Royal Institute of Technology (KTH)
Aalto University School of Science and Technology (TKK)
Degree programme of Nordic Security and Mobile Computing

Rerngvit Yanggratoke

GPU Network Processing

Master's Thesis
Stockholm, June 15, 2010

Supervisors: Professor Peter Sjödin, Royal Institute of Technology (KTH)
Professor Tuomas Aura, Aalto University School of Science and Technology (TKK)
Instructor: Hareesh Puthalath M.Sc. (Tech.), Ericsson Research, Stockholm

Aalto University
School of Science and Technology
Faculty of Information and Natural Sciences
Degree Programme of Nordic Security
and Mobile Computing

ABSTRACT OF THE
MASTER'S THESIS

Author: Rerngvit Yanggratoke

Title: GPU Network Processing

No. pages: 10 + 70 **Date:** 15 June, 2010 **Language:** English

Professorship: Data Communications Software **Code:** T-110

Supervisors: Professor Peter Sjödin
Professor Tuomas Aura

Instructor: Hareesh Puthalath M.Sc. (Tech.)

Networking technology is connecting more and more people around the world. It has become an essential part of our daily life. For this connectivity to be seamless, networks need to be fast. Nonetheless, rapid growth in network traffic and variety of communication protocols overwhelms the Central Processing Units (CPUs) processing packets in the networks. Existing solutions to this problem such as ASIC, FPGA, NPU, and TOE are not cost effective and easy to manage because they require special hardware and custom configurations.

This thesis approaches the problem differently by offloading the network processing to off-the-shelf Graphic Processing Units (GPUs). The thesis's primary goal is to find out how the GPUs should be used for the offloading. The thesis follows the case study approach and the selected case studies are layer 2 Bloom filter forwarding and flow lookup in Openflow switch. Implementation alternatives and evaluation methodology are proposed for both of the case studies. Then, the prototype implementation for comparing between traditional CPU-only and GPU-offloading approach is developed and evaluated.

The primary findings from this work are criteria of network processing functions suitable for GPU offloading and tradeoffs involved. The criteria are no inter-packet dependency, similar processing flows for all packets, and within-packet parallel processing opportunity. This offloading trades higher latency and memory consumption for higher throughput.

Keywords: GPU, Network processing, Openflow switch, Bloom filter

Utfört av: Rerngvit Yanggratoke
Arbetets namn:
GPU Network Processing

Nätverksteknik ansluter fler och fler människor runt om i världen. Det har blivit en viktig del av vårt dagliga liv. För att denna anslutning skall vara sömlös, måste nätet vara snabbt. Den snabba tillväxten i nätverkstrafiken och olika kommunikationsprotokoll sätter stora krav på processorer som hanterar all trafik. Befintliga lösningar på detta problem, t.ex. ASIC, FPGA, NPU, och TOE är varken kostnadseffektivt eller lätta att hantera, eftersom de kräver speciell hårdvara och anpassade konfigurationer.

Denna avhandling angriper problemet på ett annat sätt genom att avlasta nätverks processningen till grafikprocessorer som sitter i vanliga pc-grafikkort. Avhandlingen främsta mål är att ta reda på hur GPU bör användas för detta. Avhandlingen följer fallstudie modell och de valda fallen är lager 2 Bloom filter forwardering och "flow lookup" i Openflow switch. Implementerings alternativ och utvärderingsmetodik föreslås för både fallstudierna. Sedan utvecklas och utvärderas en prototyp för att jämföra mellan traditionell CPU- och GPU-offload.

Det primära resultatet från detta arbete utgör kriterier för nätvärksprocessfunktioner lämpade för GPU offload och vilka kompromisser som måste göras. Kriterier är inget inter-paket beroende, liknande processflöde för alla paket. och möjlighet att köra fler processer på ett paket paralellt. GPU offloading ger ökad fördröjning och minneskonsumption till förmån för högre troughput.

Språk: Engelska

Acknowledgments

I would like to thank wholeheartedly to my supervisors and instructor: Professor Peter Sjödin of Royal Institute of Technology for giving me an opportunity to work in this challenging research area and providing useful feedbacks, Professor Tuomas Aura of Aalto University School of Science and Technology for his excellent supervision and guidance, Instructor Hareesh Puthalath from Ericsson Research for continuous supports and inspiring discussions.

I dedicate my thesis to my family for encouragement and moral support in this long working period. I would like to thank as well to my friends here in Stockholm for inviting me to many parties and funny events. Because of them, I am active and have worked along this period with full energy.

Lastly, I want to acknowledge Ericsson Research for providing equipments and resources for this thesis. I would like to also express my gratitude to my colleagues in the company including Bob Melander, Jan-Erik Mångs, Karl-Åke Persson, and Annikki Welin. Without their supports, this work would not have been completed.

Stockholm 15 June, 2010
Rerngvit Yanggratoke

Abbreviations and Acronyms

VoIP	Voice over IP
ID	Identifier
PC	Personal Computer
TOE	TCP Offloading Engine
NPU	Network Processor
API	Application Programming Interface
GPU	Graphical Processing Unit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
IP	Internet Protocol
CIDR	Classless Inter-domain Routing
FPGA	Field-Programmable Gate Array
ASIC	Application-specific integrated circuit
DRAM	Dynamic Random Access Memory
SIMD	Single Instruction Multiple Data
Gb/s	Gigabits per seconds
MB	Megabyte
KB	Kilobyte
Mb/s	Megabits per seconds
RAM	Random Access Memory
SIMT	Single Instruction Multiple Thread
NIC	Network Interface Card
VLAN	Virtual Local Area Network
ICMP	Internet Control Message Protocol
HTTP	Hypertext Transfer Protocol
FTP	File Transfer Protocol
SMTP	Simple Mail Transfer Protocol
Latency	The time it takes to process an individual packet (seconds/packet)
Throughput	The number of packets processed over a period of time (packets/second)

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem statement

Recent development of networking trends and technologies are putting high pressure on the packet processing operations in core networks. The trends and technologies include moving of fixed line to VoIP [36], security threats [45], and incoming of 4G network [8]. The Central Processing Units (CPUs) in the networks become bottlenecks because there are much more data to process and the processing operations are increasingly complicated.

This problem may be solved by using specialized network processing equipments such as ASIC, FPGA, Network Processing Unit (NPU), or the TCP Offloading Engine (TOE). Nonetheless, the solutions may not be cost effective and easy to manage because they require special hardware and configurations.

This thesis approaches the problem differently by offloading the network packet processing to off-the-shelf Graphic Processing Units (GPUs). GPUs have considerable computation power because they are designed for computation intensive applications such as premium quality video or interactive games [29]. In addition, their computing powers and memory bandwidths are increasingly higher than CPUs as shown in figure 1.1 and figure 1.2.

This approach is feasible because of the recent development of general purpose APIs for GPUs. In the past, APIs for GPUs were limited to computer graphic environment such as OpenGL [20] or DirectX [22]. As a consequence, a programmer targeting for general purpose applications must map his/her problem domain into graphical context in order to utilize the unit [12]. For example, the programmer had to transform his/her data struc-

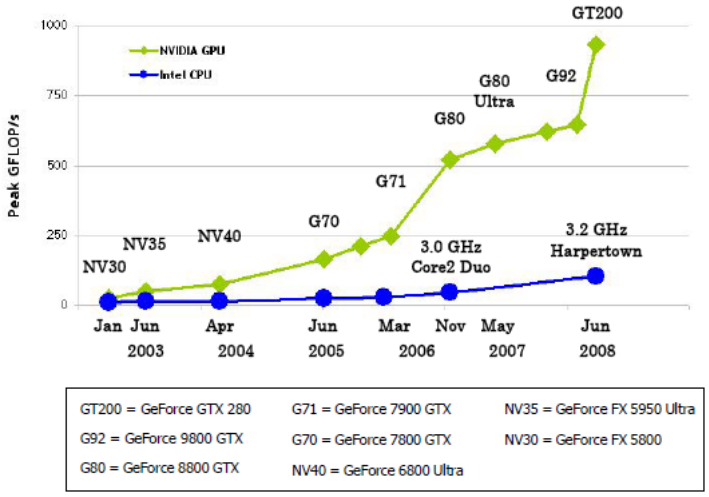


Figure 1.1: CPU and GPU computation power comparison [30]

tures into graphical pixels, ask the APIs to process such pixels, and retrieve the results [9]. This made GPU programming cumbersome and error prone. Furthermore, because the APIs were not designed for such applications, the result software was not very efficient.

This was changed because of the new APIs available for general purpose applications such as CUDA [29] and OpenCL [19]. Hence, programming the applications with the new APIs is more convenient, less complex, and can result in better performance than the past.

In addition, GPUs usually already exist in general PC configurations [12, 21]. As a result, the cost of ownership of this approach would be lower compared to the dedicated hardware solutions. Furthermore, because there is no specialized hardware configurations required, the system would be simpler to manage.

The main objective of this thesis is to study how GPUs should be used for offloading network processing functions. This means the thesis should identify the tradeoffs involved in the offloading. In the course of doing so, the project aims to have the following contributions:

- Properties of general purpose GPUs for network processing including suitable algorithms and its datapath organization / capacities.
- Software development issues for applications on a heterogeneous environment with CPUs and GPUs.

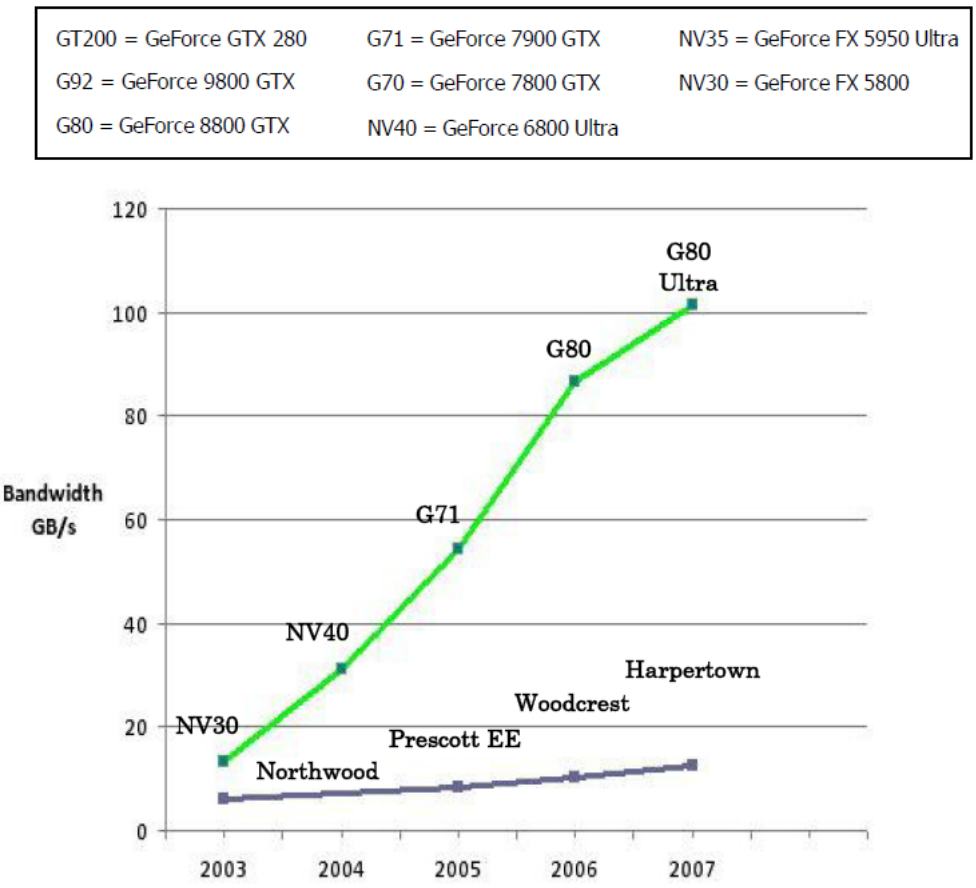


Figure 1.2: CPU and GPU memory bandwidth comparison [30]

- Selection of few representative scenarios for case studies. The case studies are the network processing functions suitable for offloading to GPUs.
- Analyzing the selected case studies implementation alternatives in term of performance, software development efforts, and advantages/disadvantages.
- Prototype implementation and evaluation of the selected case study.

1.2 Research approach

This thesis follows the case study approach. The approach consists of four stages: literature review, case studies selection / alternatives analysis, implementation / data collection, and the result evaluation.

The first stage of the thesis work mainly focuses on the literature study. The primary target is to find out the properties of the general purpose GPUs and software development issues on the heterogeneous environment with CPUs and GPUs. After the literature review is done, the second stage can be started.

Based on the first stage, few network processing case studies suitable for offloading to GPUs are selected. Beside the selection, the second stage includes the case studies' implementation alternatives analysis and evaluation methodology. The analysis concentrates on the performance, software development efforts, and advantages / disadvantages. In the methodology, the specific measurements for evaluating the system properties are specified. The methodology focuses on the comparison between the conventional CPU-only and GPU-offloading implementation.

For the third stage, one of the alternatives is selected for prototype implementation. The implementation is run and the performance data according to the evaluation methodology is collected.

After the data is collected in the third stage, the last stage or result analysis can be done. The behaviour of the system is discussed based on the empirical test result.

1.3 Thesis outline

This thesis is systematically organized to provide the reader with required knowledge before proceeding into deeper detail of the topics. As a result, the reader is suggested to read step by step started from chapter 2. Chapters 2 to 4 come from the literature study. The author's contribution starts from chapter 5. The thesis outline is as followed:

- Chapter 2 provides background knowledge required for the reader to understand the thesis. The knowledge includes GPU, network processing functions, and related works.
- Chapter 3 elaborates the properties of GPGPU including its historical

development, suitable applications, and network processing data path.

- Chapter 4 discusses the software development issues on the heterogeneous environment between CPUs and GPUs. The software development issues include compilation / execution, architecture, programming model, performance aspects, heterogeneous computing between CPUs and GPUs, and multiple GPUs.

- Chapter 5 explains the usage of GPU for network processing including the criteria for suitable network processing functions, general tradeoffs involved, performance limitation, and the case studies selection.

- Chapter 6 and 7 analyze different alternatives possible for the case studies of using GPU for layer 2 Bloom filter forwarding and flow lookup in Openflow Switch respectively. The analysis covers performance, software development efforts, and advantages / disadvantages. It also states the evaluation methodology and selected implementation alternative.

- Chapter 8 contains the analysis of the result from the implementation in chapter 6.

- Chapter 9 concludes the thesis and suggests future research areas.

Chapter 2

Background

This chapter provides background knowledge for the reader to understand the thesis. The reader is assumed to know basics of computer architecture and IP network before proceeding. Section 2.1 briefly introduces the Graphic Processing Units (GPU). Section 2.2 discusses overview of the network processing functions selected as the case studies. Section 2.3 shows existing works on offloading network processing functions to GPU.

2.1 GPU

This section discusses overview of GPU. Section 2.1.1 explains graphical processing in GPU, its primary purpose. Section 2.1.2 elaborates the GPU processing resources.

2.1.1 GPU for graphical processing

GPU was originally designed for graphical processing. The processing consists of five graphical pipelines including vertex programs, geometry programs, rasterization, pixel programs, and hidden surface removal [21]. All pipelines share their works via the GPU memory (DRAM).

As shown in figure 2.1, a graphic programmer starts the process by defining the 3D graphical primitives. The primitives are sets of triangles defined by several coordinates or vertices. After the programmer feeds the sets to GPU, the vertex programs process the graphics in terms of the coordinates or vertices. In this step, GPU assembles the triangles to form geometrical shapes.

After the shapes are formed, the geometry programs such as rotation, transposition, and scaling can be started. Then, all shapes go into the rasterization process which is projecting the shapes into 2D ready to be visible in the screen.

When the shapes are projected, they are transformed into (x,y) locations in the screens or pixels. The pixel programs can then begin to operate on the graphics. The last operation is hidden surface removal. There usually are some shapes sharing the same location in the screen with other shapes. If they are simply put to the screen sequentially, the last shape will take the location and impede previous shapes for being visible. Thus, the processing operation for the previous shapes is wasteful. The GPU solves this problem by examining the depth of the shapes and removing the hidden surface in this operation.

