidia, 2015b).

2.2.2.4 Shared memory

Every thread block has access to a so called shared memory (Sanders, J, Kandrot, E, 2012) that is memory shared between all threads within the same block. No threads outside this block are able to retrieve data in this memory. In cases were different threads, in the same block, are operating on the same data it is useful to move the data to the shared memory that are hundreds of times faster to access compared to the global memory (Nvidia, 2015a). The reason for that is because the shared memory is, together with cache and registers, the only memory residing on the GPU-ship.

2.2.2.5 Texture memory

Texture memory resides in global memory and is a read-only memory specialized for texture access patterns (Sanders, J, Kandrot, E, 2012, p. 115). Texture memory is cached to the on-chip texture cache (Nvidia, 2015a) which can increase memory performance for applications with suitable access patterns.
2.2.2.6 Constant memory

Constant memory (Sanders, J., Kandrot, E., 2012, p. 95) is another type of read only memory residing in global memory. It is limited to 64 kB in size but are cached to the on-chip constant memory cache (Nvidia, 2015a) which can give increased performance. When a thread access a memory location it also broadcast the data to its half warp (one warp is typically 32 threads) closest neighbors. The memory is initialized by allocating it from CPU code and then calling cudaMemcpyToSymbol to fill it with data.

2.2.3 Synchronization

Inside a GPU kernel every thread within the same block can synchronize with each other using the command __syncthreads(). Then all threads inside the same block would wait until all threads have arrived to the same line of code in the program. The only way to synchronize threads between different thread blocks is to terminate the kernel and synchronize them via the CPU or keeping a program state in the global memory. Threads within the same warp does not have to explicitly state __syncthreads() since they are always running synchronized by default.

2.2.4 Coalescing memory access

In order to use the global memory effective you need to consider something called coalescing (Nvidia, 2015b). This means that by operating on data in consecutive memory, from different threads, you need less access to the global memory. The first thread are going to fetch a cache line, consisting of either 32 or 128 bytes, from the global memory. If this line, which now are saved in the cache, contains the data that the other threads need they can access them through the cache memory instead of the global memory.

For data that cannot be accessed consecutive by different threads it is preferable to cache the data in L2 cache compare to the L1 cache because of its smaller cache lines. This access pattern will result in retrieving 32 bytes instead of 128 bytes for each read in the memory. For example if a thread wants to read 8 bytes from the global memory 75% of the retrieved data transfer would be wasted compared to 94% if the L1 cache were used. However whether this is possible to specify depends on the GPU architecture.

2.2.5 Streams

CUDA streams (Sanders, J., Kandrot, E., 2012, p. 185) are used to build queues of GPU operations to be executed. The GPU operations are executed in the same order as they are added to a stream but operations from different streams can be interleaved as the scheduler finds most effective. By using streams the GPU can also process its tasks in a pipelined fashion, doing asynchronous memory transfer at the same time as it run kernels. Another benefit of streams is that the CPU spends less time waiting for the GPU since it queues all operations and then can move on to do other things.

2.2.6 Nvidia profiling

A large part of our thesis work were to optimize our application. Nvidia has developed a tool for profiling CUDA applications called NVVP (Nvidia Visual Profiler) (Nvidia, 2015g) that we found helpful. It supports profiling both the CPU and the GPU parts of the code and can by that be used to identify bottlenecks of the complete system. This tool has both command line and a graphical interface and comes as a part of the CUDA-toolkit. This tool was used through the whole development and was also used to present our results at the end of the report.
2.2.7 Debugging

Nvidia also provides CUDA-GDB (Nvidia, 2015c) which is a tool to debug both CPU- and GPU code. It has both a terminal and a graphical interface very similar to the standard GDB with some extra commands corresponding to the extra GPU functionality. In order to identify memory errors on the GPU there is also a tool called CUDA-memcheck (Nvidia, 2015d) that is able to read memory error codes from the GPU execution. These debugging tools are not mentioned further in the report but were of big help when we were struggling with errors during the development.

2.3 Hardware

The choice of GPU and hardware platform has greatly affected many of the design choices we have been forced to make for our prototype. In this chapter we give a short description on the Nvidia Jetson TK1 development board and a more detailed presentation of the Kepler GPU architecture which the GPU are based upon.

2.3.1 Nvidia Jetson TK1 development board

As a hardware platform, for our prototype, we have been given an Nvidia Jetson TK1 development board. This platform consists of a Tegra SoC (system on chip) containing a quad core corex-A15 ARM processor and a Kepler GPU consisting of one streaming multiprocessor. Besides this it has a gigabit Ethernet port, a USB 3.0 port and 2 GB of RAM that is shared between the CPU and GPU (Nvidia, 2015f).

The development board has compute capability version 3.2 (NVidia, 2015i) which means that it is not able to use the read-only data cache on the streaming multiprocessor that require compute capability version of at least 3.5 (Nvidia, 2015h). The GPU on the development board has a L2 cache used to cache global memory accesses and a L1 cache used to cache accesses to the local memory (Nvidia, 2015h). The constant memory has a size of 64 kB and is cached in an 8 kB part of the L1 cache.

Jetson TK1 comes preloaded with a modified version of Ubuntu called Linux for Tegra and is quite easy to get started with. For routing applications the board is somewhat limited because it only has one Ethernet port, but more can be added either by USB 3.0 or miniPCI Express network cards. Compared to ordinary x86 based PCs the board is both cheap and uses less power but it also offers less performance and memory. The ARM processor is developed to be power efficient and is aimed for mobile applications. The GPU is also a low power integrated model without dedicated memory and the expansion possibilities are limited. The ARM architecture, although it has gotten a lot more support in recent years, can also be harder to find good libraries and tools for compared to the more common x86 architecture.
<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared memory and L1 cache</td>
<td>64 kB shared between them</td>
</tr>
<tr>
<td>L2 cache</td>
<td>131 kB</td>
</tr>
<tr>
<td>Constant memory</td>
<td>49 kB</td>
</tr>
<tr>
<td>Constant memory cache</td>
<td>8 kB</td>
</tr>
<tr>
<td>Max # resident threads per SM</td>
<td>2048</td>
</tr>
<tr>
<td>Max # resident warps per SM</td>
<td>64</td>
</tr>
<tr>
<td>Max # resident blocks per SM</td>
<td>16</td>
</tr>
<tr>
<td>Max # resident threads per block</td>
<td>1024</td>
</tr>
<tr>
<td>Global memory</td>
<td>2 GB</td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
</tr>
<tr>
<td>Compute capability version</td>
<td>3.2</td>
</tr>
<tr>
<td>Max # registers per SM</td>
<td>32k</td>
</tr>
<tr>
<td>Max # registers per thread</td>
<td>255</td>
</tr>
<tr>
<td>Max # registers per block</td>
<td>32k</td>
</tr>
</tbody>
</table>

**Table 1**, specification summary of the development board (NVidia, 2014b; Nvidia, 2015h).
2.3.2 Kepler architecture

In Figure 5 you can see the architecture of the latest Kepler generation of Nvidia streaming multiprocessors. Every Nvidia GPU with a Kepler architecture consist of one or several of these streaming multiprocessors.

The streaming multiprocessor consist of 192 CUDA cores and has four warp schedulers that allows four warps to be executed concurrently. Moreover, the scheduler can issue two independent instructions per each warp, in each cycle (ILP of 2 for each warp). Each warp scheduler tries to do this in such a way so it maximizes the utilization of the GPU. For example, it may swap out a warp that is waiting for memory with another warp that is able to continue directly. The on-ship memories are the shared memory, registers and a data cache used by the constant and texture memory.
The L1 cache and the shared memory resides in the same physical memory that are shared between them. The CUDA API supports different splits for partitioning this memory (Nvidia, 2015h). This memory is shared by all threads on the same streaming multiprocessor (Nvidia, 2015h).

2.4 Communication with lower OSI-layer

A task of a working router is to retrieve packets from an interface and also output the packet to another interface once the processing is completed. In this thesis our task is to focus on the processing part but even if we are not going to focus on the packet retrieving and outputting to the network it is still interesting to consider. We also need some way to deliver packets to our application in order to be able to test it correctly.

One solution could be to use a kernel module like pf_ring, netmap or DPDK (Data Plane Development Kit) but a small investigation regarding these showed that they lack the support for our development board due to its ARM processor. Another alternative is to use RAW sockets. Already on the link layer, on the OSI model, these sockets are able to fetch and place packets. In theory this should be a quite good solution because if we fetch the packets early in the packet processing chain, we do not have to involve the Linux kernel more than necessary.

Figure 6, example usage of GPUDirect (NVidia, 2014c).

Nvidia has introduced a technique called GPUDirect (Nvidia, 2010). GPUDirect is a way to reduce the CPU involvement in GPGPU programs. Figure 6 shows a sketch of a way for a GPU to directly communicate with another GPU over a network without the CPUs involvement. According to Nvidia it should be possible for the network card to directly access GPU memory, using RDMA (remote direct memory access) over the PCI Express bus. This would dramatically reduce the CPU overhead (Nvidia, 2015e). For our application this could be especially useful if we had used a dedicated graphics card with a separate memory since it could remove the need to pass the packets through the system memory. In the case of our development board, which has a shared memory between the CPU and the GPU it would be less useful and the network card on our development board also lacks support for this feature.
Our decided solution was to use capture files that are files containing real packets. A benefit of using this method to simulate traffic is that we get rid of the demands to connect physical devices, with Ethernet cables, to our development board. Now instead we could run everything on one single machine which shortens the setup phase and make the development and debugging easier. It also become easier to benchmark our application and compare solutions when we can have full control over the traffic scenarios.
Chapter 3   Related work

During our research we found a number of papers relevant to IP forwarding using GPUs that we used mainly to get ideas of good implementations. Because of the lack of comparable researches we were not able to compare our research to theirs.

In (Tsung-Hsien Li, 2013) the authors propose to perform forwarding table lookups in a Trie data structure that is both level compressed and path compressed. Their work however was based on a conceptual architecture where the packet headers could directly be transmitted from the NIC to the GPU memory without CPU involvement. The GPU then performs lookup operations and sends the result back to the NIC. In the real test setup this was not the case and since the memory transfer between the CPU and the GPU memory has to go through a PCI Express bus, which became a bottleneck, this time was excluded from the measurements. During tests on different graphic cards they achieved between 1,3 and 3,6 billion lookups per second in a forwarding table of more than 350 thousand entries and they achieved a latency of 7,68 µs per packet. The tests were performed on randomly generated packets and a couple of different public available routing tables that gave different sizes of the Trie structure due to different strides between the IP addresses. A larger stride between IP addresses in the forwarding table means a shallower tree that is faster to traverse. Since they only send IP addresses to the GPU instead of whole packets in this research they are able to achieve an impressive number of lookups per second. We aim to work with the whole packets on the GPU instead of just using IP addresses and are including other operations besides the forwarding lookup operations such as header editing. Therefore our results are not comparable with this research paper.

In (Zhu, 2011), the researchers used simulated hardware where the CPU and GPU share the same GDDR5 memory. In this way the transfer time between the GPU and CPU could be ignored but still they faced bandwidth limitations when communicating with the NIC. In order to avoid packet reordering they suggested to use a delay commit queue. This unit only commit, the processed packets, after the packets with a lower index (packets arrived earlier) have been committed. One additional problem with IP forwarding on GPUs discussed in the paper is the throughput vs latency problem. In order to achieve a large throughput the GPU needs to be served with enough packets to fully utilize its hardware resources. This means that the CPU needs to buffer up enough packets and send them in bunches to the GPU. The GPU then distribute the workload between its threads. This way of working introduces the delay where each packet in the buffer will have to wait before being transmitted until the buffer contains enough packets. Smaller buffers introduces smaller latencies but give a lower GPU utilization and therefore a smaller throughput. In order to achieve a good IP forwarding performance you need to find a good tradeoff for this problem. Finally in order to achieve align memory accesses from the GPU they set their variable sizes to be larger than necessary to fit the bus bandwidth. This mechanism requires extra memory space but was worth it since it increased the total performance of the program. The report states that using their simulated GPU hardware, compared to a previous implementation on existing CPU and GPU hardware, they are able to improve the throughput by a factor of five.

Another practically oriented paper we found was (Shuai Mu, 2010) where the authors investigate the possibility to use GPUs to increase software routers performance. As an experiment they chose to offload both the forwarding table lookups and perform network intrusion detection on a GPU instead of on a CPU. To do this they used a dedicated Nvidia GTX280 Graphics card to which they copy the
packets using the PCI Express bus. The GPU then does intrusion detection by a deep packet inspection that applies a text filter to the whole packet data. After intrusion detection the GPU does the forwarding lookups. Comparisons to their own CPU solution shows that a GPU can be used to speedup routing applications for at least one order of magnitude. Further this report also found out that the main bottleneck for routing functionality on a GPU is the memory transfer between GPU and CPU. According to this research the throughput of the application would become 5 times faster if the packet transfer time between CPU and GPU would be ignored. This transfer time could be hidden using streams but would still increase the latency for each incoming packet. From a single packet point of view the total processing time would be the same even if the GPU was able to hide its idle states with computations on other packets. Many packets in an IP network have a strict deadline which means that the latency each router introduces along its path increases the risk for the packet to miss its deadline.

PacketShader is a GPU accelerated software router framework that was first presented in (Han, 2011). This framework includes a fast and highly optimized I/O engine for packets which possibly could be used to complement our prototype in this thesis in order to get a more fully functional router. The main thing that they have done in order to improve the I/O engine by handling packets in batches instead of as single packets which decrease the per packet overhead. The authors evaluated this framework by building four prototypes on top of it. The most related prototype to our work is the prototype that performs ipv4 forwarding. The only thing that was GPU accelerated within this prototype was the lookup operation, all remaining operations were carried out by the CPU. The conclusion after evaluating this prototype is that its performance is related to the number of memory accesses. Basically more memory accesses gives a more effective GPU solution compared to the CPU. The GPU, with its large number of threads, is able to hide the memory latency which the CPU is not able to do in as high degree. This router support a throughput of 39 Gb/s and was limited by the I/O packet engine. By supporting this throughput, according to the authors, this router was the first software router to support multi 10G throughput. The downside of this router is that it has an end to end latency of 140-260 µS for ipv4 packets which is fairly high comparing to other routers.

Another interesting paper is (Kalia, 2015) which investigate if the advantage of using GPUs for packet processing really is as effective as earlier studies has shown. In this research they developed their own prototype and compared it to (Han, 2011). According to their research the GPU has outperformed the CPU in earlier researches not because of the problem itself, but because of how the problem has been expressed. Neither the power consumption of using a GPU accelerated IP forwarding over a CPU based one would be improved. The reason for that is because the GPU version of IP forwarding, used in this research, required almost as many CPU cores as the application that only uses the CPU. Further, there are several problems when using GPUs for IP forwarding where one of the main disadvantages is that it introduce a higher latency compared to CPU implementations. One reason for this problem is the kernel launching overhead which we were able to almost remove in our prototype in this thesis work. Another problem the authors saw was the non-coalesced memory accesses which we also were able to improve in our prototype by doing coalesced and align accesses to packets. Finally an additional thing that would make it interesting to try their hypothesis again, after our thesis work, is that our prototype performs additional operations on the GPU like packet header changes, quality of service and traffic shaping which was not the case in this research. By this extra amount of work on the GPU the impact of the GPU overhead decrease.
Chapter 4  Method

4.1 Evaluation

To perform the evaluation tests on our application we used capture files that were created in a separate program in order to simulate different traffic scenarios. The settings of these traffic scenarios are described in this chapter.