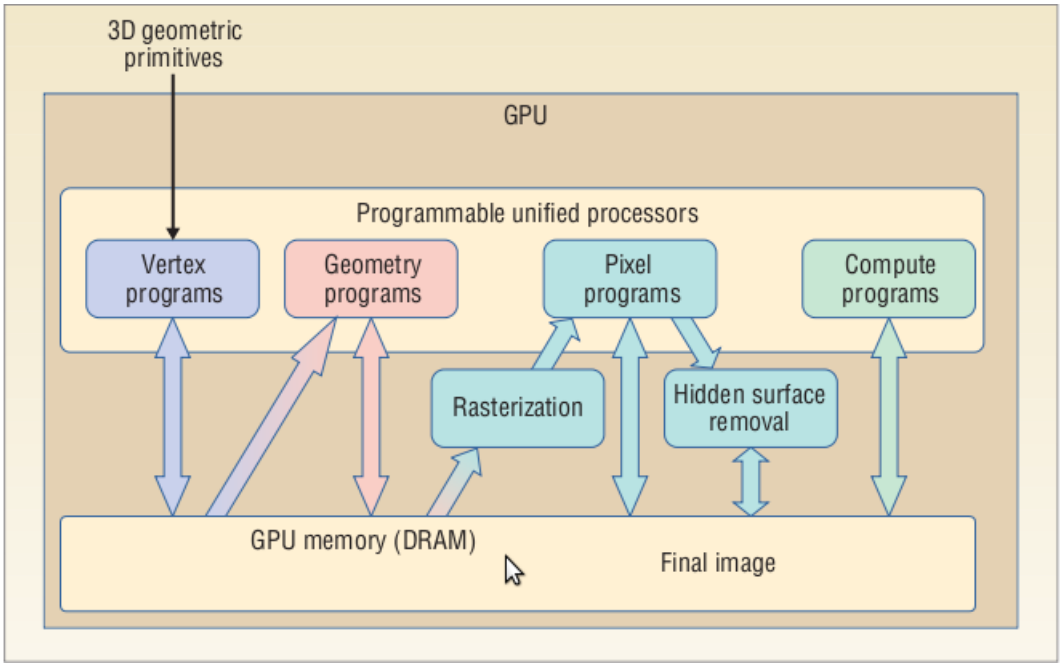


Figure 2.1: GPU architecture [21]

2.1.2 GPU processing resources

The processing discussed in previous section drives the allocation of GPU processing resources. The processing resources must support highly data

parallel operations because there are many pixels and vertices to be processed at the same time. In other words, computing each pixel or vertex sequentially is too slow considering the size of the problems.

There are three kinds of processing resources including Arithmetic Logic Unit (ALU), cache, and control unit. ALU as shown in figure 2.2 works by taking two inputs (A,B), calculating the result according to the function (F) specified by the control unit, and returning the result (R) back.

The cache is the second highest memory in the memory hierarchy, after only the processor registers. It is responsible for storing computed result for very fast retrieval later without slow memory access. The control unit is in charge of decoding instructions for the processor to process.

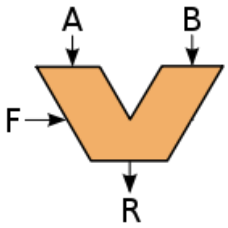


Figure 2.2: Arithmetic Logic Unit [41]

As illustrated in figure 2.3, GPU has significantly more ALUs but less control units and caches than CPU. In other words, the GPU focuses more resources on data processing than CPU.

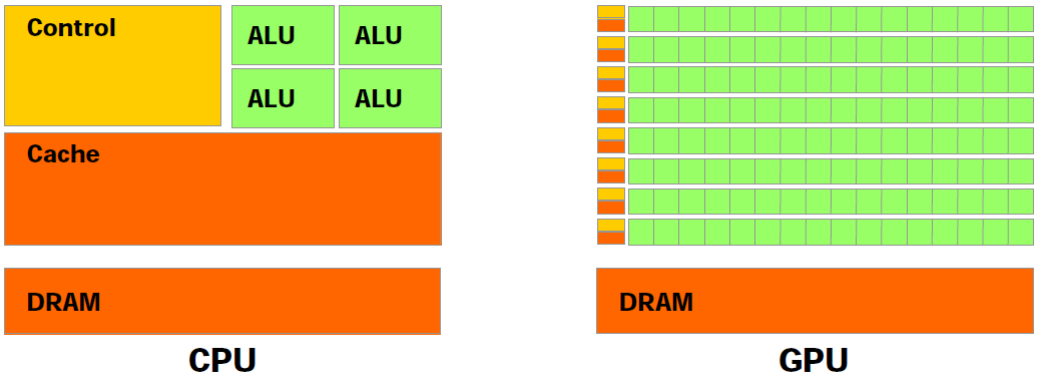


Figure 2.3: CPU and GPU processing resources comparison [30]

2.2 Network processing functions

This section explains the overview of network processing functions selected as the case studies for this thesis. Section 2.2.1 discusses layer 2 Bloom filter forwarding. Section 2.2.2 elaborates flow lookup in Openflow switch.

2.2.1 Layer 2 Bloom filter forwarding

This section explains the layer 2 Bloom filter forwarding. It discusses the introduction of Bloom filter and how to apply the Bloom filter for layer 2 forwarding respectively.

Bloom filter

Bloom filter is a space efficient data structure for supporting membership query [5]. It relies on hash functions to test whether an element is a member of a set. In doing so, it trades performance and space efficiency with the false positive. The false positive comes from the possibility of a hash collision. The filter provides two operations including adding member and querying membership status. The working principle of the Bloom filter is followed:

- The filter's data structure is a bit array of m bits. This array is used for adding and querying members in the filter.
- To add a member, the member is hashed by k different hash functions. Then, corresponding bit positions in the bit array are set according to the hash results.
- To query for a member, the member is again hashed by k different hash functions. And, whether all bits corresponding to the hash results are set determines the query result. If all bits are set, the result is true. Otherwise, the result is false.

From the working principle discussed above, let examine the false positive rate. Let n be the number of members are going to be kept in the Bloom filter and assume that the results of k hash functions are uniformly distributed. After setting bits in the bit array m according to k different hash functions for n members, the probability that one particular bit of the result is not set or still zero is $(1 - \frac{1}{m})^{kn}$. Then, the false positive occurs when all k hash

results are in the same bits being set before. It then can be calculated as shown in equation 2.1 [6].

$$(1 - (1 - \frac{1}{m})^{kn})^k \approx (1 - e^{\frac{-kn}{m}})^k \tag{2.1}$$

Layer 2 Bloom filter forwarding

Layer 2 Bloom filter forwarding is using Bloom filter to lookup the next hop in the switch network [46]. This is done by creating a Bloom filter for every next hop in a switch. Then, destination Ethernet addresses are stored in the Bloom filters. When a packet arrives, the packet's Ethernet destination address is checked with the Bloom filters. If the destination address matches with a Bloom filter for an output port, the switch forwards the packet to that output port. Because the Bloom filter guarantees no false negative, the packet is guaranteed to reach its destination. The motivation for using Bloom filters instead of a simple hash table is to fit the forwarding tables into fast memory such as CPU cache. With this approach, [46] reported 10% better forwarding performance compared to a simple hash table implementation.

The caveat of the Bloom filter is its false positive. Using Bloom filter with layer 2 forwarding results in some packets being forwarded to an output port which they should not be. This wastes resources and reduces network utilization. Nonetheless, the false positive rate is adjustable by the number of hash functions k and size of bit array m as discussed in section 2.2.1.

2.2.2 Flow lookup in Openflow switch

Openflow is an open standard for decoupling the control and data path in a switch. It aims to provide a highly configurable and scalable switch. It works with two separated components including a controller and an Openflow switch as figure 2.4 illustrated.

The controller can be in the same machine or other machine on the network. The controller controls the Openflow switch via the secure channel communicating with the Openflow protocol. The controller defines the logic of the switch by a flow in the flow table. Each flow is accompanied with the flow action executed by the switch if a packet is matched against the flow. The example actions are dropping the packet or forwarding the packet to a given port.

Each flow consists of 12 fields as shown in table 2.1. Not every field is

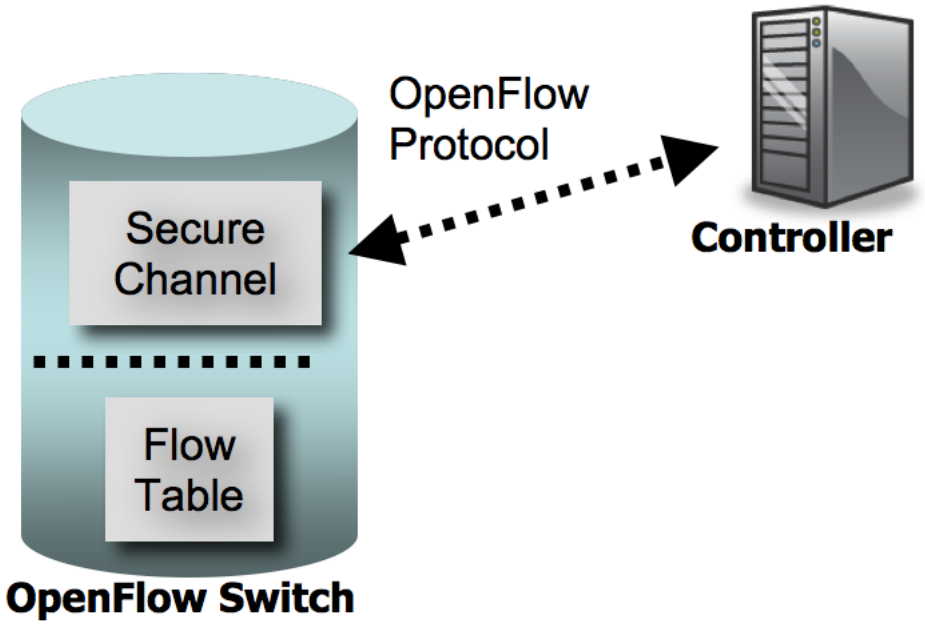


Figure 2.4: Openflow switch [31]

applicable for every packet. This depends on the packet type as noted in the last column of the table. Each field inside the flow can be specified with exact or *ANY* value. If the flow contains at least one field specified with ANY value, the flow is a *wildcard matching flow*. Otherwise, the flow is an *exact matching flow*.

The flow matching in the flow table works as followed: If both exact and wildcard matching flow match a packet, the exact matching flow is always selected. Each wildcard flow has a priority value as an attribute. If multiple wildcard flows match the same packet, the wildcard flow with the highest priority should be picked. The priority value can be the same for different wildcard flows. Hence, it is possible that there is more than one flow with the highest priority. In that case, one of the flows can be selected.

2.3 Related works

There are existing works on offloading network processing functions to GPU in different areas. The areas include intrusion detection, signature matching,

No.	Field	When applicable
1	Ingress Port	Every packet
2	Ethernet Source Address	Every packet on enabled-ports
3	Ethernet Destination Address	Every packet on enabled-ports
4	Ethernet Type	Every packet on enabled-ports
5	VLAN id	Every packet with Ethernet Type equal to 0x8100
6	VLAN priority	Every packet with Ethernet Type equal to 0x8100
7	IP source address	Every packet with Ethernet Type equal to 0x0800(IP) and 0x0806(ARP)
8	IP destination address	Every packet with Ethernet Type equal to 0x0800(IP) and 0x0806(ARP)
9	IP Protocol	Every IP, IP over ethernet, and ARP packet
10	IP ToS bits	Every packet with Ethernet Type equal to 0x0800(IP)
11	Transport Source port / ICMP Type	Every TCP, UDP, and ICMP packet
12	Transport destination port / ICMP Code	Every TCP, UDP, and ICMP packet

Table 2.1: Openflow header fields [31]

and network coding.

[40] augmented the performance of Snort [33], Opensource intrusion detection system, by offloading pattern matching operations from CPU to GPU. This was done by implementing the Aho-Corasick algorithm [2], multi-pattern matching algorithm, in GPU. With the implementation, they reported the throughput of 2.3 Gbit/s of artificial network trace. For online processing, the performance is double compared to the traditional CPU-only Snort.

[38] made a prototype implementation of network signature matching on GPU. They compared between Deterministic Finite Automata (DFA) and Extended Finite Automata (XFA) matching [37]. While XFA consumes less memory than DFA, the XFA processing is more complex than DFA. They ran a test on several network protocols including HTTP, FTP, and SMTP.

The CPU model in the test is Pentium 4 whereas the GPU model is Nvidia G80. The test results are 8.6 and 6.7 times better than CPU-only implementation for DFA and XFA respectively. Despite smaller memory access time, XFA performs poorer because of the complex control flows causing divergent threads.

[14] explained GPU-offloading network coding implementation. The implementation included both the complete offloading to GPU and CPU-GPU coordination. In the CPU-GPU coordination encoding, they systematically divided the work between CPU and GPU by proposing α as the percentage of work done in GPU. The rest of the work is done in CPU. They tested on different α values and found the optimal encoding throughput as 116.7 MB/s for $\alpha = 51\%$. In addition to the systematic coordination, they found that the network decoding is less suitable than the encoding for offloading to GPU. This is because the decoding requires result from previous calculation while encoding does not. Because of the problem, their decoding performed poorer than the CPU-only implementation.

2.4 Summary

This chapter summarized the background for the reader to understand this thesis including GPU, network processing functions, and related works. From the topics discussed, the reader is assumed to have enough background to understand the following chapters. On next chapter, the discussion moves to the properties of GPGPU including its historical development, suitable applications, and network processing data path.

Chapter 3

GPGPU

High data computation power as discussed in section 2.1.2 is a motivation for using GPU as a general purpose computing platform (GPGPU). This chapter focuses on properties of the GPGPU. Section 3.1 elaborates its historical development. Section 3.2 discusses the suitable applications. Section 3.3 explains the network processing data path including its organization and capacity.

3.1 Historical development

Many researchers investigated and developed applications with GPGPU in the past. The development started from graphical APIs, then by a vendor-specific general purpose APIs, and very recently with the Open Standard technology.

Researchers began to utilize the GPU computing power for general purpose applications with the graphical APIs such as OpenGL [20] and DirectX [22]. The APIs were vertex and pixel programs discussed in section 2.1.1. The pixel programs were more popular because they were easier to map to general purpose problems than the vertex programs [9]. In the pixel programs, temporary data was kept in graphical textures. For example, [23] and [9] used textures as a temporary memory to offload pattern matching computation and AES encryption to GPU respectively.

Nonetheless, the programmer had to map his/her problem domain to a graphical programming model. This mapping required significant efforts and background knowledge in the graphical programming because the APIs were not designed for that purpose [23, 44]. The development of general purpose APIs

for GPU such as CUDA [24], ATI Stream SDK [3], and DirectCompute (based on DirectX version 11 [22]) offers solutions to the problem. The APIs enable developers to create general purpose applications in GPU easier and more efficient than before. For instance, it is possible for the programmer to use random access memory which is never applicable before with the graphical APIs.

Even though the APIs discussed above are more suitable for general purpose applications on the GPU, all of them have the vendor lock-in issue. In other words, the APIs are supported only in particular devices or platforms. For example, CUDA is supported only on Nvidia graphic cards or ATI Stream SDK can be run only on ATI cards. This makes an application developed on one platform not able to be run on another platform.

This vendor lock-in problem may be solved with the coming of the Open Computing Language (OpenCL). OpenCL is an Open standard API for developing software in heterogeneous computing environment [19]. The standard includes APIs for controlling and programming a GPU device from different vendors provided that the OpenCL driver for the device is available.

Although OpenCL looks like a bright future, there are two major problems with the current status of OpenCL including immaturity and performance. OpenCL is very new. As an illustration, the graphical card manufacturers recently released their stable OpenCL drivers: ATI on October 2009 [4] and Nvidia on December 2009 [28]. This indicates that few people have tried the system for practical implementations.

OpenCL also imposes the trading of performance for portability. To actually achieve the best performance, one has to fully utilize the underlying computing architecture, for example, using particular memory location to optimize the access latency [29]. Nonetheless, programming in OpenCL does not allow the programmer to directly do so because it does not offer low level mapping to a particular platform.

Because of the two problems discussed above, this thesis made a decision to focus on vendor specific APIs such as CUDA or ATI Streaming SDK. Then, the thesis selected the CUDA architecture because the CUDA came first in the industry and thus has better community support. The CUDA software development will be further discussed in chapter 4.

3.2 Suitable applications

As shown in section 2.1.2, GPU architecture focuses on the data processing unit much more than the control flow and caching unit. As a result, the applications suitable to be run on GPU should be data intensive, no complex control flows, and few relying on large caches.

The applications should be data intensive or have high data to instruction ratio. The data then can be mapped to each ALU easily. In other words, the applications should process similar operations on different input data. As a result, multiple ALUs can process different inputs simultaneously with the same instruction.

Some ALUs share the same instruction unit. Hence, if there are two groups of ALUs would like to process different control flows, both control flows have to be taken sequentially. This will reduce the system utilization. As a result, the applications with complex control flows are not suitable for GPU.

The GPU's caches are typically smaller than CPU's caches. For example, the CPU's L2 cache in the HP xw8400 workstation, test machine for this thesis, is 2 MB whereas the texture and constant cache for the GPU are 6-8 and 8 KB correspondingly. Hence, the application that depends heavily on large caches to perform well is not appropriate for GPU.

3.3 Network processing data path

This section illustrates the data path involved in GPGPU with primary focus on GPU for network processing. Hence, it concentrates on the data path between GPU, CPU, Network Interface Card (NIC), and random access memory (RAM). The data path includes its organization and capacity.

The typical data path organization is shown in figure 3.1. The Northbridge is responsible for moving data between the central processing unit (CPU) and fast devices including RAM and Graphic card. On the other hand, the Southbridge is in charge of slower devices, for example, network adapters and harddisks.

The capacity of the data path varies between systems because of the different buses and chipsets embedded inside the motherboards. Nonetheless, the capacity for the HP xw8400 work station [16] used in this thesis is shown in figure 3.2. As illustrated from the figure, the bottleneck is in the connection between NIC and the Southbridge because it has the lowest capacity

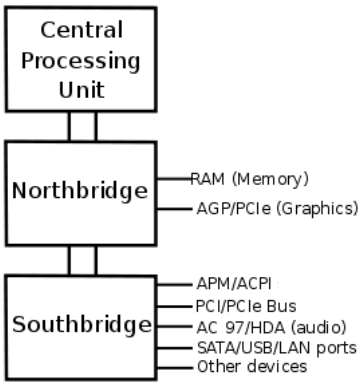


Figure 3.1: GPU data path organization [43]

(6.4Gb/s).

In addition to the bottleneck, another important point is the comparison of CPU and GPU datapath. As clearly visible from the figure, the bandwidth between GPU and RAM (32Gb/s) is significantly lower than CPU and RAM (68Gb/s). Hence, reducing the data transfer between GPU and RAM by transferring only to-be-processed data is mandatory in order to gain benefits compared to traditional CPU-only approach.

In the network processing context, this reduction requires a preprocessing of network packets. The preprocessing is extracting the relevant data from the packets and send only that data to GPU. For example, if the network processing function is the IP prefix lookup, the only required data is the packet's destination IP address. As a consequence, only the destination IP address is needed to be transferred to GPU. The whole packet can be kept temporarily in main memory while the GPU is processing. This will minimize the problematic data transfer.

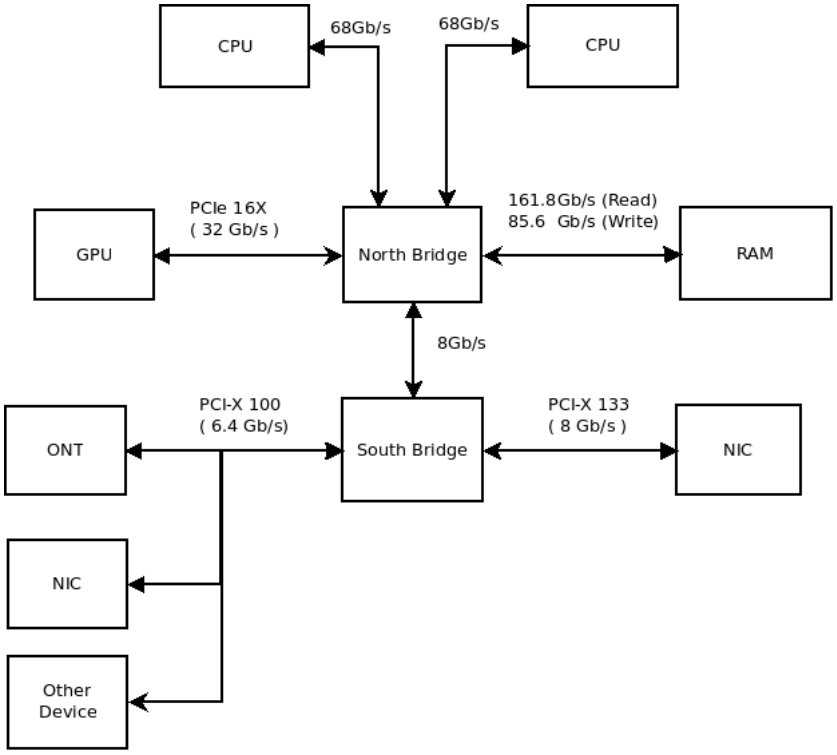


Figure 3.2: GPU data path capacities of the test environment

3.4 Summary

This chapter discusses the properties of GPGPU including historical development, suitable applications, and the network processing data path. From the section 2.1.2 and the issues discussed in the section 3.2, the conclusion is that GPU and CPU are appropriate for different operations. While GPU is good at processing a lot of data with simple control flows, CPU is good at processing limited amount of data with possible complex control flows and cache utilization. Thus, the best practice is to use both the GPU and CPU together. This is done by letting the CPU manage the complex control flow, pack the data, and ask the GPU to process the data intensive operations. This best practice requires a software development managing both CPU and GPU. This thesis selected CUDA as a platform for the development. The CUDA software development or CUDA programming will be discussed in next chapter.

Chapter 4

CUDA programming

As stated the justification in section 3.1, this thesis focuses on Compute Unified Device Architecture (CUDA) from Nvidia. This chapter aims to discuss software development environment of CUDA for managing the collaboration between CPUs and GPUs. Section 4.1 states the compilation and execution of a CUDA program. Section 4.2 elaborates its hardware architecture. Section 4.3 shows its programming model. Section 4.4 focuses on the important aspects of CUDA application performance. Section 4.5 discusses the heterogeneous computing between CPU and GPU. Section 4.6 explains the multiple GPUs programming.

4.1 Compilation and execution

A CUDA sourcecode is a mixing of code for both CPU and GPU [29]. The code for CPU can be programmed with C++ language while the code for GPU must be defined with C language. The code then is compiled with the *nvcc*, CUDA C compiler, as shown in figure 4.1. If the CUDA sourcecode made a call to existing optimized library such as mathematic, Basic Linear Algebra Subprograms (BLAS), and Fast Fourier Transform (FFT), the library would be included in the compilation as well.

There are two parts resulted from the compilation including the machine independent assembly (PTX)[27] and CPU host code. Then, the PTX is loaded for execution in GPU with the CUDA driver. During the time of execution, the developer can perform runtime investigation with the debugger and profiler tool. For the CPU host code, it is compiled with standard C compiler such as GCC to be run on CPU.

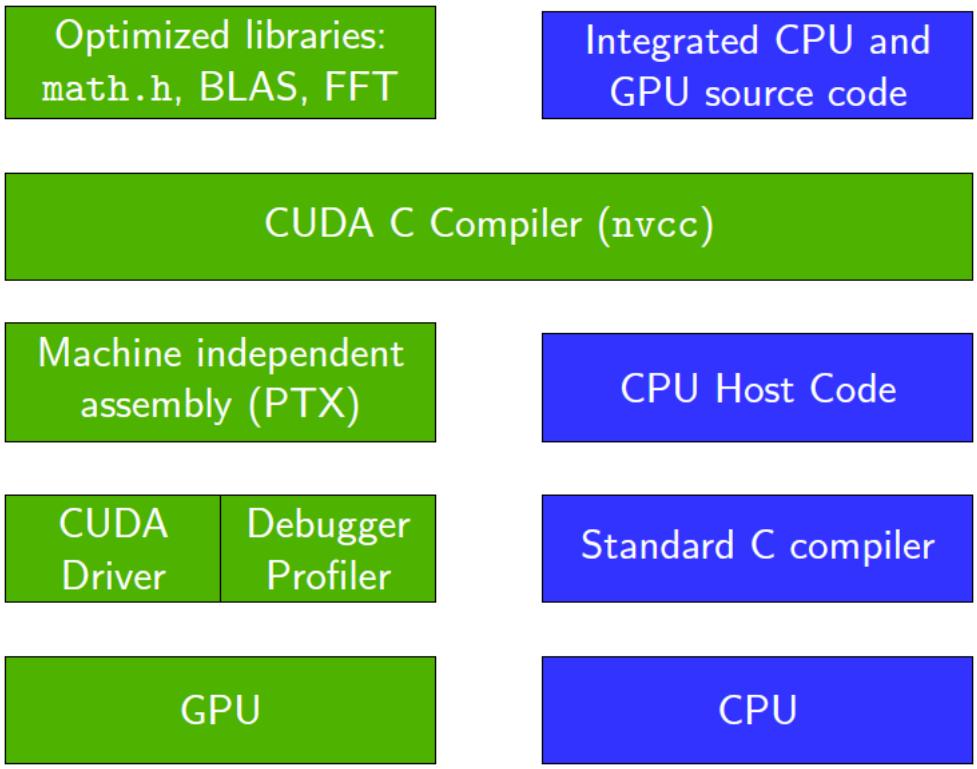


Figure 4.1: CUDA compilation and execution [35]

4.2 CUDA architecture

The CUDA architecture consists of several multiprocessors as shown in figure 4.2. The numbers of multiprocessors are different between GPU models. Each multiprocessor has 8 stream processors with a common instruction unit, shared memory (16Kb), constant cache (8Kb), and texture cache (6-8 Kb) [29]. The shared instruction unit makes all stream processors within a multiprocessor work in the Single Instruction Multiple Data (SIMD) manner. In other words, executing many stream processors simultaneously within a particular multiprocessor requires the instruction to be the same.

The capacities of the on-chip memories including the registers, shared memory, constant cache, and texture cache are quite limited as shown above. The stream processor has an option to keep data in a very larger memory namely device memory. Nonetheless, the device memory is off-chip and shared among all multiprocessors [29]. As a result, its latency is significantly higher than

the on-chip memories.

The on-chip memories discussed above are suitable for different situations [29]. The shared memory is the only writable memory. The constant cache is fully optimized when multiple threads access the same memory address. On the other hand, the texture cache is designed for 2D spatial locality. The locality occurs when threads next to each other in x-axis or y-axis access the memory location next to each other in x-axis or y-axis as well.

Although the shared memory is the only memory that is both writable and fast, it has one major drawback over other memories. The drawback is that its life time is limited to a *CUDA thread block*. The CUDA thread block is elaborated in detail in next section. After the block finishes execution, the data in the memory is undefined [29]. Because of the limited life time, every new block has to load the data to the memory before execution. This loading is wasteful when the data is expected to be the same for all blocks. On the contrary, the device, constant, and texture memory do not have this limited life time. And, the data can be loaded only once and used by all blocks afterward.

4.3 Programming model

Programming applications in CUDA architecture can be thought as a Single Instruction Multiple Thread (SIMT) programming [30]. In SIMT, the programmer defines the same code to be executed by multiple threads. As shown in figure 4.3, the multiple threads are organized in a thread block and the thread blocks are organized in a grid. One thread executes concurrently and independently with other threads. The synchronization can be done only for the threads within the same block. Each thread has a thread ID namely *threadIdx* to identify the position within the block and block ID namely *blockIdx* to identify the position within the grid. Both the threads and blocks can be specified in 1, 2, or 3 dimensions. The decision on the dimensions depends totally on the applications of GPGPU.

The grid of thread blocks in CUDA is called *kernel*. To execute the code in GPU, the programmer must invoke the kernel in GPU (device) from the CPU (host) as shown in figure 4.4.

After the kernel is invoked, the available multiprocessors are given thread blocks to execute. The thread blocks scheduled to be executed in a multiprocessor are called *active thread blocks* in the multiprocessor. There is a limitation to the number of active thread blocks per multiprocessor. This

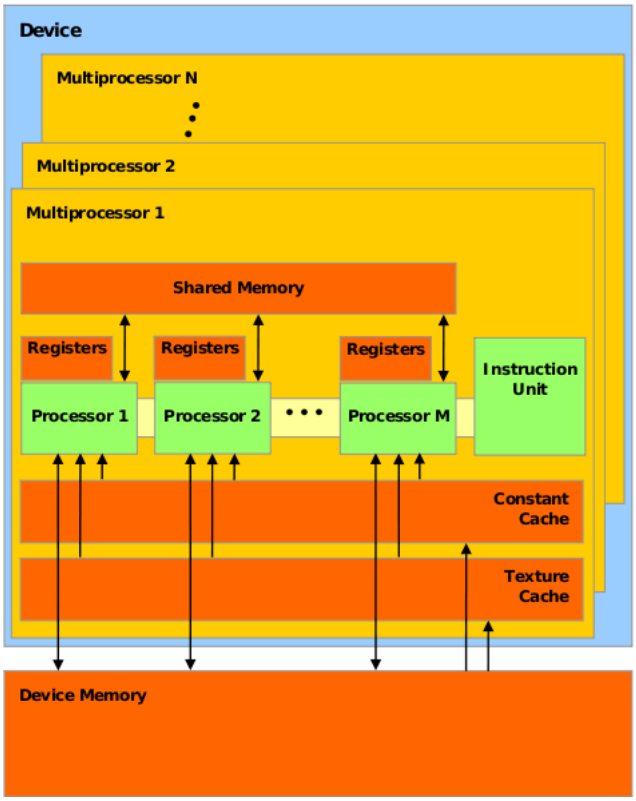


Figure 4.2: GPU architecture [30]

number depends on the GPU model and resources needed by the blocks. The final number can be obtained by using CUDA profiler tools including CUDA visual profiler [26] and CUDA occupancy calculator [25].

Each multiprocessor executes a thread block by dividing the threads in the block into a smaller execution unit called *warp*. The warp always consists of *warp size* threads. The warp size for the current CUDA architecture is 32. The division is done deterministically. As an illustration, each warp takes consecutive 32 threads by the thread IDs from the block. Then, when a multiprocessor is free, the SIMT scheduler schedules the next warp to be executed in the multiprocessor.

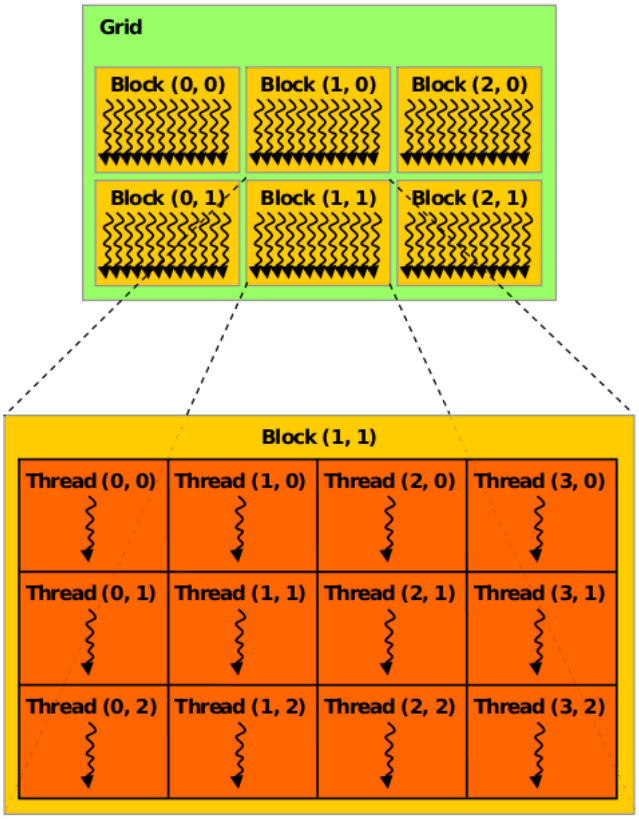


Figure 4.3: Grid of blocks in SIMT [30]

4.4 CUDA performance

This section elaborates the major aspects influencing the CUDA application performance including warp divergence, latency hiding, and memory conflicts. As discussed in section 4.3, warp is a group of 32 threads executed together by the multiprocessor. Then, if the threads inside the warp take different control flows, all flows are executed sequentially, and some threads are disabled on particular flows. This situation is called a *warp divergence*. As illustrated in figure 4.5, after the threads in a warp test with the evaluation process (Eval P) such as if-else code, threads are enabled only on some execution paths. Because of this warp divergence, the CUDA programmer can only obtain best performance by making all threads in the warp take the same control flow.