4.1.1 Test traffic

Depending on where in the network a router is used it will see traffic of different kind and with different distributions. To be able to benchmark our solution we needed to have test traffic that made our figures comparable with benchmark tests on other routers. Because of this we started looking at different traffic models that we could use for our tests.

For the distribution of packet sizes we used the IMIX (Internet Mix) (Agilent technologies, 2007). Internet Mix is a common model used to describe the distribution of packet sizes on the internet. The simple version uses packet sizes of 40, 576 and 1500 bytes that are distributed 7:4:1 respectively. The correlation with real network traffic has been measured to 0.892 by Agilent. Since our IP forwarding on the GPU only uses the Ethernet and IP headers the size of the total packet has no effect on the measurements that we performed. For that reason we also ran some tests with packets with a minimal payload in order to be able to buffer more packets in RAM. These tests could measure the kernel time over a longer time and give a more precise measurement. Comparison with shorter runs with the IMIX traffic showed no difference in throughput.

Regarding different priorities between packets we let all packets be evenly distributed over the priority levels. All destination IP addresses are randomly generated.

4.1.2 Measurement

In order to be able to test the stability of our application we had to test it multiple times to make sure it never crashed due to errors. We tested this in several ways. The first test included a test script that automatically ran our application multiple times and reported the test results. Secondly, we shaped the traffic to test it for different bandwidths. Finally we performed a stress test where we tested the application for ten consecutive hours.

When performing benchmark tests we quickly discovered that the speed with which we could read packets from the file system was much lower than the speed with which we could process them. Because of this we did our measurements between the time the packets were put in the buffer to the time they were ready to be written back to file. This eliminate the file system delay.

If nothing else is mentioned the forwarding table consists of 4000 randomly generated entries including random masking. It requires about 40kB memory space and is stored in the shared memory.
The application was tested with one million packets. To ensure correctness of the results, each test was performed several times.

4.2 Implementation

In this chapter we will describe important steps of the implementation phase of our thesis work. We will explain which assumptions that the development relies upon and give intermediate results to argue for the design choices that we have made.

4.2.1 Forwarding table

We chose a bitwise Trie to represent the forwarding table. These structures are widely used for IP forwarding today and have support for implementing all needed features for the forwarding data plane. The Trie we implemented was path compressed and the largest constraint we had regarding this table was that it had to fit into either constant or shared memory which have a much lower access time compared to the global memory.

<table>
<thead>
<tr>
<th>Child index</th>
<th>Interface index</th>
<th>Skipped bits</th>
<th>Nr of skipped bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bits</td>
<td>8 bits</td>
<td>32 bits</td>
<td>8 bits</td>
</tr>
</tbody>
</table>

Figure 7, representation of a node in the Trie.

In Figure 7 you can see the representation of a node in the Trie. All four fields are byte aligned, which makes accessing them easier, and together each node is 64 bits long. Both access to the constant and the shared memory has a bandwidth of 64 bits per memory access (Nvidia, 2015h). This means that we can retrieve one node in one memory access for the shared memory. For the constant memory, which results are cached in the L2 cache with a cache line of 32 bytes, it means that four nodes are retrieved once a thread request a node. This is performed in 4 memory accesses.

We chose to only store one child index in each node according to the discussion in Chapter 2.1.3.2. According to this thesis specification the number of entries in the Trie would have to be up to 10000. A field of 16 bits could then handle 32 768 entries, since it requires 2n-1 nodes to represent n entries, which means that it is possible to handle more entries than required. The information about each node child index is stored in the field child index.

Besides this each node also has to contain an index to a gateway which is stored in the field interface index. We chose to represent this index with 8 bits, which gives support for up to 256 Ethernet ports. Given that each gateway port supports a standard Ethernet interface with a throughput of 10 Gb/s this will together support a throughput of 2560 Gb/s. If the average size of a packet is 340 bytes (Agilent technologies, 2007) that corresponds to 7.5 Gpackets/s which would not be a bottleneck for the overall system performance.

The next field is the skipped bits field that is needed when introducing path compression and the tree traversal would no longer be enough to retrieve the IP address. This field stores the actual skipped bits. It has a size of 32 bits to be able to contain whole IPv4 addresses which is the maximum number of bits that can be skipped.

Finally we need a field to represent the number of stored bits in the skipped bits field which is stored in the field Nr of skipped bits. Without it the application would not know how many bits in the skipped bits field it should compare with the destination IP address. The number of skipped bits could be between 1 and 32 which means that this bit field had to be at least 5 bits.
To create the Trie we created a text file with the same structure as the route file in an Ubuntu operation system. The application read from this file line by line until all entries are inserted. For each entry there are several steps that need to be performed in order to insert it into the Trie structure. First we have to find the node, furthest down in the tree, that matches the most bits of the IP address of the route that are going to be inserted. Then the skipped bits are compared to see where they differ and should be split into two parts, the old branch for the nodes that already exist in the tree and the new branch. When this is performed you set the number of skipped bits of the new node to the total depth of the tree minus the current depth. For the old node you change the number of skipped bits to match the newly changed tree structure.

In order to support masking we had to make the Trie implementation able to contain indexes to interfaces in all nodes, not only in the leafs. Even if an IP address does not match completely down to a leaf of the tree, it should be matched to the interface index for the last traversed node with an interface index.

![Diagram](image)

**Figure 8, introducing the need of dummy nodes when working with masking.**

The main difficulty with adapting to this functionality was that it introduced the need of dummy nodes. We could no longer merge a parent, with only one child, with its child because of the cases where intermediate interface indexes were needed to be stored. For example in Figure 8 the nodes to the right of the root node seems to be able to be merged together. However if the parent node contains an intermediate interface index for all IP addresses starting with one and its child are more precise, having another interface index, this is not possible. As mentioned earlier we only store an index to the leftmost child in each node and assume that the other child are on the next index. For that reason we need a dummy child to keep this structure even if the dummy node does not contain any useful data by itself.
In our Trie implementation we chose to exclude the level compression optimization. We did this because it would mean that updates to the Trie would be much more expensive, making it harder to do dynamic changes in the Trie. If level compression were used, a single new entry to the forwarding table could change the branching factor for several nodes, creating the need for big parts of the Trie to be rebuilt. It would be possible to maintain several copies of the forwarding table where one is currently used and one is currently updated. These two tables could then switch roles at some interval which would hide the updating computation of the application. However final tests on the Trie data structure show, as can be seen in the result chapter, that the forwarding table was already faster than the remaining parts of the application. Implementing level compression that supports masking would also introduce an extra level of complexity which would require more work.

4.2.2 IP forwarding packet processing on the GPU

Since the GPU offers more processing power compared to the CPU, for tasks that could be parallelized, it would be beneficial to have the IP forwarding processing on the GPU. In order achieve that we put packets into different chunks where each chunk was processed by the GPU when they were either full or there was a timeout. The reason why we implemented this timeout mechanism was because of the scenarios when a low rate of packets arrive, the other packets latency within a non-full chunk, should not be too largely impacted. Each one of the chunks could be handled independently from each other which give the application the possibility to process them in parallel on different workers which can be seen in Figure 9. Each chunk consist of at least the same amount of packets as there are threads on the GPU. By this approach all threads on the GPU can, in parallel, process one or several threads in each chunk.