As elaborated in section 4.2, the memory bandwidth to the device memory

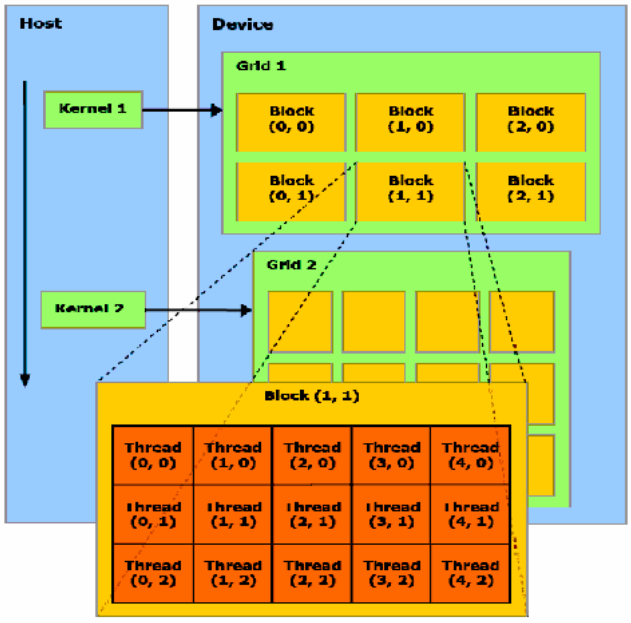


Figure 4.4: Invoking kernel for execution in GPU [29]

is quite slow and thus has high latency. The CUDA multiprocessor hides this latency by swapping the warp accessing the memory with other warps ready to do some computations. As shown in figure 4.6, when *warp 0* is accessing the slow memory, *warp 1* is swapped in to do some computations. This continues to happen until *warp 0* finishes accessing the memory and ready to do the computations again. This way the throughput of the system is not dropped because of the latency. Nonetheless, in order for this *latency hiding* to happen, the application must have enough computation steps and active warps in the multiprocessor.

CUDA threads explained in section 4.3 have to access the device memory in particular access pattern to achieve the best performance. The pattern is that contiguous threads access contiguous memory locations as shown in figure 4.7. This gives the best performance because the memory access for the threads can be packed together into one memory call. The pattern resembles the way pixels are processed in the graphical applications where contiguous threads process contiguous pixels. If the threads do not follow the pattern such as an access in figure 4.8, there will be *memory conflicts*. With the memory conflicts occur, the accesses have to be done sequentially and result in worse performance.

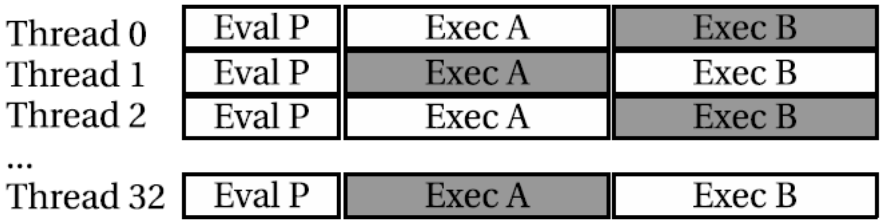


Figure 4.5: Warp divergence [39]

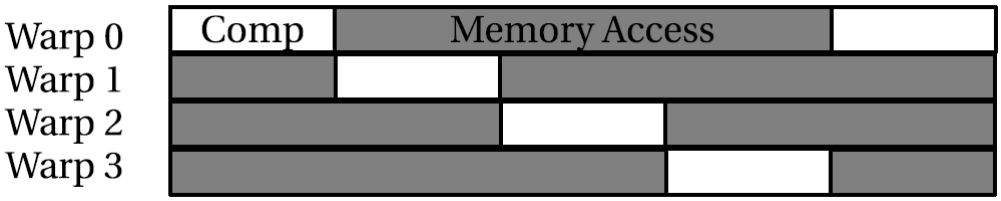


Figure 4.6: Warp latency hiding [39]

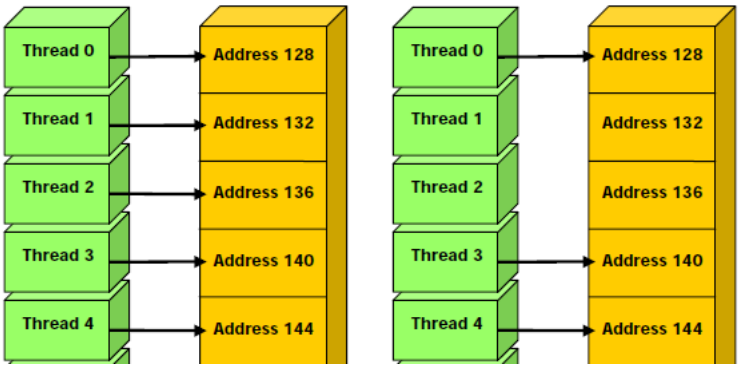


Figure 4.7: The packing together memory access pattern for the device memory [29]

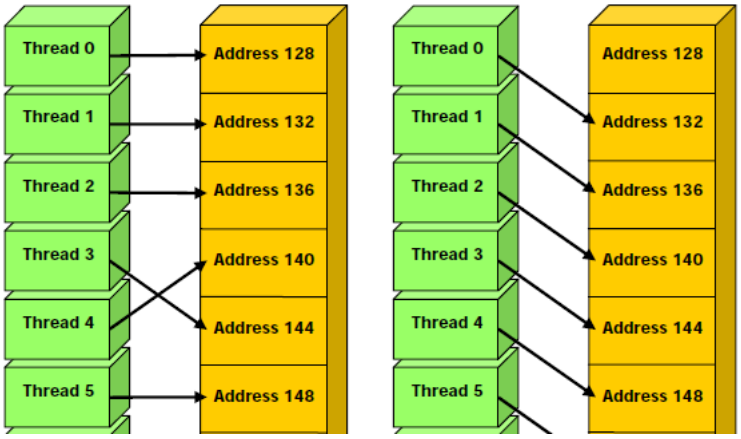


Figure 4.8: Memory conflict access pattern for the device memory [29]

4.5 Heterogeneous computing between CPU and GPU

The typical processing flow of CUDA applications between CPU and GPU consists of four steps as illustrated in figure 4.9. The steps are copying data from the main memory to GPU's memory, CPU ordering the GPU to process the copied data, GPU processing the copied data by multiple cores in parallel, and copying the processed result back to the main memory.

The copying to be processed data from the main memory to the GPU's memory is done by the CUDA function calls. The function calls are CUDA memory allocation namely *cudaMalloc()* followed by the CUDA memory copy function call namely *cudaMemcpy()* with flag *cudaMemcpyHostToDevice* [29]. Then, the CPU instructs the GPU to process the data by the kernel invocation discussed in section 4.3.

To copy the data back to the main memory, the *cudaMemcpy()* should be called again with different flag namely *cudaMemcpyDeviceToHost*. After the application finishes using the allocated memory, the CUDA device's memory free function namely *cudaFree()* should be called.

With several iterations of the CUDA processing flow discussed above in a typical CUDA application, the execution characteristic is a mixing of CPU and GPU execution. As shown in figure 4.10, while CPU (Host) is responsible for serial execution with single thread, GPU (Device) is in charge of parallel

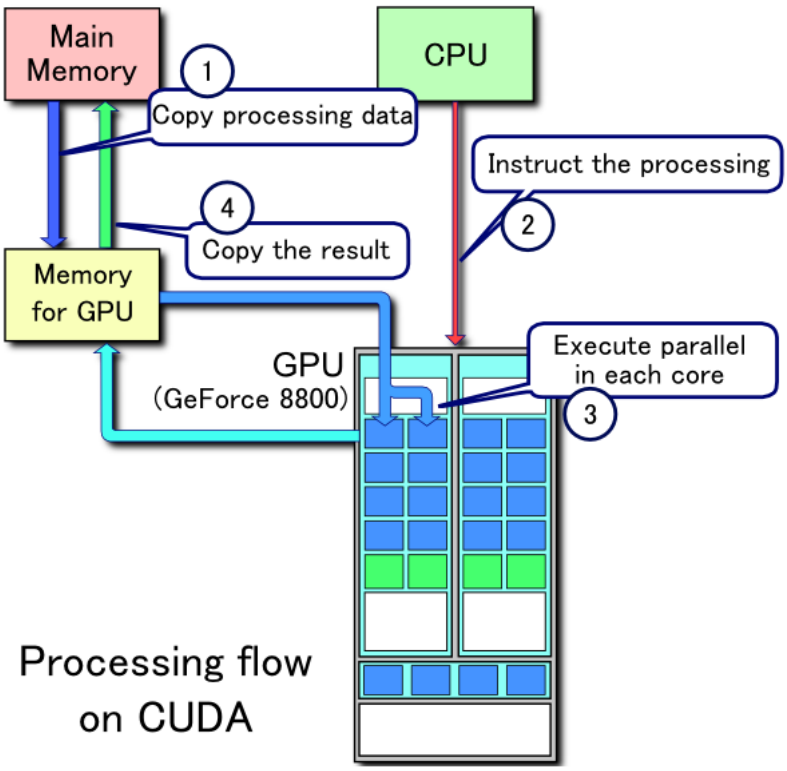


Figure 4.9: CUDA processing flow [42]

execution with multiple threads. This is how CPUs and GPUs work together in the heterogeneous environment.

4.6 Multiple GPUs

This section discusses the motivations and different configurations for multiple GPUs. Running applications on many GPUs is the promising solution to overcome the limited resources on a single GPU such as the on-chip memories explained in section 4.2. This can be done by distributing problems over multiple GPUs. For instance, [7] solves the fast Conjugate Gradient by distributing the rows of a matrix to several GPUs.

The multiple GPUs mean even more data parallel execution units. This indicates the potential of even higher performance than a single GPU. Many existing works already started using multiple GPUs: [13] got a GPU cluster 4.6 times faster than CPU cluster on lattice Boltzmann model. [1] used a

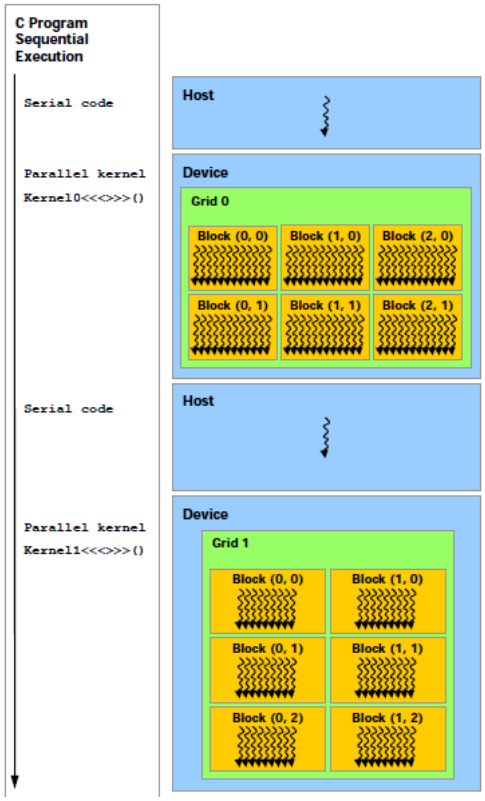


Figure 4.10: Heterogeneous CPU and GPU [29]

cluster of GPU and CPU to calculate prices of European market options.

Even though running multiple GPUs indicates many benefits as discussed above, it creates a new challenge which is the communication between GPUs. The communication between several GPUs depends heavily on configurations of the system. [34] categorized the configurations into shared system GPUs and distributed GPUs configuration. The shared system GPUs configuration is a one single host having multiple GPUs inside. The communication is done via the shared main memory in the system. Programming CUDA in this system requires multiple CPU threads because of the CUDA restriction. The restriction is that a CPU thread can manage only one GPU device at a time [29]. Hence, controlling several GPUs requires several CPU threads.

Distributed GPUs configuration is a setup which multiple GPU-capable hosts connect to each other via a network. The communication between GPUs depends totally on the network connection. As a result, programming CUDA in this configuration resembles parallel programming in distributed CPUs

system where a message passing is mandatory.

4.7 Summary

The CUDA is a platform for managing computation between CPUs and GPUs. This chapter discusses the software development on CUDA including compilation/execution, architecture, programming model, performance aspects, heterogeneous system between CPUs and GPUs, and multiple GPUs. This software development is a stepping stone to an analysis of GPGPU applications. Nonetheless, how does the development apply to the context of network processing? This would be addressed in next chapter.

Chapter 5

GPU for network processing

This chapter elaborates the usage of GPU in the context of network processing. Section 5.1 shows GPU-friendly network processing functions. Section 5.2 states general tradeoffs in offloading network processing functions to GPU. Section 5.3 explains the major performance limitation in the offloading. Section 5.4 discusses the selected case studies for this thesis.

5.1 GPU-friendly network processing functions

This section explains the criteria for network processing functions suitable to be offloaded to GPU. The criteria are no inter-packet dependency, similar processing flows for all packets, and within-packet parallel processing opportunity.

The function should have *no inter-packet dependency*. In other words, the processing of the function should not rely on other packet's processing result. With this condition, processing multiple packets in GPU at the same time is possible. The processing is done by assigning each CUDA thread to a packet. As stated in section 4.3, the CUDA threads work concurrently and independently with other threads.

The second condition is *similar processing flows for all packets*. Although multiple packets can be executed concurrently when the first condition is satisfied, the different processing flows between packets reduce the concurrency level. This results from the warp divergence explained in section 4.4. With the warp divergence, both flows are taken sequentially. Thus, the divergence reduces the benefits of executing the packet processing in GPU.

The last condition for the functions is *within-packet parallel processing opportunity*. The functions have the opportunity when some tasks in the functions for processing an individual packet can be done at the same time. In addition to the concurrent packet processing discussed above, with this condition applied, a programmer designing an algorithm for GPU has another option. The option is processing several parts within a single packet simultaneously. This speeds up the individual packet processing as well.

5.2 General tradeoffs

This section elaborates the tradeoffs involved in offloading network processing function to GPU compared to traditional CPU-only approach. The offloading trades higher latency and memory consumption for higher throughput.

A typical CPU-only network processing application processes a network packet sequentially. In other words, it processes the packet immediately without a delay. This makes the processing have low latency (seconds/packet). However, offloading the network processing function to GPU must change the application logic from a sequential processing to a batch processing. This is because batching multiple packets compensates the data transfer startup overhead. As stated in section 4.5, processing data in GPU requires a data transfer from the RAM to GPU's memory and vice versa. The data transfer has a high startup overhead. If a packet is processed individually, this overhead would significantly reduce the throughput of the system as shown in [40].

In addition to the latency, the batch processing also implicates higher memory consumption. This is because the packets have to be kept temporarily in the main memory instead of being processed immediately. Hence, more packets in order to compensate the overhead means more memory consumption required for the GPU offloading network processing application. The benefit would be gained from trading the two items is higher throughput (packets/second). The throughput is higher because of the higher number of data processing units or ALUs as explained in section 2.1.2. In other words, all packets in the batch are processed faster in GPU than in CPU which they are processed sequentially.

5.3 Performance limitation

Every network processing application requires a network packet reception and transmission from/to physical or virtual NICs. There is no exception with the GPU-offloading application. There are two implementation locations for the reception and transmission including kernel space and user space. The kernel space applications always have better performance in network packet reception and transmission than the user space applications. This is because they can receive or transmit network packets with lower overhead. For example, no context switching to the user space is required. This is the reason why most of the network processing applications, for instance, Linux IP router, stateless firewall, and IPSec gateway are implemented in kernel space.

However, processing network packets in GPU has to be in user space because of the availability of the GPGPU APIs. The GPGPU APIs such as CUDA are only available from the user space. As a result, implementing network processing applications from such APIs always results in user space applications. This translates to the lower rate the GPU-offloading applications can receive or transmit network packets compared to the kernel space applications.

There are two solutions to this limitation but are beyond the scope of this thesis. The solutions are increasing the packet capturing and transmitting rate in the user space such as [11, 10] or making the GPGPU APIs available in the kernel space.

5.4 Case studies selection

This section explains the case studies selected for this thesis based on the criteria in section 5.1. The selected case studies are layer 2 Bloom filter forwarding and the flow lookup in Openflow switch. The basic operations of the functions are elaborated in section 2.2.

The layer 2 Bloom filter forwarding meets all of the criteria. The calculation of every packet is independent from other packets and the processing flows between different packets are the same. In other words, several packets can be processed concurrently with full power of GPUs.

The within-packet parallel processing opportunities are in a calculation of different Bloom filters and k hash functions. The calculation for a Bloom filter does not depend on another Bloom filter's result. As a result, multiple

Bloom filters can be processed in parallel for a single packet. And, because the hash functions are interdependent to each other, the functions can be calculated concurrently. This speeds up the processing of a single packet even more.

The flow lookup also fits all criteria. The flow lookup of an individual packet does not depend on the flow lookup results of other packets. Hence, the lookup for multiple packets can be processed concurrently.

The within-packet parallel processing opportunity is in the flow table. The flow table can be divided into several parts and each CUDA thread discussed in section 4.3 can be responsible for each part.

5.5 Summary

This chapter introduces the GPU in the context of network processing. It begins by explaining the general criteria for network processing functions suitable for offloading to GPU and general tradeoffs in the offloading. It continues to elaborate the performance limitation of this offloading. And, it ends with the discussion of the selected case studies based on the criteria. The discussion only introduces the possibility of GPU offloading but it does not explain how to practically implement the offloading. The practical explanation is in chapter 6 and 7 for layer 2 Bloom filter forwarding and flow lookup in Openflow switch correspondingly.

Chapter 6

GPU layer 2 Bloom filter forwarding

This chapter elaborates the detail of the first case study, GPU layer 2 Bloom filter forwarding. Section 6.1 discusses implementation alternatives for the case study and its analysis in term of performance, software development efforts, and advantages/disadvantages. Section 6.2 explains the evaluation methodology.

6.1 Implementation alternatives

Multiple dimensions of the alternatives are proposed sequentially. The prior dimensions influence subsequent dimensions. As a result, the reader is suggested to read step by step from the first dimension. Section 6.1.1 discusses the algorithm alternatives. Section 6.1.2 provides the CUDA thread organization alternatives. Section 6.1.3 explains the hash function alternatives. Section 6.1.4 elaborates the memory mapping alternatives. The selected alternatives from all dimensions are summarized in table 6.2.

6.1.1 Algorithm alternatives

As stated in section 5.4, GPU layer 2 Bloom filter forwarding has 3 parallel opportunities including packets, network interfaces, and hash functions. As a result, algorithm alternatives base on how many opportunities should be utilized. The alternatives are summarized in table 6.1. The number of opportunities implemented is proportional to the performance but also to the

system complexity. The higher system complexity indicates higher software development efforts required in order to design and implement such system. The number 3 with all three dimensions executed in parallel is selected because of the best performance and utilization of GPGPU computing.

No.	Packets	Network Interfaces	Hash functions
1	Parallel	Sequential	Sequential
2	Parallel	Parallel	Sequential
3	**Parallel**	**Parallel**	**Parallel**

Table 6.1: Algorithm alternatives

6.1.2 CUDA thread organization alternatives

The previous section concludes that packets, network interfaces, and hash functions should be executed in parallel. To make the parallel execution possible, the CUDA threads are designed as 3 dimensional threads as shown in figure 6.1. The threads with the same x values are responsible for the same packets. The threads with the same y values work on the same Bloom filters or network interfaces. The threads with the same z values are in charge of the same hash functions.

The reason that the packets have to be in the x axis is for optimizing memory access pattern. The thread in the x axis will be the first to be divided into the warp stated in section 4.3 followed by y and z axis. As a result, if the x axis is chosen for the packets, threads on the same warp will operate on different packets in such a way that adjacent threads access or will access adjacent packets. In other words, the thread i accesses packet i when the thread $i+1$ accesses packet $i+1$ and so on. This is the good memory access pattern for device memory and yields high performance as shown in section 4.4. On the other hand, swapping the axis between the Bloom filters and hash functions will make no difference.

6.1.3 Hash function alternatives

Multiple hash functions are required for the Bloom filter as explained in section 2.2.1. This section discusses two alternatives for implementing such hash functions including real and salt based different hash functions.

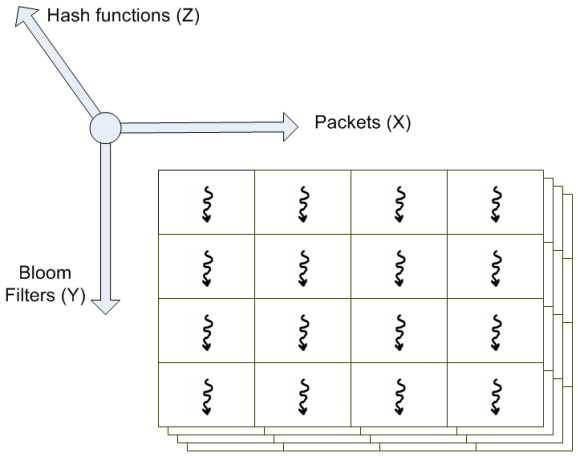


Figure 6.1: CUDA thread organization alternatives

Real different hash functions are done by implementing completely different calculation steps for different hash functions. This guarantees the diversity of the calculation results and does not put a restriction on the hash functions. Nonetheless, the required software development efforts for this alternative are proportional to the number of hash functions because all of the functions have to be developed individually. This alternative is not suitable because the number of hash functions is assumed to be arbitrary to support wide range of false positive rate for the given m and n elaborated in section 2.2.1.

To workaround the limitation, another method to generate different hash functions is proposed and selected for implementation. The method is salt based hash function. The salt based hash function uses only one base hash function but generates new hash functions by modifying a single bit before sending the modified input to the base hash function. This method allows arbitrary number of different hash functions with limited software development efforts needed.

Nonetheless, this creates a restriction on the base hash function. In order to guarantee the randomness of the generated hash functions, the base hash function needs to have the avalanche effect [15]. The avalanche effect is the property of a hash function in which modifying a single bit of the input results in totally different hash result. SHA1 and MD5 are proposed as alternatives for the base hash function because they have the property and are widely studied. Then, among the two, MD5 is selected because it is less computation intensive than SHA1 and existing Opensource implementation in GPU is publicly available [18].

6.1.4 Memory mapping alternatives

After MD5-based hash function is selected, the next alternative is how to map the processing elements to the GPU memory. In other words, this alternative discusses where to put the data to be processed on the CUDA architecture. The CUDA architecture has different memory types including device, shared, constant, and texture memory as stated in section 4.2. Each of them has different capacity and is optimized for different situations. The processing elements include destination MAC addresses of the packets, MD5 calculation, MD5 constants, and Bloom filters.

The MAC addresses of the packets have to be in the device memory because the number of packets is expected to be very high. Mapping the MAC addresses to other memory locations does not fit at all.

MD5 needs a location to store temporary calculation result while processing [32]. For the CUDA architecture, this can be mapped only to either device or shared memory because the location has to be writable. Nonetheless, shared memory is selected because MD5 calculation is computation intensive. In other words, threads access this temporary result many times and they should have the lowest memory access latency.

MD5 computation requires round-dependent MD5 constants [32]. The constants should be on the constant cache because the constants are small enough to fit in the cache and read-only. In addition, the constants are accessed as a round-based basis. In other words, if the constants are loaded to an array, the memory location that will be accessed at round i can be expressed with index $f(i)$ to the array. This access pattern maps very well to the characteristics of constant memory where it is optimized when all threads in the warp accessing the same location as stated in section 4.2.

The last processing element to select a memory location is the group of Bloom filters. The group must be rapidly accessed by multiple threads so the choices are limited to on-chip memories including shared memory, constant cache, and texture cache. Then, the shared memory is removed because the forwarding tables or Bloom filters are expected to be the same for all CUDA thread blocks. Putting the data that is expected to remain static between the blocks requires wasteful loading as explained in section 4.2.

Then, among the constant and texture cache, the texture cache is selected because of the memory access pattern. As shown in figure 6.1, the threads with the same x and z value operate on the same packet and hash function respectively. Hence, given a group of threads where x and z are the same, they have the same hash result and lookup in the adjacent memory locations

for the Bloom filters in the y dimension. Intuitively, if the Bloom filters are put in the texture cache, the processing will have the spatial locality in the y dimension.

Formally speaking, let T be a set of threads where x and z are the same and $t_i \in T$ has $y = i$. Because x and z are the same in T, all t_i have the same hash result namely h_k. And, let $tex2D(r_x, r_y)$ be a texture access function for Bloom filter lookup where r_x is the bit position in the Bloom filter and r_y is the Bloom filter index. The access pattern is that t_i accesses $tex2D(h_k, i)$ while t_{i+1} accesses $tex2D(h_k, i+1)$ and so on. This is the 2D spatial locality in the y dimension and optimized access pattern for the texture cache as stated in section 4.2.

Alternative	Decision
Algorithm alternatives	
Algorithm	Parallel packets Parallel Bloom filters Parallel hash functions
Thread organization alternatives	
Thread X Thread Y Thread Z	Packets Bloom filters Hash functions
Hash function alternatives	
Multiple hash functions Base hash functions	Salt based MD5
Memory mapping alternatives	
Packets' MAC address MD5 Calculation MD5 Table Bloom Filter	Device memory Shared memory Constant memory Texture memory

Table 6.2: Summarization of the system design decision for the layer 2 Bloom filter forwarding

6.2 Evaluation methodology

The evaluation based on the comparison of offline processing performance between the CPU-only and GPU-offloading implementation. Both implementations are asked to process a batch of artificial packets with layer 2 Bloom filter forwarding. The processing time is then measured for different tests. The tests are discussed and reported in chapter 8.

The packets are simulated by preloading data into the main memory. The data is not the whole packet but just 6 bytes value representing the destination MAC addresses. However, when the data is moved to GPU, it is moved in an 8-byte buffer to align the memory with multiple of 32 bits for maximizing the access performance in GPU.

The CPU model here is a Quad-core Intel Xeon Processor 5300 inside the HP xw8400 workstation. Nonetheless, only one CPU core is active because the software is not multi-threaded. On the contrary, the GPU model is Geforce GTX 275 with 240 cores and all cores are active because of the parallel programming.

The number of output ports in the configuration is 8. As a result, there are also 8 Bloom filters. The Bloom filters have to fit in the texture cache of the GPU which size is 6 Kbytes. As a result, the size of each Bloom filter is 768 bytes. The number of hash functions is 8. With this configuration, the system can store up to 360 MAC addresses / Bloom filter or 2880 MAC addresses in total while the maximum false positive rate is 0.038%. The false positive rate is calculated from the equation 2.1. As stated in section 6.1.2, the threads are organized in 3 dimensions. The thread organization for this setup is (x,y,z) = (8,8,4).

The processing time for CPU-only implementation includes only the CPU calculation time because no extra operation is needed. On the other hand, processing in GPU requires also initialization and data transfer. The initialization is for starting up the GPU for processing general purpose application, loading constants, and allocating memory. The data transfer is for moving to-be-processed data between the main memory and the GPU memory as stated in section 4.5. As a result, the processing time for GPU-offloading implementation consists of GPU initialization time, input data transfer time, calculation time, and output data transfer time. The detail description of each individual GPU processing time is provided in table 6.3.

Time	Description
Total CPU processing Time	CPU calculation Time
Total GPU processing Time	GPU initialization time + GPU input data transfer time + GPU calculation time + GPU output transfer time
GPU initialization time	Time for starting the GPU for CUDA programming, loading MD5 constants to constant memory, loading Bloom filter to texture memory, and allocating memory for packets
GPU input data transfer time	Time for transferring input packets' MAC address from main memory to GPU memory. The workload to transfer is 8 bytes / packet
GPU Calculation time	Time for calculating layer 2 Bloom filter forwarding by multiple cores in GPU
GPU Output data transfer time	Time for transferring output forwarding result from GPU's memory to main memory. The result is 1 byte / output port / packet. Because there are 8 output ports, the workload to transfer is 8 bytes / packet.

Table 6.3: Layer 2 Bloom filter forwarding processing time measurement

6.3 Summary

This chapter discusses the detail of the first case study, GPU layer 2 Bloom filter forwarding. The issues discussed are the implementation alternatives, analysis of the alternatives, and the evaluation methodology. This case study is implemented and the test result is provided and analyzed in chapter 8. The next chapter focuses on the detail of the second case study, GPU flow lookup in Openflow switch.

Chapter 7

GPU flow lookup in Openflow switch

This chapter explains the second network processing case study for this thesis, GPU flow lookup in Openflow switch. Section 7.1 discusses the implementation alternatives for the case study. Section 7.2 provides the analysis in term of performance, software development efforts, and advantages / disadvantages. Section 7.3 elaborates the evaluation methodology for this case study.

7.1 Implementation alternatives

Two alternatives are proposed here including GPU hash-then-linear in section 7.1.1 and GPU flow exact pattern lookup in section 7.1.2.

7.1.1 GPU Hash-then-linear lookup

Hash-then-linear lookup is implemented in the Openflow reference implementation version 1.0. The idea is using the hashing lookup for an exact matching flow followed by the linear lookup for a wildcard matching flow as illustrated in figure 7.1. The definition of the exact and wildcard matching flow are elaborated in section 2.2.2. If the hashing lookup fails, the search continues to the linear lookup in order to find the wildcard matching flow. If the linear lookup also fails, the packet is then reported not matched.

As discussed in section 2.2.2, the wildcard matching flow that should be

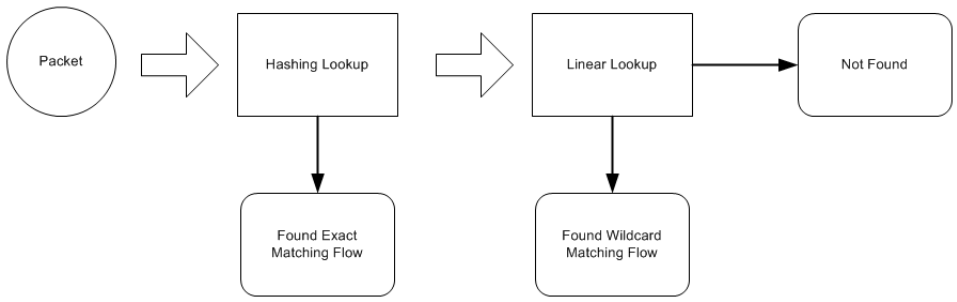


Figure 7.1: Hash-then-linear lookup

selected is the matched flow with the highest priority. The linear lookup searches for this flow by inserting the flow according to its priority and then looking up from the highest priority flow as shown in figure 7.2.

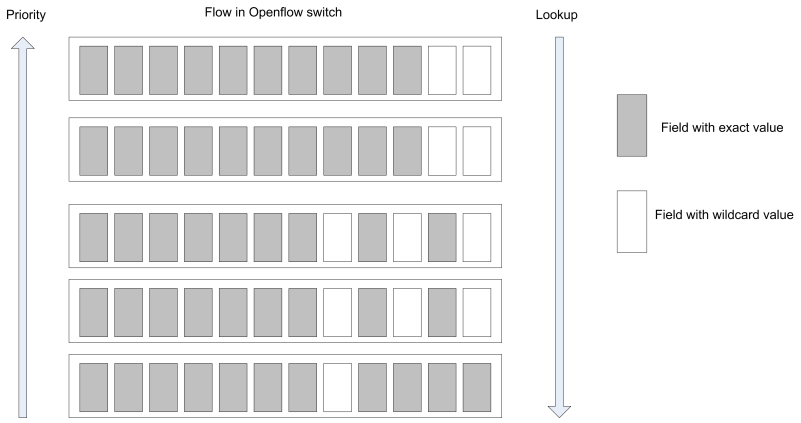


Figure 7.2: Linear lookup for wildcard matching flow

Then, the GPU hash-then-linear lookup works in the same way as the hash-then-linear lookup. The working principle is simply moving all lookups into the CUDA kernel as illustrated in figure 7.3. As a result, multiple packets and flows can be looked up in parallel. Nonetheless, this algorithm has a severe performance problem if it is implemented in GPUs. The problem comes from the fact that this lookup breaks the second condition of GPU-friendly network processing functions, similar processing flows for all packets. The condition is elaborated in detail in section 5.1.

As visible from figure 7.1, the processing flows of this lookup depend on the content of the packets. If the processing flows are different, both of the

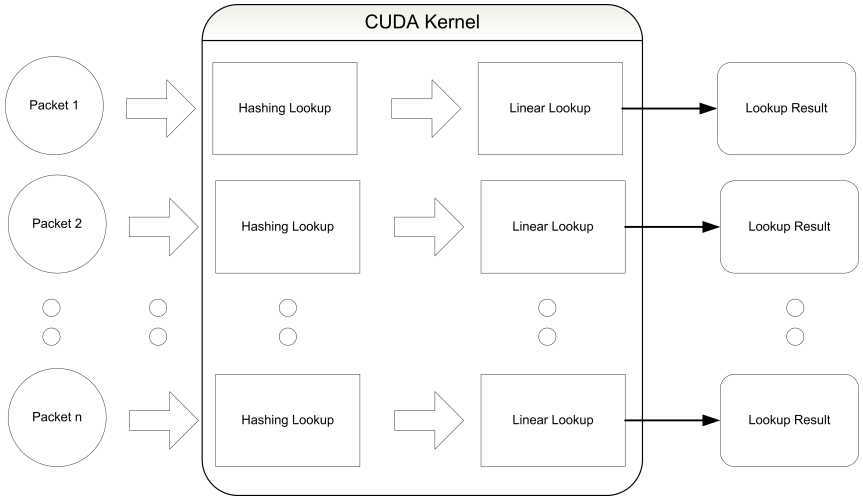


Figure 7.3: GPU hash-then-linear lookup

flows have to be taken sequentially and the utilization of the GPU is heavily diminished. The formal explanation of this problem is discussed in section 7.2. This problem motivated the author to design and propose a new flow lookup algorithm that is ideal for SIMD processors like GPU. The algorithm is explained in detail in next section.