The application consists of three separate workers. Using as many chunks as workers, give the application the possibility for each worker to simultaneously work on one chunk each. Therefore packets are stored in a circular buffer consisting of three separate chunks with packets. To remove the need of

Figure 9, sketch over the program structure.
having to copy chunks back and forth between the CPU and the GPU memory we used zero-copy memory that is shared between them. The first worker read packets, stored in a capture file (Pcap file), and puts them into the circular buffer. The second worker process the packets, within the chunks in the circular buffer, by launching a GPU-kernel and perform a number of operations. First of all it decrease each packets TTL. Secondly it perform a lookup operation in the forwarding table (FIB). Finally it change each packets MAC-addresses by matching the returned index from the lookup operation to an output interface which corresponds to a destination MAC-address. The source MAC-address is set to the routers MAC-address. The last worker then reads the processed packets from the circular buffer and writes the packets back to a capture file.

Each chunk has three flags. The first flag corresponds to if the chunk is ready to be processed meaning that it is either full with packets or there was a timeout. Further the second flag corresponds to whether the chunk has been processed or not. Finally the third flag corresponds to if the chunk is ready to be filled with new packets. When a worker is done with their work on one chunk they set a flag that shows that the chunk are ready for the next step in the processing pipeline. Since all packets were put into this circular buffer and the internal order were never changed there exist no packet reordering in this solution.

4.2.3 Align memory access

Aligned memory access to packets, that were going to be cached, was something we had not considered so far in our development. The width of a cache line in the L2 cache, which is used for caching accesses to the unified memory, is 32 bytes. This lead to the conclusion that if the memory accesses were going to be aligned, each packet had to be a multiple of 32 bytes. In order to achieve this we let each packets size grow to a multiple of 32 bytes which gave our application an increase in performance by 12%. Since we used fixed sizes of the buffers there are no extra overhead for letting each packet size be larger than before. Once the program is started, the CPU creates the circular buffers and allocate space for all chunks that they can contain before launching the GPU kernel. All packets are then assumed to be smaller or equal to the space allocated to them in the buffers. The maximum size of a packet, which is allocated in the memory, depends on which protocols the buffer should support.

4.2.4 Coalescing

Coalesced memory accesses are about accessing memory spaces close to each other in memory from different threads instead of doing accesses widely spread in the memory. This is another important optimization when working on GPGPU applications. For that reason we split the chunks of packets into several smaller sub-chunks. The first sub-chunk contains the first 22 bytes of the packets that are fixed fields for the Ethernet header where the most interesting fields for us are the source- and destination MAC-addresses (IEEE, 2012). The next 20 bytes of the packets are stored in the next sub-chunk which are mandatory fields for the IP header (RFC, 1981). The interesting fields that we used in this buffer were the checksum, TTL and destination IP address. All the remaining bytes were put in one last sub-chunk since our application were not performing any deep packet inspection or similar that would require these parts of the packets. After a packet has been processed the packet will be merged together again. By doing this the application could read and write data in both the IP header and the Ethernet header in a coalesced manner on the GPU. This implementation gave us a huge gain in performance and our GPU application throughput increased by about 700%.
4.2.5 Kernel launch evaluation

In this section of the work we wanted to investigate the overhead of launching kernels with different sizes of the chunks. Each chunk as mentioned earlier, consists of a number of packets. Smaller chunks mean that the kernel has to be launched more times compared to using larger chunks in order for the application to process the same amount of packets. In this test we used the test scenario described in Chapter 4.1. The number of threads per block was kept constant (512) while the number of blocks was adjusted to make the total number of threads equal to the chunk size.

Latency is important since a too long delay in the transport of a packet over a network could result in little or no use of the packet when it arrives to its destination. An example of that could be VOIP packets where a too high latency results in a delay in the conversation which would mean a lower quality of the service. Basically a large buffer means a higher throughput with a higher latency which was introduced as the throughput vs latency problem in Chapter 3.

![Kernel execution time on one chunk](image)

**Figure 10, kernel execution time depending on the size of a chunk.**

Figure 10 shows that the kernel launch time is almost the same for different chunk sizes. The reason for that is because the number and sizes of the arguments that the kernel is launched with are always constant. The only parameters that varies in these different scenarios are the number of packets in each chunk that are stored in zero-copy memory and therefore is not a part of the kernel launch. The kernel launch introduces an extra delay of about 100 µs for every packet which is not acceptable for routing purposes. The reason for that is because the end to end delay of a complete router (the time from when a packet arrives to a router to the time it exits) is about 50 µs. In other words the kernel launching and terminating overhead is twice as high as the latency a complete router introduces. This extra delay needs to be removed or hidden in some way and for this reason we started to investigate if there would be possible to launch the kernel less often and let it remain processing several chunks before shutting down.
4.2.6 Infinite kernel

An infinite kernel means that a kernel is running for an infinite amount of time. In our case we launched one kernel at the beginning of our application and then kept it running through the whole application life time. By doing this we got rid of the kernel launch time that was discussed in Chapter 4.2.5. Figure 11 shows the structure of this version of the application, which is similar to the version shown in Figure 9. Since the CUDA kernel is only launched once there was no need to maintain a CPU process thread, whose only responsibility was to launch kernels. This worker is completely moved to the GPU. All Nvidia GPUs with a compute capability version of at least 2 has support for unified virtual address space (Nvidia, 2015h). This means that both the CPU and the GPU are able to use the same pointers to access the memory. By using this the GPU is able to directly fetch chunks from the circular buffer and are only terminating if the complete application is stopped.

To be able to have an infinite kernel, all threads working on one chunk on the GPU would have to be able to communicate in order to choose a chunk to process and to shape the throughput. To achieve this effectively they need to do this through shared memory which is only shared by each block. Our development board has a maximum size of 1024 threads per block and for that reason the program contained only one single block with 1024 threads. One of the threads becomes the leader, in our case thread 0, and handled the chunk flag set and get operations. When a chunk becomes ready all threads cooperate to process the whole chunk and when all packets have been processed the leader thread signals the writer thread by setting the processed flag.

Since one block corresponds to one streaming multiprocessor we utilize the fact that our development board only has one them. If the GPU on the development board would consist of several streaming multiprocessors it would make all of them but one idle since we only use one block. That would lead to the fact that we would need several blocks to keep the other multiprocessors busy and avoid wasting hardware resources. Since threads within different blocks are not able to communicate effectively, the program would have to be separated into one worker for every block where each one of them operates on different chunks. Some synchronization such as fetching chunks in a synchronized manner would have
to go through the global memory, which would slow down the program execution. If there were more blocks than streaming multiprocessors it would mean that the shared memory would be divided between them and if the Trie is stored there each block would still need to keep a unique copy of it. This would mean that fewer entries would be able to fit in the forwarding table.

4.2.7 Quality of Service

In order to achieve some simple prioritizing of packets we decided to use three different circular buffers, instead of one as in earlier versions of the application, for packets of different priorities. The program structure was update as reflected by Figure 12. The number of buffers could easily be changed in future versions. These buffers are each assigned an unique id and each packet is put in one of these different buffers according to their priority decided by the DSCP field in the IP header. The GPU processing worker is now divided into several threadgroups that process different chunks in parallel. This means that the size of a chunk was decreased to match the fewer number of threads that operate on each chunk.

Each priority buffer is divided into several chunks where each chunk consists of a warp number (32) of packets. When a chunk is either full or there is a timeout, the chunk gets assigned to a threadgroup on the GPU. The threadgroup consists of a warp number of threads, one for each packet in the chunk. Each threadgroup operates independently from all other threadgroups and flag the chunk as completed once the forwarding operations are complete. Threads in the same threadgroup are synchronized implicitly since they all belong to the same warp. When there now exist several threadgroups, that work independently on the GPU, there have to be at least as many chunks in the buffers as there are threadgroups in order to support all threadgroups working simultaneously.

![Figure 12, program structure with QoS buffers.](image-url)
Since each buffer has a different priority there are different number of threadgroups assigned to each buffer where the highest priority buffer has the largest amount of assigned threadgroups. Each threadgroup first prioritize working on their own priority level but if there is currently no available chunks on that level they will start looking for available chunks on the other priority levels as well. The threadgroup will continue looking on different priority levels until it finds an available chunk to process. By implementing work stealing the application can guarantee a high throughput even if the packets are not even distributed between the different priority levels. In our implementation once a threadgroup thread changes priority it would process at most one chunk on another priority buffer until the threadgroup returns looking for a chunk on its original priority level again. This approach would introduce an extra delay for the case where a chunk becomes ready directly after a threadgroup has decided that there are no available chunks in the current buffer and changed its priority level. The extra introduced delay would be about 27μs in the worst case because of the time it takes to process a chunk, which can be seen in the result chapter. There is a tradeoff between idle waiting for a chunk to become ready in the current buffer and where the threadgroup immediately start looking for available chunks in other buffers. If the threadgroup starts looking for available chunks in other buffers immediately, that leads to a higher throughput and waiting for a chunk to became ready in the current chunk gives a lower latency.

Since different threadgroups now are looking for chunks to process at the same time they need to synchronize so that several threadgroups do not start to process the same chunk. To achieve this we use a lock free synchronization technique called compare-and-swap (CAS) (M.Stone, 1990). Since the compare-and-swap synchronization algorithm has the property of lock-freedom (Gao H, 2006), which means that at least one thread always makes progress, it does not suffer from any deadlocks.

The main idea with this way of synchronization is to ensure that shared data is not altered by more than one thread at a time. Several threads are allowed to simultaneously try to change the data but only one of them will succeed. This synchronization mechanism consist of a loop that starts with that a thread reads the current value of the variable it wants to change and typically store it in a variable called oldVal. The new value of the variable is calculated and is typically stored in a variable called newVal. The loop ends with a comparison; if the variable still has the value of oldVal it means that no one has changed it since the thread read it and it is safe to update it to newVal. This update is done atomically. Otherwise if the value is not the same the thread has to restart all operations in the loop and try everything once again.

Below you can see pseudo code of our implementation of this synchronization.

```c
__shared__ currentChunk;
Reset priority level to original level;
do {
    while(true){
        oldVal = currentChunk;
        newVal = oldVal + 1;
        if(oldVal is ready to be processed){
            break;
        } else {
            priorityLevel = (priorityLevel + 1) % nr prioritylevels;
        }
    } while (oldVal != atomicCAS(currentChunk, oldVal, newVal));

In this implementation currentChunk is the shared variable that threads are trying to gain access to. This variable corresponds to the next chunk that should be processed and is unique for each priority
```
buffer. The differences from the standard CAS synchronization algorithm is that our implementation also
contains an inner loop. In order to break out from this loop oldVal has to be ready to be processed,
otherwise there is no need to get exclusive access to it if it is not ready anyway. If the chunk was not
ready the chunk stealing mechanism get started and the thread change the priority level to where it looks
for chunks to process.

Since the CAS synchronization is between threadgroups that are currently working on the same
buffer no threadgroup working on other buffers can disturb. This means that at least one chunk in each
buffer is always processed, if there is available chunks in it, which is important in order to ensure that all
buffers have a limited maximum latency.

Since all threadgroups now suddenly are working independently it became harder to ensure that the
application do not perform any buffer reordering for packets that could take harm by this approach.
Packets on different priority levels have no such dependencies so packets in different priority buffers
could be reordered without any risk. For packets in the same buffer some mechanism for reordering
packets has to take place. Our solution became to use the write thread to implement a delay commit
queue. This queue only let packets, in the same buffer, out in the same order as they were put in the
buffer. In other words even if the chunks in the same buffer were processed out of order they were
released in the right order anyway.

One additional thing to consider is the number of chunks each priority buffer is going to consist of. A
large number of chunks in each buffer would result in a higher maximum latency while less chunks
would result in a higher packet loss (Zhu, 2011). Since different amount of threadgroups are working
concurrently on each priority buffer it makes sense that the higher priority buffers, with more assigned
threadgroups, would consist of more chunks. The number of chunks could then reflect the number of
threadgroups that are assigned to the buffer in order to make the latency of all buffers equal. The highest
priority buffer has half of all chunks, the middle priority buffer two thirds of the remaining chunks and
the remaining were assigned to the lowest priority buffer. These gave, as can be seen in Chapter 5.6, an
equal latency for packets in the different buffers but with a higher throughput for higher priority buffers.
The total number of chunks is a configurable setting and its performance is also shown in the mentioned
chapter.

### 4.2.8 Traffic shaping

If the traffic for some reason has to be shaped to a traffic model you need a traffic shaper. An
example of this could be that the bandwidth that one operator has bought from another has a limitation of
10 Gb/s. In order to implement this functionality in our prototype we use a slightly modified token
bucket where each clock tick correspond to a token. This means that the number of available tokens
increases with the rate of the clock frequency. In the following discussion we will use the notation clock
ticks instead of tokens in order to make the description easier to follow. Below you can see pseudo code
of the shaper.

```c
//Calculate number of required ticks to send chunk
ticksNeeded = clock frequency * size of chunk / desired throughput per second;

//Schedule tick to send packets, atomic operation to avoid race conditions
scheduledTick = atomicAdd(lastScheduledTransfer, ticksNeeded);

//Wait for scheduled tick to send
while(scheduledTick > clock{}){}
release chunk;
```
This code is executed by one thread in each threadgroup. The first thing that needs to be calculated is the time that needs to elapse, since the previous transfer of any threadgroup, before the current chunk can be released. This is calculated by dividing the size of the chunk by the desired throughput.

\[ \text{Time Needed} = \frac{\text{Size of chunk}}{\text{Desired throughput per second}} \]

To convert this to GPU clock ticks it is simply multiplied with the GPU frequency. This is then atomically added to a shared variable `lastScheduledTransfer` in order to get the scheduled time for the current chunk. By the usage of this shared variable the threadgroups keep a common schedule for transfers where each threadgroup can atomically reserve timeslots. After this the threadgroup has a scheduled clock tick when it is allowed to release the current chunk. The threadgroup performs busy wait until that clock tick arrives.

One thing that is missing in the above example is the calculation of the size of all packets within a chunk. This is used to calculate the number of ticks that are required to wait before being allowed to release a chunk. This calculation is the only part of the shaper where the GPU can take advantage of all its threads. Pseudo code of this can be seen below.

```c
//Retreive size of own packet and store in packetSize
//Calculate total size of the warp chunk
for (i=16; i>0; i = i/2) {
    packetSize += __shfl_down(packetSize, i);
}
```

In CUDA there is a mechanism called warp shuffle (Nvidia, 2015h). What it basically does is that without using the memory it enable the threads to share variables between their neighbors in the same warp. Each thread start by retrieving their packet size, that are stored in the packet header. The summation of packet sizes is then done by an optimized parallel reduction algorithm. Half of the threads in the warp are adding their packet size together with the size of the packet controlled by the thread with id equal to their id plus 16 (half warp). These operations are issued in a coalesced manner since all reading operations are from threads next to each other in memory. After this operation the first 16 threads in the warp have a size of two packets each. The hop length are then divided by two and becomes 8 and the same operations are repeated again. Now the first 8 threads are storing the size of four packets each. These operations continue until all packet sizes are stored in the first thread in the warp.