7.1.2 GPU flow exact pattern lookup

The GPU flow exact pattern lookup takes different approach than the hash-then-linear by using hashing lookup for both exact and wildcard matching flow. This is done by hashing lookups in the *flow exact pattern* hash tables. Flow exact pattern is the pattern for grouping flows with similar exact value fields in the flow table. Hence, the number of flow exact pattern is always equal or less than then number of flows. Each flow exact pattern has its own hash table for storing the flows within the pattern as shown in figure 7.4.

The number of flow exact patterns depends on the flows in the flow table but the maximum can be calculated. The maximum number of flow exact patterns is the number of 12-field combinations plus one, special pattern which every field is a wildcard field. Hence, the number can be calculated as followed:

$$\binom{12}{12} + \binom{12}{11} + \binom{12}{10} + ... + \binom{12}{3} + \binom{12}{2} + \binom{12}{1} + 1 = 4096$$

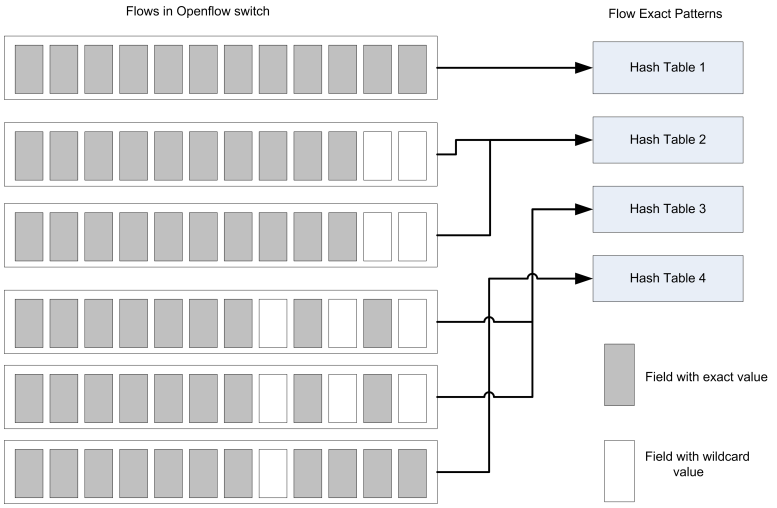


Figure 7.4: Flow exact patterns

Lookup a packet in the flow exact pattern lookup consists of two phases including *parallel flow exact pattern lookup* and *parallel flow selection* phase as shown in figure 7.5. In the parallel flow exact pattern lookup phase, the lookup for a single packet works according to algorithm 1.

As stated in the algorithm, the flow exact patterns are distributed equally among different CUDA threads T. And, the operations inside the loop including $LF(e)$, $Priority(f)$, and comparisons are constant time operations. Hence, the computation complexity for each thread in this algorithm is $O(E/T)$ where E is the number of flow exact patterns (maximum number is 4096) and T is the number of CUDA threads assigned to the single packet. The output from this phase is the $MaxF$ array containing matched flow indexes with local max priority and the array size is T. This array is an input to the second phase.

In the second phase, the $MaxF$ is searched by the parallel flow selection in order to find the flow with maximum priority. Parallel flow selection divides the works among the CUDA threads assigned to the single packet. The search iterates through several rounds until the flow with maximum priority is found as shown in figure 7.6. Let t be the number of CUDA threads active and n be the number of flows to search in each round. The first round starts with $t = n/2$. In each round, both n and t are reduced by half. X_i is the flow index to the flow table at location i of the input array $MaxF$. The arrows represent the priority comparison between X_i and X_j. M_{ij} indicates the flow index with maximum priority from location i to j. The search proceeds until

Algorithm 1 Parallel flow exact pattern lookup for a single packet

$T \leftarrow$ Set of CUDA threads assigned to the single packet
$T_i \leftarrow$ CUDA thread index i
$LF(e) \leftarrow$ Lookup a flow index from a flow exact pattern hash table according to the input flow exact pattern e, return -1 if not found, otherwise return the flow index
$Priority(f) \leftarrow$ Lookup the priority value from input flow index f, return minimum priority value if f is -1, always return maximum priority for exact matching flow
$N(X) \leftarrow$ Number of elements in Set X
$MaxF \leftarrow$ Shared array across T. The array keeps flow indexes with local max priority. The array size is $N(T)$
$MaxF_i \leftarrow$ Flow index with local maximum priority for each T_i. Initialize at -1
$E_i \leftarrow$ Set of flow exact patterns distributed equally to T_i
for each T_i concurrently **do**
  for each $e \in E_i$ **do**
    $f \leftarrow LF(e)$
    if $f \neq -1$ and $Priority(f) > Priority(MaxF_i)$ **then**
      $MaxF_i \leftarrow f$
    end if
  end for
end for

n is 2 and t is 1 where one last comparison gives the final answer.

The parallel flow selection demands $O(log_2 n)$ for each thread where n is the number of input flows. Because the number of input flows from the first phase by the array $MaxF$ is T, this phase demands $O(log_2 T)$ for each thread. Combining both of the phases, the total computation complexity for the GPU flow exact pattern lookup is $O(E/T + log_2 T)$.

7.2 Analysis

This section provides an analysis between the two alternatives in term of performance, software development efforts, and advantages / disadvantages. The summarization of the analysis is in table 7.1.

Looking up flows in both of the alternatives has to be done in a batch of packets in order to compensate the overhead as explained in section 5.2.

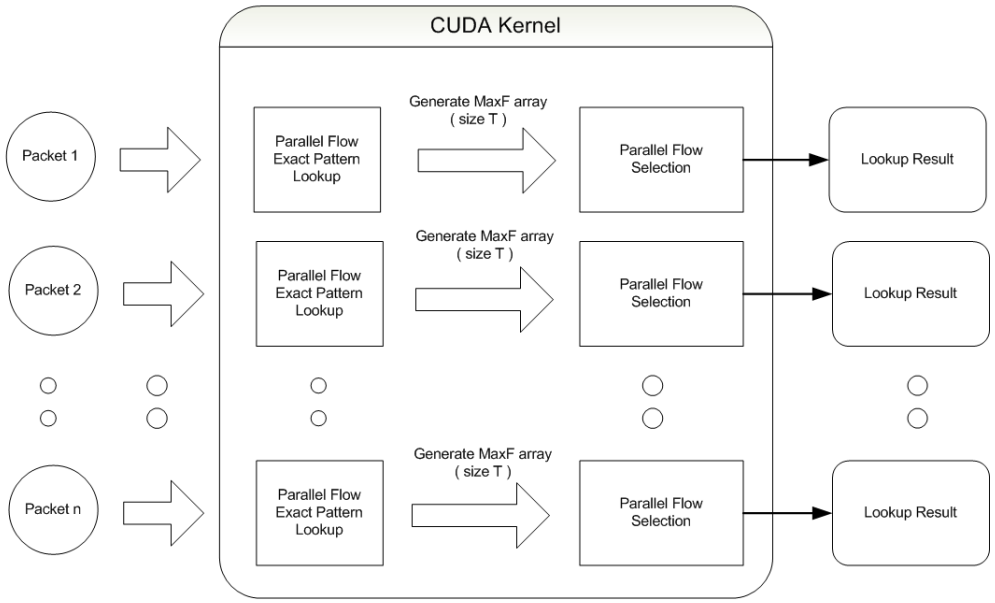


Figure 7.5: flow exact pattern lookup

Flow matching in the batch depends on the incoming packets' content. The matching is in three different cases. First, if all packets in the batch match with only exact matching flows, this lookup is an *exact batch lookup*. Second, if all packets in the batch match with only wildcard matching flows, this lookup is a *wildcard batch lookup*. Lastly, if all packets in the batch match with a mixing of exact matching flows and wildcard matching flows, this lookup is a *mixing of exact and wildcard batch lookup*.

For the exact batch lookup, the GPU hash-then-linear lookup requires less computation than the GPU flow exact pattern lookup. While the GPU hash-then-linear demands a single hashing calculation, the GPU flow exact pattern lookup requires $O(e/t)$ hashing calculations per packet (t is a number of concurrent threads, e is a number of flow exact patterns as discussed in section 7.1.2).

Nonetheless, for the wildcard batch lookup, the GPU hash-then-linear performs worse than the GPU flow exact pattern lookup. This is because the GPU hash-then-linear lookup has to perform a linear lookup and demands $O(n/t)$ wildcard comparisons (n is the number of flows in the flow table which is unbounded) while the GPU flow exact pattern lookup is still able to use the hashing lookup.

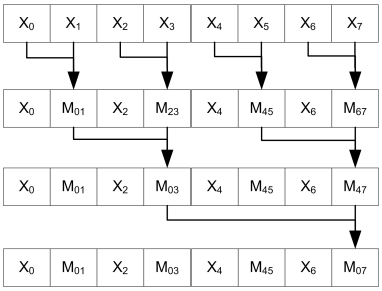


Figure 7.6: Parallel flow selection

The last case is the mixing of exact and wildcard batch lookup. In this case, for the GPU hash-then-linear lookup, some packets are processed with the hashing lookup while other packets are processed with the linear lookup in different CUDA threads. Nonetheless, the hashing lookup packets likely to finish first have to wait for the linear lookup packets because the CUDA kernel discussed in section 4.3 can not be returned unless all threads finish their executions. This makes the whole lookup degrade to the linear lookup. On the other hand, the GPU flow exact pattern lookup performs the same as hashing lookup in this case because the operations are not changed. To sum up, for the lookup operation, the GPU flow exact pattern lookup performs better than the GPU hash-then-linear lookup in every case except the exact batch lookup.

Nonetheless, the hash-then-linear lookup requires less software development efforts and can support subnet mask for the IP source and destination field easily. Both GPU hash-then-linear and GPU flow exact pattern lookup base on the Openflow reference implementation version 1.0. The reference implementation already uses the hash-then-linear lookup. Hence, implementing the GPU hash-then-linear lookup demands a modification only in the lookup operation. On the contrary, implementing the GPU flow exact pattern lookup requires a change of data structures inside the reference implementation. This change needs a modification to all operations including adding, deleting, editing, and lookup operation.

The Openflow specification states an optional support of the IP source and destination subnet mask [31]. This is easy to be done for the GPU hash-then-linear lookup. It demands only few extra computation steps when comparing a packet with the wildcard matching flows. However, this flow comparison is not possible for the GPU flow exact pattern lookup because the IP source and destination fields are totally hashed. If one would like to add this subnet mask

support to the GPU flow exact pattern lookup, he/she has to convert the subnet mask flow to multiple flows with exact values of IP source and destination. For example, adding a flow with IP destination 192.168.1.0/24, the flow must be converted to 2^8 flows rendering all possible values of 192.168.1.0/24 subnet. Considering all tradeoffs discussed above, the GPU flow exact pattern lookup is selected for the prototype implementation. Nonetheless, the subnet mask support is optional and thus not included.

Areas	GPU Hash-then-linear lookup	GPU flow exact pattern lookup
Performance (per packet)		
Exact batch lookup	hash lookup $O(1)$	hash lookup $O(e/t)$
Wildcard batch lookup	linear lookup $O(n/t)$	hash lookup $O(e/t)$, $e \leq n$, $e \leq 4096$
Mixing batch lookup	linear lookup $O(n/t)$	hash lookup $O(e/t)$, $e \leq n$, $e \leq 4096$
Software development efforts required		
Modification required	Lookup operation only	Adding, deleting, editing , and lookup operations
Advantages / disadvantages		
Subnet mask support	with single flow	with 2^s flows, s is number of bits in the mask

Table 7.1: Summarization of the analysis between the alternatives for the flow lookup in Openflow switch

7.3 Evaluation methodology

This section shows how to evaluate the GPU flow lookup system properties. The evaluation based on the comparison of online processing performance between the Openflow reference implementation version 1.0 and the GPU-offloading implementation with flow exact pattern lookup. Section 7.3.1 elaborates the evaluation setup. Section 7.3.2 discusses the different measurements based on the setup.

7.3.1 Setup

The evaluation setup consists of the HP xw8400 workstation and the IXIA, traffic generator and analyzer as shown in figure 7.7. The Openflow switch installed in the workstation forwards a packet to its destination port by looking up in the flow table.

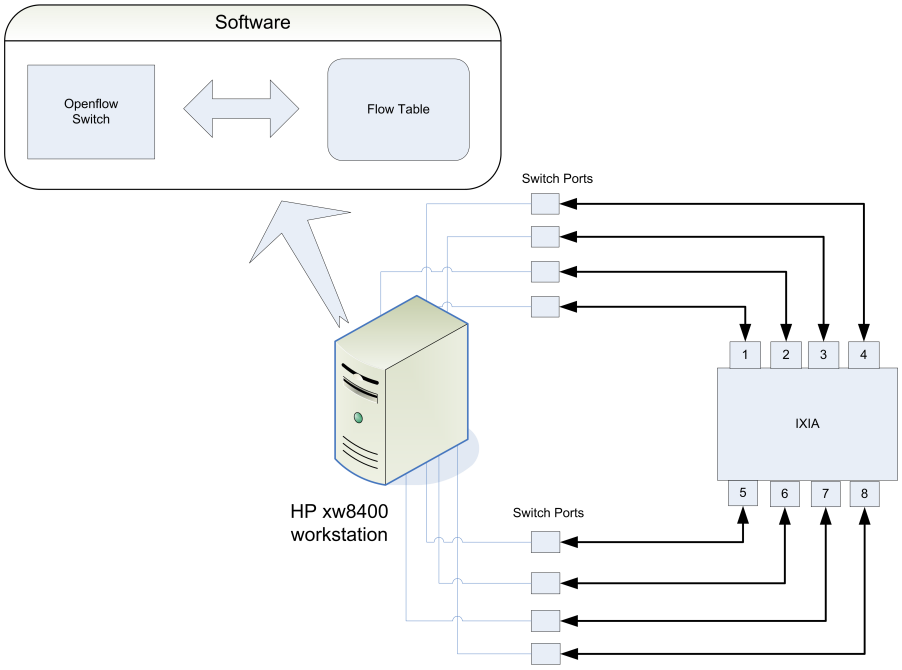


Figure 7.7: Flow lookup in Openflow switch evaluation setup

Then, the flow table in the Openflow switch discussed above is inserted with two sets of flows. Both sets are capable of forwarding a packet as shown in table 7.2. The sets are exact and wildcard matching flow set. The exact matching flow set is fixing all fields as illustrated in figure 7.8. The flows in the wildcard matching flow set have only one exact field which is the Ingress port, the port received by the switch. The other fields are wildcard as shown in figure 7.9.

7.3.2 Measurements

This section discusses the measurements for flow lookup in Openflow switch. The measurements focus on the tradeoffs discussed in section 5.2 including throughput, latency, and memory consumption. And, the tradeoffs become

No.	Forwarding between IXIA ports
1	1 <-> 5
2	2 <-> 6
3	3 <-> 7
4	4 <-> 8

Table 7.2: Packet forwarding in the evaluation setup of the flow lookup in Openflow switch

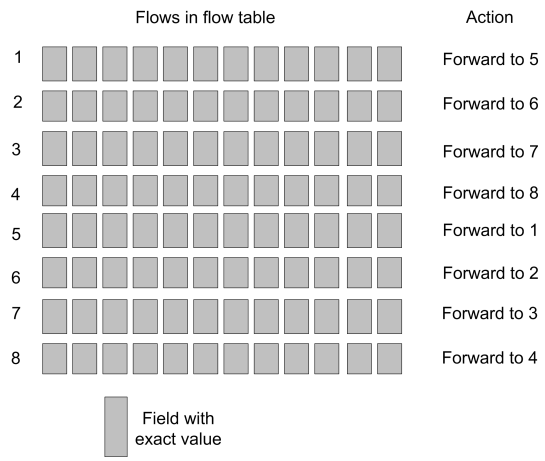


Figure 7.8: Exact flows in evaluation setup for flow lookup in Openflow switch

output variables of the measurements. The input variables for the measurements are the data rate and the processing complexity. There are two sets of measurements concentrating on different input variables. The first set is the exact flow lookup and the second set is the wildcard flow lookup.

The throughput and latency are obtained by the IXIA report console while the memory consumption is obtained by running monitoring tools in the workstation. The recorded output variables are plotted as graphs with y-axis as the output variables and x-axis as the input variables.

The exact flow lookup set concentrates on the packet processing in different data rates. All 8 ports in IXIA are asked to send to its corresponding destination in table 7.2. The input variable is the data rate defined by the summation of input frame rates for all IXIA ports. The frame sizes for all ports are fixed. 8 different data rates are tested. On each rate the Openflow reference implementation version 1.0 is run followed by the GPU-offloading

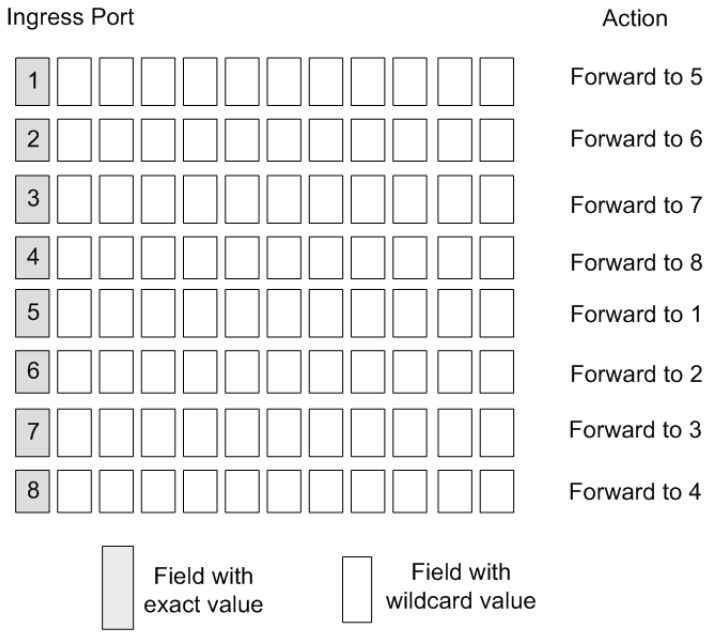


Figure 7.9: Wildcard flows in evaluation setup

implementation with flow exact pattern lookup.

The wildcard flow lookup set focuses on packet processing on different processing complexity. The input frame rate and frame size are fixed in this set. The complexity is defined by the wildcard flow index that matches a packet. There are 8 wildcard flows as illustrated in figure 7.9. As a result, 8 wildcard flow indexes are tested sequentially. On each wildcard flow index, the Openflow reference implementation version 1.0 is run followed by the GPU-offloading implementation with flow exact pattern lookup.

For the Openflow reference implementation, the higher wildcard flow index demands more computation steps than the lower index because of the linear lookup requirement. The minimum number of steps is in the first flow with only one step for each packet. The maximum number of steps is in the 8^{th} flow with 8 steps for every packet. Nonetheless, the GPU-offloading processing time for different wildcard flow index is expected to be the same because of the capability of the algorithm discussed in section 7.1.2.

7.4 Summary

This chapter discusses the detail of the second case study, GPU flow lookup in Openflow switch. The issues discussed are the implementation alternatives, analysis of the alternatives, and the evaluation methodology. The next chapter focuses on the analysis of the empirical results.

Chapter 8

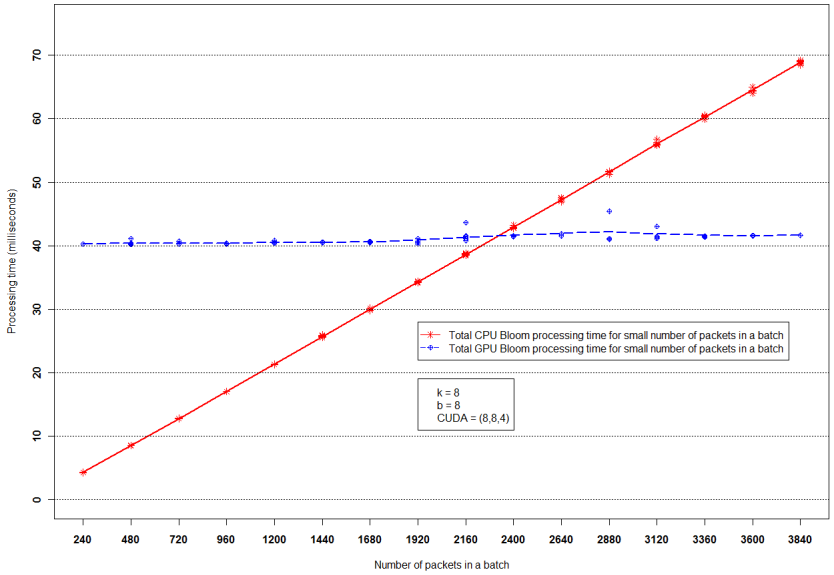
Analysis result

This chapter elaborates the analysis of the empirical results for the first case study, GPU layer 2 Bloom filter forwarding. The results based on the evaluation methodology discussed in section 6.2. Several tests are designed iteratively. In other words, the prior tests influence the design of subsequent tests. Section 8.1 shows the comparison of total processing time between CPU and GPU. Section 8.2 elaborates the GPU individual processing time test result. Section 8.3 explains preinitialized GPU processing test result. Section 8.4 discusses test results from variety of CUDA thread setups.

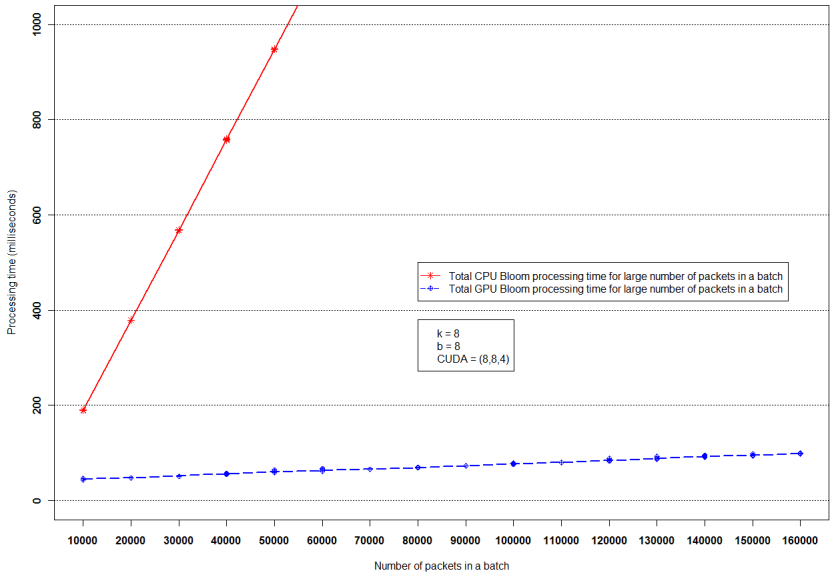
8.1 Comparison of total processing time between CPU and GPU

Figure 8.1 shows the total processing time of CPU and GPU on two configurations. The configurations are small (240-3840 packets) and large (10000-16000 packets) number of packets. Each of the packet size is tested and reported 5 times. Three observations are made here.

First, one can observe that for less than 2400 packets in the batch, CPU-only implementation takes less time to process than the GPU-offloading implementation. Nonetheless, when the number of packets in the batch increases, the processing time of CPU-only implementation grows faster than the GPU-offloading implementation. This goes on until the batch size equal to 2400 packets which the GPU-offloading implementation starts to take less time to process than the CPU-only implementation. Because the throughput (packets/second) is inversely proportional to the processing time for a batch of packets, the throughput in GPU-offloading implementation is higher than in



(a) configuration for small number of packets (240-3840 packets)



(b) configuration for large number of packets (10000-16000 packets)

Figure 8.1: Comparison of total processing time between CPU and GPU on layer 2 Bloom filter forwarding

CPU-only implementation from 2400 packets onward.

Second, the GPU-offloading implementation has higher latency (seconds/packet) than CPU-only implementation. This is because GPU-offloading implementation processes the packets as a batch. The latency of all packets in the batch is the same. On the other hand, CPU-only implementation processes the packets sequentially. Hence, its average latency is the batch processing time divided by the number of packets in the batch. For this test, the average latency for CPU-only implementation is $30/1680 = 0.0184$ milliseconds. This is significantly lower compared to the latency of GPU-offloading implementation for the same batch size which is approximately 41 milliseconds.

Third, even though, the higher number of packets in the batch means the higher throughput of using GPU-offloading over CPU-only implementation, extremely large batch is not always preferable. This is because larger batch indicates higher latency and memory consumption. From the configuration for large number of packets, one can observe that the higher number of packets increases the total processing time as well. And, the latency is proportional to the total processing time as stated above.

The larger batch size also means higher memory consumption. This is because more packets need to be kept temporarily in the main memory. The maximum Ethernet packet size is $s = 1500$ bytes [17]. As a result, the main memory should be preserved for the batch is $n * s$ bytes where n is the number of packets in a batch or batch size. The appropriate n then depends on the application and running platform in order to satisfy required latency and available main memory. It is worth to note that the three observations inline with the theoretical general tradeoffs discussed earlier in section 5.2.

8.2 GPU individual processing time

This section analyzes the processing time components of GPU-offloading implementation individually. The detailed description of the components is explained before in table 6.3. Each component from the configuration for small number of packets in figure 8.1 is measured and shown in table 8.1.

The results in the table clearly indicate that the major time consuming part is the GPU initialization time. If we assume that the GPU device is dedicated for network processing, one optimization can be done by preinitializing the GPU device at application boot time. Then, for the critical processing path, the initialization step is not repeated. This preinitialized version or the optimized GPU-offloading implementation is discussed in detail in next

section.

Individual Time (milliseconds)	Minimum	Maximum	Median	Mean
GPU Initialization Time	38.49	44.33	39.95	40.23
GPU Input Transfer Time	0.013	0.039	0.027	0.027
GPU Calculation Time	0.272	1.371	0.816	0.810
GPU Output Transfer Time	0.014	0.037	0.026	0.026

Table 8.1: GPU layer 2 Bloom filter forwarding individual processing time

Figure 8.2 elaborates the GPU input transfer time, calculation time, and output transfer time from the configuration for small number of packets in figure 8.1. Three points can be seen from the graph. First, the GPU calculation time grows faster as the number of packets increase than both the input and output data transfer time. In other words, the part taking most of time after the GPU initialization time is the GPU calculation time. Second, the input and output transfer time grow linearly as the number of packets in the batch increases. The workloads for both input and output transfer are 8 bytes/packet as explained in section 6.2.

Lastly, the GPU calculation time grows as a step function of 960 packets when the number of packets in the batch increases. In other words, one can observe that when the number of packets jumps from the multiple of 960 packets, the processing time increases more than jumping from other number of packets. For example, when the number of packets changes from 960 $\rightarrow$ 1200 the increased processing time is visibly higher than when the number of packets changes from 240 $\rightarrow$ 480, 480 $\rightarrow$ 720, 720 $\rightarrow$ 960, even though the number of packets changed is the same as 240 packets. This 960 as a step size comes from the *multiprocessor iteration* which is elaborated in next section. On the contrary, the processing time of CPU-only implementation grows linearly when the number of packets in the batch increases as shown in figure 8.1.

8.3 Preinitialized GPU processing

The preinitialized GPU-offloading implementation made an assumption that the GPU is dedicated for network processing. As a result, the initialization steps discussed in table 6.3 including time for starting GPU for CUDA programming, loading MD5 constants to constant memory, and loading Bloom filters to texture memory are executed at application boot time. Then, only necessary operations are executed in the critical data path. The operations

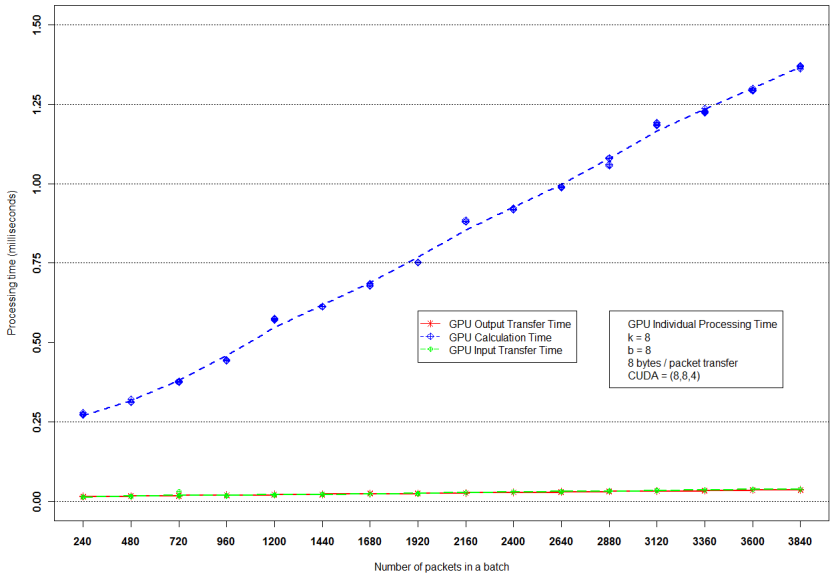


Figure 8.2: Individual processing time varied on number of packets

include allocating memory space for the packets, transferring the input data from the main memory to GPU's memory, calculation of layer 2 Bloom filter forwarding, and transferring the results back to the main memory.

Figure 8.3 shows the preinitialized GPU-offloading result. Each of the packet size is tested and reported 5 times. Three observations are made from the figure. First, one can see that the initialization time which is on average 40.23 milliseconds is significantly reduced. This shrinks the difference of processing latency between GPU and CPU processing explained in section 8.1. However, the GPU still has higher latency because the minimum latency visible from the graph is 0.25 milliseconds when the CPU latency is only 0.0184 milliseconds.

Second, the step function on 960 packets introduced before in previous section is also seen in the figure. One can observe the time gap when the number of packets increases from a multiple of 960 packets. This number of packets comes from the *multiprocessor iteration*. Multiprocessor iteration is the number of packets in the batch when all multiprocessors are completely filled with active thread blocks discussed in section 4.3. In other words, the number of active thread blocks reaches the maximum active thread blocks in the multiprocessors. Hence, when the number of packets increases more than the

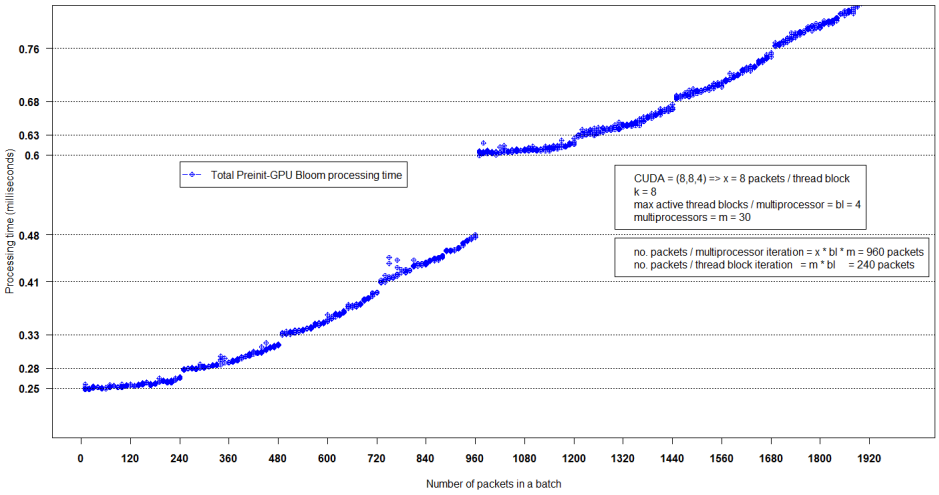


Figure 8.3: Preinitialized GPU total processing time test result

iteration, all thread blocks can not stay in the multiprocessors at the same time and have to be rescheduled for next iteration. This rescheduling creates the gap clearly seen in the figure.

The multiprocessor iteration for layer 2 Bloom filter forwarding can be calculated from $x * bl * m$ where x is the number of threads running different packets in a thread block, bl is the maximum number of active thread blocks / multiprocessor discussed in section 4.3, and m is the number of multiprocessors of the GPU device. In this configuration, x is 8 packets / thread block because the thread organization discussed in section 6.1.2 is $(8, 8, 4)$. And, bl obtained from CUDA profiler tools discussed in section 4.3 is 4 active thread blocks / multiprocessor. m is 30 multiprocessors for the GPU device in this setup. This yields 960 packets as the multiprocessor iteration as visible from the figure.

Third, in addition to the step function on 960 packets discussed before, one can observe a smaller step function on 240 packets as well. This smaller step function comes from the *thread block iteration*. Thread block iteration is the number of packets in a batch when all multiprocessors are partially filled with r active thread blocks discussed in section 4.3 where $r \in \mathbb{N}$ and $1 \leq r < bl$. For example, when the number of packets is 240, all multiprocessors in this setup are filled with one active thread block. As a result, r is 1 in this case. Then, adding a single packet to be executed makes one of the

multiprocessors have two active thread blocks. The rescheduling of that particular multiprocessor for the second active thread block creates the gap. This occurs the same when r is 2 and 3.