Once the shaper was integrated into our application the performance decreased by approximately 9%. The settings used in this test follow the specification in Chapter 4.1. The clock that are used to generate tokens has a maximum value of \(2^{64}\) which means that the time from starting our application to the time it overflows become:

\[2^{64}/(852 \times 10^6 \times 3600 \times 24 \times 365) \approx 686 \text{ years}\]

Since this application would never run for such a long time this will not be any problem. 852 \times 10^6 is the clock frequency of the GPU.
4.2.9 Memory storage of forwarding table

![Figure 13, performance comparison between storing the lookup table in different types of memory.](image)

The forwarding table lookup operations is an important part of the application and we would benefit from having as fast memory access to the table as possible. There are mainly three memory alternatives to where to store it namely in global, constant or shared memory. Regarding the shared memory the forwarding table has to be uploaded once every kernel launch which for us means only at the start of the application. This extra upload time is included in the measurements. The result of this test shows, as can be seen in Figure 13, that global memory is really slow compare to the constant and shared memory. Further the constant memory cache all fetched data in an 8kb on-ship cache which means that the cached data, from the constant memory, has as fast access speed as the shared memory. However if the forwarding table is large more accesses will go through the constant memory, residing in off-ship memory, instead of the constant cache. This result in a performance degradation which can be seen in the figure.

The above reasoning lead to the conclusion that the shared memory is the best solution for this application. It offers an equal or better performance compared to other alternatives. If we were to upload the forwarding table several times to the shared memory, in the case of several kernel launches, we would see a similar trend as in Figure 10 where more kernel launches introduce an extra overhead. Then the constant memory could be a better alternative.

Each one of these three memory types have different sizes where the global memory is by far the largest one and could compared to the others hold an infinite number of entries. The constant memory is able to store about 4000 entries and the shared memory about 3000 entries. In order to support 10000 entries, which was the upper bound of entries in our thesis specification, it would not be enough use both the shared and the constant memory together. Instead the global memory would have to be used and the constant or shared memory could act as a cache for retrieved Trie nodes. This would introduce a decrease in performance but given a good architecture where most accesses to the forwarding table was through the constant cache or the shared memory the performance could still be acceptable. Due to higher priorities of other tasks in this thesis this architecture were never implemented.
4.2.10 Optimal number of threadgroups

The GPU processing is as mentioned earlier split into several threadgroups where each one consists of a warp number of threads. Since more threadgroups offer a higher potential degree of parallelism the program should benefit from using more threadgroups up to a certain degree. At some level the GPU would not be able to schedule more threadgroups to be processed in parallel because of the limited hardware resources and software constraints. We would like to find the optimal number of threadgroups given our application. The settings in this test are described in Chapter 4.1. As mentioned earlier the kernel consist of only one block.

Figure 14 show that the highest throughput is achieved by using 512 threads which means 16 threadgroups of 32 threads each. There are several possible explanations to the trend that we can see in the figure. As mentioned earlier more threads gives the warp schedulers more warps to schedule in parallel which gives a potentially better GPU utilization. The reason why the throughput does not increase even further depends on a couple of reasons. First of all the CAS operation is only able to grant access to three threads simultaneously, one for each priority buffer. Secondly our development board has several hardware constraints when working on special functionalities such as warp shuffling, bit shifting and type conversions to and from 64 bit integers (Nvidia, 2015h). The development board has support for 32 shifting and shuffling operations to be done in parallel and for type conversion only 8 threads can operate in parallel. This means that only one warp is simultaneously able to summarize its packet sizes, using warp shuffle, which means that other threadgroups will be queued to get access to this hardware. When comparing IP destinations in the Trie we have to shift out irrelevant bits from the IP destination address in order to match it against the skipped bits in the Trie. Here it is also only possible for one warp to operate simultaneously. The type conversion to and from 64 bit integers is used in the shaping functionality to retrieve tokens from the token bucket and is performed once by each warp for every chunk they process. When you add all these constraints together you can draw the conclusion that a higher number of warps do not increase the level of parallelism and therefore neither the total throughput.
4.2.11 Final prototype

This final prototype includes everything that has been discussed in this chapter. Additionally it also includes correcting the checksum field in all packets. That is performed by incrementing the checksum field for every packet to respond to the decreased TTL field. Since the TTL change is the only change in the IP header it is an effective solution for maintaining a correct checksum field. The changes of MAC-addresses corresponds to the Ethernet header which is at a lower level of the packet and therefore does not affect the IP header checksum calculation. In the following chapters it is this prototype that is going to be evaluated and discussed.
Chapter 5    Results

All measurements in this chapter follow the test setup described in Chapter 4.1. Some changes are made but these are then clearly stated for each test result. For all tests the GPU kernel consist of one block with 512 threads that are decisions motivated in Chapter 4.2.

5.1 Throughput

![Throughput Graph]

**Figure 16, throughput given different types of traffic.**

Figure 16 shows how the throughput depends on different types of traffic. There are two different scenarios. The first is when all packets are located in the same traffic priority buffer, due to their DSCP field in the packet header. There are three different priority buffers but the throughput they offer are the same so this traffic scenario reflects all of them. The second traffic scenario is when all packets are evenly distributed over all priority buffers.

5.2 Shaper performance

The shaper that we have developed decrease the total program throughput by 9 % and has a fault marginal of less than 0,001 %. We have verified that it is able to control throughput speeds from 10 Kbit/s up to 1000 Mbit/s. These tests were performed by testing the application several times with both disabled and enabled shaper.
5.3 Trie lookup

Figure 17 shows the throughput of the application when only performing forwarding table lookup operations. In these measurements we used different amount of entries in the Trie. The figure is showing that the best performance is achieved on small amount of entries and that the throughput is decreasing linearly by the entries. In the test environment we disabled all operations on the GPU except the lookup operations and let everything else be the same as described Chapter 4.1.

5.4 GPU utilization

Figure 18, time spent on different kind of operations by the kernel.
Figure 18 shows the utilization of different units on the GPU and are collected from the analysis of the application in NVVP. We can see that the memory accesses are dominating the execution time of the application whereas the actual computation where active for less than 20% of the time.

### 5.5 GPU execution count

![Figure 19](image1.png)

Figure 19, shows the usage of functional units by the kernel.

There are several possible types of instructions the GPU are able to handle. This table, collected from the NVVP, shows the distribution in percent between them and give a picture over what the GPU kernel in our application is doing on an instruction level. The inactive column represent idle threads.

### 5.6 Latency

![Figure 20](image2.png)

Figure 20, shows how the latency depends on the size of the buffers.

This figure shows how different amount of buffered packets affect the total latency of an incoming packet. This test where performed by buffering up all packets before starting to process them. The processing time was then measured with timers from the CPU side.
An incoming packet would have to wait until all previous packets in the same priority buffer have been processed. This leads to an increased latency for buffers containing more packets. The x-axis shows the total number of packets which are distributed over all priority buffers. 1/6 of all packets are in buffer 0, 2/6 of all packets are in buffer 1 and 3/6 of all packets are in buffer 2. Since the different priority buffers have corresponding number of threadgroups assigned to them all buffers clear their chunks in an equal speed. Therefore a packet has an equal latency independent of which buffer it is assigned to but the throughput of different buffers varies. Since our development board has 192 cores which means hardware support for 192 threads executing in parallel this figure does not contain measurements with less than that amount of packets.

5.7 Power consumption

<table>
<thead>
<tr>
<th>Test case</th>
<th>Power consumption [W]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>3,12</td>
</tr>
<tr>
<td>CPU only</td>
<td>4,92</td>
</tr>
<tr>
<td>CPU and GPU</td>
<td>5,52</td>
</tr>
<tr>
<td>Keyboard, mouse and HDMI</td>
<td>0,72</td>
</tr>
<tr>
<td>Maximal power consumption GPU</td>
<td>4,56</td>
</tr>
<tr>
<td>Maximal power consumption CPU</td>
<td>11,4</td>
</tr>
</tbody>
</table>

Table 2, shows power consumption of development board in different use cases.

These tests were investigating the power consumption of the application and was measured using a Fluke 87V industrial multimeter (FLUKE, 2015). All these test cases includes the power consumption by the keyboard, mouse and HDMI which was also measured and stated in the table. Further the test case idle is showing the power consumption when the development board is booted but is not doing anything. CPU only shows the power consumption when only the CPU part of the program is running and the file I/O is disabled. CPU and GPU shows the same but were the GPUs power consumption also is included in the measurements. Finally the maximal power consumption for both CPU and GPU is shown in Maximal power consumption. The test were performed by a system benchmark CPU test for the CPU and the CUDA matrix multiplication example, from the CUDA samples, for the GPU.
Chapter 6  Discussion

In this chapter we will discuss the results of our thesis work and describe what our results mean in a wider perspective.

6.1 Throughput

The highest throughput is achieved when having an equal distribution between the different priorities of packets which is natural due to the fact that the threadgroups do not have to steal chunks, which takes further time, from other priority buffers as much as in the other traffic scenarios. Further we can see that our chunk stealing algorithm works well and the application can offer a high throughput even if the packets are not well distributed over the different priority levels.

Since our prototype does not copy packets back and forth to the GPU, but instead only reads them from main memory, our processing time is not affected by the size of the packets. Because of this we have chosen to measure our throughput in packets per second instead of bytes per second. To compare our result with throughput specified in bytes per second we can simply multiply by the average packet size for the test data.

6.2 Shaper

Traffic shaping is by nature a sequential operation that is hard to parallelize. Each threadgroup, within this application, are able to operate completely in parallel but not the threads within each threadgroup. The only thing that can run in parallel within a threadgroup is the summarization of the packet sizes in each chunk. Since many parts of the application are of this nature, where threadgroups are able to work independently from each other but not the threads, this shaper does not decrease the overall parallelism. To summarize, the shaper achieved a good precision but decreased the processing speed by 9%.

The shaper that we has developed for our prototype builds on the assumption that the GPU only has one streaming multiprocessor which is true in our case. If we would have used another GPU with several streaming multiprocessors our current implementation would not have been equally effective since different streaming multiprocessor could not share the same shared memory. Without this the tokens, used in the shaper, could not be stored in the fast shared memory and easily be shared by all threads. Also the same problem arises if using more than one block.

One possible solution to this problem would be by using one shaper for each multiprocessor where each multiprocessor gets a fair share of the allowed throughput. But if the load balance gets uneven between the streaming multiprocessors the throughput would be lower than desired. There would have to be some load balancing functionality between
different multiprocessors to get this solution to work effectively. That on the other hand would introduce an extra overhead which would decrease the overall performance.

Another more promising solution would be to have available tokens in the global memory from which each multiprocessor can fetch to distribute on its threadgroups. This would introduce more accesses to global memory but would give a good precision to the overall throughput precision. This means that it would decrease the performance by more than 9% because of the extra access to global memory. But even if more accesses to global memory means a lower throughput it may still be worth it. If more streaming multiprocessors were used they would by that increase the overall throughput so in the end the total throughput could still be increased by this approach.

6.3 Trie

Figure 17 shows that the Trie could offer up to 10 times higher throughput compared to the whole application. No matter how much we would improve this Trie structure it would not lead to any significant improvement of the total application performance. However, level compression could decrease the average search depth of the tree which would improve its performance. If the memory bottleneck would be removed, which would speed up the rest of the application, this would have been an interesting thing to investigate.

Another interesting thing to consider is that the GPU is more effective on handling sequential memory accesses than by performing random memory accesses. Again, if the rest of the application would have been more effective, this knowledge could be used to speed up the Trie. By sorting the Trie structure and by that achieve more sequential memory accesses, to the nodes within the forwarding table, the lookup time would decrease.

6.4 Hardware

This thesis show that the big bottleneck of GPU implementations of IP forwarding is the memory. The memory controller is, as shown in Figure 18, active for almost 60% of the application execution time. This could be compared to the actual computation that is not even active for 20% of the total application execution. There could be several explanations for this fact but where some factors seem more likely to have a large impact. First of all many computations relies on memory accesses and if the memory is not able to fetch data from memory in equally high speed as they are processed the requests would have to be queued. Another explanation for the low computation utilization is that there are many sequential operations within a threadgroup where all threads but one are idling. Recall Amdahl’s law (Mark D. Hill, 2008) that describe that only a part of an application could be improved by parallelization. Some parts would still have to be processed in a serial manner. The conclusion of this still leads to that if the memory accesses were able to be more effective, by for example using of faster memory, the application overall performance could increase up to a couple of times. But the memory bandwidth is not the only problem. Another problem is the memory latency that is the reason to why the memory could not be active for the complete program execution. The memory accesses to off-ship memories have a latency of 200-400 clock cycles as mentioned earlier in Chapter 2.2.2. This latency has to be hidden by computations of other threads which we were not able to do fully in this prototype due to the need of many sequential operations and system bottlenecks described in Chapter 4.2.10.

Also the size of the different memories the GPU offers are limiting the maximum number of entries in the forwarding table. If the forwarding table grows too large, in our case over
4000 entries, parts of it would have to be moved to other memories which would increase the memory bottleneck even further.

Figure 19 shows the usage of different functional units in the application. In our program we used the first thread in each threadgroup as the leader for all threads within the same threadgroup. This introduces idle waiting for all threads except the first one within a threadgroup which are reflected in the inactive column. Recall to the occupancy concept that compare the number of active threads to the theoretical maximum number of active threads. By only letting one thread per warp be active on parts of the application the occupancy decreases which is a problem for an effective CUDA program. In earlier versions of the program, when using larger threadgroups, the problem was even worse due to more inactive threads. Since threads has to work in groups of at least a warp number of threads this problem is impossible to completely get rid of. There are always going to exist sequential parts in the program like choosing chunks to process, shape traffic and set chunk flags. By using threadgroups of exactly a warp number of threads the problem is minimized as much as possible and the occupancy is kept as high as possible.

Profiling of the final application show that the register usage support up to two blocks per streaming multiprocessor which set a limit on the number of possible concurrent blocks the GPU can support. Moreover as discussed in Chapter 4.2.6 more blocks would lead to more accesses to global memory and complications of where to store the forwarding table. In the current implementation we are using one block but further experiments on using several blocks would bring clarifications if that could improve the overall performance. The total processing time per packet would not decrease since more blocks only mean that the idle states of threads can be hidden. The thing that could be improved is the overall throughput. Figure 18 shows that there are latency problems in the program meaning that the threads are idle for long periods of time due to memory latency. This makes it promising to use more blocks that could hide these idle periods. Finally most new Nvidia GPUs have support for at least two times more registers per streaming multiprocessor which means that more blocks are able to fit. Using one of them would be interesting in order to even further be able to hide memory latency.

6.5 Latency

The number of packets in a router's buffer has a large impact on its latency. In Figure 20 you can see how the latency of our application depends on how many packets there are in the buffer. A large maximum size of a buffer would introduce a high latency with less packet loss compared to a small buffer but who has a smaller maximum latency. By limiting the total size of the buffer you can set the worst case latency. Which configuration that is the best and what is most important regarding latency or packet loss depends on the network. In general large packet buffers are unnecessary for traffic that cannot accept large delays. This figure also shows that the best achievable latency is about 27µS in the case where the GPU has a low load of packets in its buffers.

Since all 16 threadgroups have to operate on different chunks there needs to exist at least 32 chunks in order to be able to buffer up one chunk while another are processed on each threadgroup. This leads to the need of a buffer size equal to at least 1024 packets. Using a smaller buffer size are not recommended since it leads to a decreased hardware utilization when all threadgroups are not able to operate simultaneously.
6.6 Quality of service

In our development when implementing quality of service we faced a number of challenges. First of all our solution needed to consist of a high number of branches which the GPU has problem to handle effectively. Branches also introduce an extra overhead that decreased the programs total performance. By allocating different amount of threadgroups to different queues we gave higher priority buffers, with more assigned threadgroups, a higher throughput. In the meantime while there are no traffic available the chunk stealing mechanism could easily give the threadgroups assignments on different priority buffers. This algorithm gives by its simple nature an effective way to assign different threadgroups to different chunks. In the case when there exist chunks, ready to be processed, in the assigned buffer the decision of choosing priority buffer is removed completely.

6.7 CUDA

Even if CUDA has made GPU programming easier and more accessible it is still a quite new technique. This can for example be noticed by a weird behavior of the compiler which is unable to inline code that resides in other files than the one with the main kernel. This can lead to great performance losses if the programmer is unaware of it and still leads to more confusing and unreadable code for the programmers who do. Throughout our development we have noticed several such quirks of CUDA that makes the development process harder and less like the development of ordinary CPU programs.

6.8 Power consumption

The development board used in this thesis work has a power consumption of 3 watts in idle state and 11 watts when it is running on full effect at DC input according to NVidias measurements (Nvidia, 2014a). Besides this we also performed our own measurements as can be seen in Table 2. The numbers seems reasonable compared to the numbers NVidia published. One interesting part here is that our application has a power consumption of 5,52 W which, since the application supports a throughput of 28 MPackets/s, gives a throughput of 5,1 MPackets/W. Another interesting fact is that the GPU part of the application only increase the power consumption by 0,6 W. This could be compared to the maximum power draft of the GPU that is 4,56 W according to our measurements. One explanation to why we could not fully utilize the GPU is because the application is memory latency bounded and spends a lot of time waiting for the memory. Another explanation is the high degree of idle threads. A better utilized GPU would offer higher throughput but also consume more power.

6.9 Comparisons against existing routers

In the test phase of the thesis we also ported our GPU solution to the CPU which decreased the performance by a factor of 200. Worth mentioning is that this was not a fair measurement due to the fact that the GPU solution was well optimized and the CPU version was not. The only thing this tells us is an intuition of how much parallel execution we were able to introduce to the application.

Since the application only covers parts of a whole forwarding data plane functionality in a complete router it is hard to compare our solution to existing routers on the market today. One important thing for us to consider is to make sure that the throughput our application offers do not become a bottleneck for the complete system. A complete router, using the application we
have developed, could never has a higher throughput than the one our part of the router offers today. Another thing to consider is that the latency our application introduces seems reasonable when comparing it to the latency of a complete router. The latency in our application has to be considerably lower than the latency a complete router introduces since a complete router introduces extra latency when fetching packets from the network.

The company UBNT states that they has developed the first router under 100 $ that could offer a throughput of 1 million packets per second (UBNT, 2015). That is half the price of our development board but then you also should consider that our application can offer over 28 times higher throughput. The latency UBNTs mentioned router introduces is between 28.8 µs and 88.0 µs, depending on the size of the packets, when disabling the firewall (Enterprises, 2012). This could be compared by our best case latency of 27µs. Worth to consider is also that our latency does not depend on the size of the packets that are processed. This delay are going to be introduced when fetching packets to and from the NIC buffers. Regarding the power consumption this router consumes 7 watt in the worst case scenarios (EdgeMax, 2014). Compared to our router it seems like we are more efficient regarding throughput per watt.

Ericsson has developed a router called TCU03 which is a more expensive router that offers more computation power. This router are able to process up to 1 billion packets per seconds with an average latency of 48 µs up to 80 µs depending on the packet sizes. The latency could be increased further for packets of sizes up to 9 kilobyte. Our application could not compete with the throughput but the latency is at least in the same order of magnitude.

6.10 Method

The method that we used in this thesis were to iteratively add functionality to a naive implementation. We have considered all things that seemed to be able to have an impact to the total overall performance in order to find the best settings for the application. This way of working gave a clear picture of the thesis work and a natural way of finding a good solution for problems.

Things that would have made the work easier would be if we earlier in the project were given clear goals of what we were aiming for in terms of throughput, latency and functionality. Then we could have optimized for these problems instead of finding a broad solution that covered more of the IP forwarding data plane. Since IP forwarding is a broad area it is hard to cover everything but it has been interesting to get a wider insight in this area and way of working without a clear specification.
Chapter 7  Conclusions

Introducing quality of service in the application showed that it was quite easy to get a prioritization between different packages. The throughput remain in a similar scale independently from the priority distribution of incoming packets.

Without doubt the largest optimization of a GPGPU program of this kind was coalescing. It has a large impact on the total performance and therefore is important to consider. By splitting packets into several sub-chunks all corresponding parts, of different packets, gets close to each other in memory which gives a coalesced memory access. The downside of this approach is that some part of the program would have to split these packets which would require extra computations. We let the CPU handle this and when we only consider the GPU processing in our measurements this come outside these measurements. Further work will have to show if the performance gain of coalescing makes up for the extra computation time of splitting packets.

Overall after implementing IP forwarding functionality on a GPU the result show that we could achieve a good performance, in terms of throughput, latency and energy consumption, compared to existing routers on the market today. For further research it would be good to investigate how well GPUDirect is working when fetching packets from the network buffer directly as discussed in Chapter 2.4. If GPUDirect works well it would remove or at least decrease the CPU involvement. Our infinite kernel could, after some small adjustments, fetch packets directly from the network buffer. The CPU overhead would be decreased and the IP forwarding could be performed effectively on real packets instead of simulated ones.

Using CUDA to launch kernels that runs indefinitely is not part of the ordinary CUDA workflow and is one of the more interesting concepts we used. In our tests we have shown that it could give an increase in performance when working with IP forwarding applications. By removing the need of launching kernels several times we were not only able to decrease the total latency but we were also able to better utilize the GPU. We have tested the infinite kernel successfully on executions that were running for 10 consecutive hours by both letting it work during a stress test and a test where we shaped the traffic to a limited throughput. Because of the introduction of GPUDirect we think that using infinite kernels will be more and more common in the future since it reduces the need of involving the CPU. One drawback of using infinite kernels that we discovered were that the Nvidia profiling tools became unstable and causes crashes in the program execution. This a problem has to be investigated further in order to find the reason for.

Another thing that has to be investigated in future work is if using more blocks per streaming multiprocessor would increase the overall performance. It would also be interesting to test the application on different hardware platforms in order to compare their performance.
References

   Available at: https://developer.nvidia.com/gpudirect
   [Accessed 05 05 2015].
   [Accessed 20 05 2015].
   [Accessed 26 05 2015].
   Available at: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#axzz3VDDmlNgZ
   Available at: https://developer.nvidia.com/cuda-gdb
   Available at: https://developer.nvidia.com/cuda-memcheck
   Available at: http://docs.nvidia.com/cuda/gpudirect-rdma/#axzz3ZB0F4QGY
   [Accessed 05 05 2015].
   Available at: http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html
29. Nvidia, 2015g. *Nvidia visual profiler* [Online]
   Available at: https://developer.nvidia.com/nvidia-visual-profile
   Available at: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#performance-guidelines
38. Tsung-Hsien Li, H.-M. C. P.-C. W., 2013. IP address lookup using GPU. Taipei, IEEE.