Moreover, one can see another phenomenon when the number of packets in a batch converges toward a multiple of 960 packets. The phenomenon is that the time it takes as the number of packets increases is significantly higher. As an illustration, the graph from $0 \rightarrow 240$ is almost a straight line while the graph from $720 \rightarrow 960$ curves up even though the number of packets for both of them are equal. This phenomenon comes from the memory contention between CUDA threads. As the number of packets comes close to fill up the multiprocessors, more CUDA threads is actively working. This results in higher processing time because there are limited memory channels and the threads compete to obtain the channels. Some threads win and get the memory channels while some threads lose and have to wait until the memory channels are free.

Figure 8.4 compares the throughput (Gb/s) between CPU and preinitialized GPU processing. We can observe three points from the figure. First, the throughput in CPU is constant regardless of the number of packets in the batch while the throughput in GPU is varied significantly by the batch size. The CPU throughput is constant because the CPU processes packets sequentially. Hence, the CPU throughput depends on how fast the CPU processes a single packet which is the same regardless of the number of packets to process. On the contrary, the GPU employs batch processing and the processing time is varied between different batch sizes as explained before in figure 8.3.

Second, there is a throughput drop when the number of packets exceeding multiple of 960 packets for preinitialized GPU processing. This comes from the gap of processing time from the multiprocessor iteration explained before. Because of the gap, the increased number of packets processed can not catch up with the increased processing time. This creates the throughput drop as visible from the figure. In addition, there is also a throughput drop when the number of packets exceeding multiple of 240 packets as well. This is the drop from the thread block iteration. This thread block iteration drop is less obvious than the multiprocessor iteration drop but also can be seen from the figure.

Third, the throughput gain by increasing the number of packets in the batch decreases when the batch size increases for preinitialized GPU processing. For example, the throughput gain by increasing from 960 to 1920 packets is considerably lower compared to from 0 to 960 packets even the number of packets increased is the same. The explanation for this lies in the reason

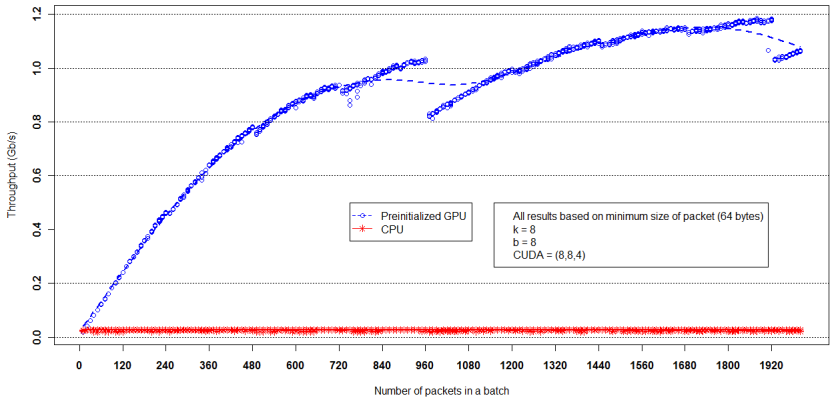


Figure 8.4: Comparison of throughput between CPU and preinitialized GPU processing

why batch processing is required. The batch processing is mandatory for GPU network processing because it compensates the data transfer startup overhead as mentioned in section 5.2. Hence, if the number of packets in the batch is already very high, the startup overhead is completely compensated and there is no throughput gain by adding more packets in the batch.

8.4 CUDA thread setup effects

As stated in section 6.1.2, there are 3 dimensions (x, y, z) of CUDA thread for layer 2 Bloom filter forwarding. This test focuses on the processing time when numbers of threads assigned to dimension x and z are different. The total number of CUDA threads is calculated by $x*y*z$ where x is the number of threads for different packets, y is the number of threads for different Bloom filters or network interfaces, and z is the number of threads for different hash functions. The total number of threads is limited according to the GPU model and CUDA kernel resource requirement. If the number is set too high, the CUDA kernel will not able to launch at all. On the other hand, if the number is set too low, the GPU resources will be underutilized. The number is set to 256 here because it is one of the best settings obtained from the CUDA optimization tools discussed in section 4.3.

Figure 8.5 shows the test result on different CUDA thread setups between (4,8,8), (8,8,4), (16,8,2), and (32,8,1). This test indicates different resources

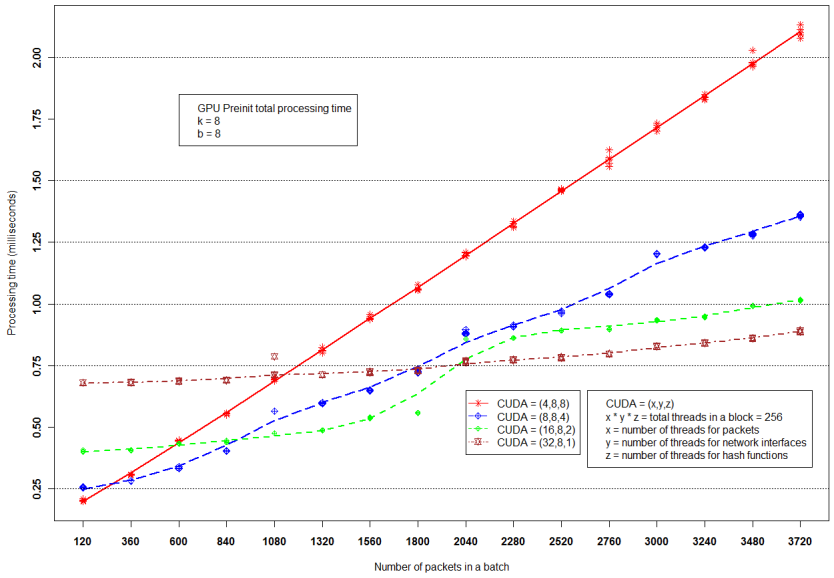


Figure 8.5: CUDA thread setup results

to be focused on between packets and hash functions in order to calculate the same workloads. One can observe that for small number of packets, the best result is obtained from focusing the resources on different hash functions. As an illustration, the (4,8,8) has the smallest processing time for 120 packets. Nonetheless, as the number of packets in the batch increases the processing time for configurations with high z and low x increases faster than configurations with high x and low z. This makes the latter have better performance in larger batch. For example, the (32,8,1) has the smallest processing time for 3720 packets.

This comes from the synchronization overhead in the z dimension. The threads in the dimension have to wait for other threads to obtain the results for different hash functions. On the contrary, the threads in x and y dimension do not have to wait for other threads because the results are interdependent. This synchronization overhead is proportional to the number of packets. Thus, when the number of packets increases, the overhead becomes visible and surpasses the benefits of focusing resources on different hash functions.

The application from this result is the guideline for configuring parameters in GPU layer 2 Bloom filter forwarding. One can know the appropriate CUDA

thread setup and size of the batch for a given application requirement. For example, if there is an application that demands the latency to be lower than or equal to 0.5 milliseconds, one can see from the figure that the best CUDA thread setup to be set is (16,8,2) and the batch size should be 1320 in order to satisfy the latency requirement and obtain the best throughput.

8.5 Summary

This chapter elaborates the empirical test results obtained from implementing the alternatives discussed in chapter 6. The tests cover various configurations including the comparison of total processing time between CPU and GPU, individual GPU-offloading processing time, preinitialized GPU-offloading implementation, and CUDA thread setup results. The next chapter concludes this thesis and provides a discussion for future works.

Chapter 9

Conclusion and future works

9.1 Conclusion

This thesis successfully reached its goal set in the beginning. The goal is to find out how GPUs should be used for offloading network processing functions. The GPUs are suitable for the offloading when the network processing functions meet particular criteria and certain tradeoffs are acceptable. The criteria are no inter-packet dependency, similar processing flows for all packets, and within-packet parallel processing opportunity. Given that the criteria are satisfied, the offloading should be done by processing multiple packets and tasks within a packet concurrently.

The empirical test results clearly show that offloading network processing to GPUs should be done via batch processing. And, the offloading is a trading of higher latency and memory consumption for higher throughput. As a consequence, the application and system must be able to tolerate certain level of latency and memory consumption in order to utilize this new approach.

In addition to the goal, other insights gained from the thesis are the performance optimization by the GPU preinitialization and predictable step function of GPU processing time relating to the batch size. The results demonstrate that initializing GPU at application boot time reduces start up overhead and improves processing performance significantly. As a result, the preinitialization is the appropriate way for GPU-offloading network processing.

The results also reveal that the processing time of GPU-offloading implementation is a step function while CPU-only implementation is a linear function in respect to the number of packets to be processed. The size of the step

can be predicted from the configuration setup. This insight implied that an optimal batch size between the trading of throughput and latency is a multiple of step size. The justification is that high throughput will be gained by sacrificing minor processing latency because the processing time will stay within the same step.

9.2 Future works

9.2.1 GPU layer 2 Bloom filter forwarding online processing

The GPU layer 2 Bloom filter forwarding prototype in this thesis is an offline processing implementation. The implementation compared processing performance between CPU-only and GPU-offloading setup with the assumption that the packets already exist in the main memory. In other words, the implementation does not involve a packet transfer between network interfaces and computing units while processing. As a result, the area for future work is to integrate the implementation with packet reception/transmission library such as *libpcap*. This will enable the studies of online processing behaviour of GPU-offloading application, for example, online processing throughput and latency.

9.2.2 GPU flow lookup in Openflow switch implementation

For the second case study of this thesis, the existing algorithm used in Openflow reference implementation is not suitable for offloading to GPUs as explained in section 7.2. This motivated the author to contribute the *GPU flow exact pattern lookup*, new algorithm for flow lookup in Openflow switch that is ideal for GPU offloading. The algorithm was shown theoretically to be better than the existing algorithm by computational complexity analysis. Nonetheless, the prototype implementation is not done because of the thesis time limitation. As a result, implementing the algorithm and comparing it with Openflow reference implementation are proposed as one of the future works.

9.2.3 Multiple GPUs Network Processing

This thesis focuses on offloading network processing functions to a single GPU. Adding more GPU devices or offloading to a GPU cluster would take the parallel execution to the next level. Nonetheless, this demands a study on how to properly coordinate between different GPU devices, for example, how to divide the workloads between several GPUs with different computing powers. Hence, this is another direction for future work.

Bibliography

[1] L. A. Abbas-Turki, S. Vialle, B. Lapeyre, and P. Mercier. High dimensional pricing of exotic european contracts on a gpu cluster, and comparison to a cpu cluster. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[3] ATI. ATI Stream Software Development Kit. `http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx`, Last Accessed February 2010.

[4] ATI. OpenCL driver from ATI. `http://www.amd.com/us/press-releases/Pages/amd-leads-industry-as-2009oct19.aspx`. Last Accessed February 2010.

[5] Bloom Burton H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[6] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.

[7] A. Cevahir, A. Nukada, and S. Matsuoka. Fast conjugate gradients with multiple gpus. In *ICCS '09: Proceedings of the 9th International Conference on Computational Science*, pages 893–903, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] Y.-J. Choi, K. B. Lee, and S. Bahk. All-ip 4g network architecture for efficient mobility and resource management. *Wireless Communications, IEEE*, 14(2):42–46, April 2007.

[9] D. L. Cook and A. D. Keromytis. *CryptoGraphics Exploiting Graphics Cards for Security*. Springer, New York, USA, 2006.

[10] M. Dashtbozorgi and M. Azgomi. A high-performance software solution for packet capture and transmission. In *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*, pages 407 –411, aug. 2009.

[11] L. Deri. ncap: wire-speed packet capture and transmission. In *End-to-End Monitoring Techniques and Services, 2005. Workshop on*, pages 47 – 55, May 2005.

[12] M. Dinesh, L. Ming C., and N. Govindaraju. Gpgpu to many-core processing: Higher performance for mass market applications, 2007. Many-core Computing Workshop, 2007.

[13] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. Gpu cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.

[14] S. Hassan, L. Baochun, and W. Xin. Nuclei: Gpu-accelerated many-core network coding. In *IEEE Infocom 2009*, Rio de Janeiro, Brazil, 2009.

[15] F. Horst. Cryptography and computer privacy. In *Scientific American*, pages 15–23, May 1973.

[16] HP. HP xw8400 Service and Technical Reference Guide. `http://h50146.www5.hp.com/lib/doc/manual/workstation/xw8400/364898-001.pdf`. Last Accessed February 2010.

[17] J. Postel and J. Reynolds. RFC 1042:A Standard for the Transmission of IP Datagrams over IEEE 802 Networks, February 1988.

[18] M. Juric. CUDA MD5 Hashing Experiments. `http://majuric.org/software/cudamd5/`. Last Accessed March 2010.

[19] Khronos Group. Opencl: The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf`. Last Accessed February 2010.

[20] Khronos Group. OpenGL: Open Graphics Library. `http://www.khronos.org/files/opengl-quick-reference-card.pdf`. Last Accessed February 2010.

[21] D. Luebke and G. Humphreys. How GPUs Work. *Computer*, 40(2):96–100, Feb. 2007.

[22] Microsoft. Microsoft Direct X. `http://www.microsoft.com/games/en-US/aboutGFW/pages/directx.aspx`. Last Accessed February 2010.

[23] Nigel Jacob and Carla Brodley. Offloading ids computation to the gpu. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pages 371–380, Dec. 2006.

[24] Nvidia. Compute Unified Device Architecture. `http://www.nvidia.com/object/IO_37226.html`, Last Accessed February 2010.

[25] Nvidia. Cuda occupancy calculator. `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`, Last Accessed April 2010.

[26] Nvidia. Cuda visual profiler version 3.0. `http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/visual_profiler_cuda/cudaprof.html`, Last Accessed April 2010.

[27] Nvidia. Nvidia compute ptx: Parallel thread execution. `http://www.nvidia.com/content/CUDA-ptx_isa_1.4.pdf`, Last Accessed March 2010.

[28] Nvidia. OpenCL driver from NVIDIA. `http://www.nvidia.com/object/io_1228825271885.html`. Last Accessed February 2010.

[29] Nvidia. NVIDIA CUDA Compute Unified Device Architecture Programming Guide, 2009. version 2.3, `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf`. Last Accessed Jan 2010.

[30] Nvidia. OpenCL Programming Guide for the CUDA Architecture, 2009. version 2.3, . Last Accessed February 2010.

[31] Openflow.org. OpenFlow Switch Specification version 1.0.0. `http://www.openflowswitch.org/documents/openflow-spec-v1.0.0.pdf`. Last Accessed February 2010.

[32] R. Rivest. RFC1321 - The MD5 Message-Digest Algorithm, 1992.

[33] Roesch M. and other Open source developers. Snort: Open Source network intrusion detection system. `http://www.snort.org/`. Last Accessed February 2010.

[34] D. Schaa and D. Kaeli. Exploring the multiple-gpu design space. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, May 2009.

[35] J. Seland. CUDA Programming. `http://heim.ifi.uio.no/~knutm/geilo2008/seland.pdf`, Last Accessed February 2010.

[36] Y. Shin, C. Yu, S. Chung, and S. Kim. End-user driven service creation for converged service of telecom and internet. In *Telecommunications, 2008. AICT '08. Fourth Advanced International Conference on*, pages 71–76, June 2008.

[37] R. Smith, C. Estan, S. Jha, and I. Siahaan. Network processor architecture for ipsec. In *Information Systems Security*, pages 158–172, 2008.

[38] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for network packet signature matching. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 175–184, April 2009.

[39] R. Suda, T. Aoki, S. Hirasawa, A. Nukada, H. Honda, and S. Matsuoka. Aspects of gpu for general purpose high performance computing. In *ASP-DAC '09: Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pages 216–223, Piscataway, NJ, USA, 2009. IEEE Press.

[40] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Recent Advances in Intrusion Detection*, pages 116–134, 2008.

[41] Wikipedia. ALU Wikipedia Picture. `http://en.wikipedia.org/wiki/File:ALU_symbol.svg`. Last Accessed February 2010.

[42] Wikipedia. CUDA Process Flow Picture. `http://en..org/wiki/File:CUDA_processing_flow_(En).PNG`. Last Accessed March 2010.

[43] Wikipedia. GPU Data Path Wikipedia Picture. `http://en.wikipedia.org/wiki/File:Northsouthbridge.svg`. Last Accessed February 2010.

[44] E. Wu and Y. Liu. Emerging technology about gpgpu. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, pages 618–622, 30 2008-Dec. 3 2008.

[45] T. Xiaobin, Q. Guihong, Z. Yong, and L. Ping. Network security situation awareness using exponential and logarithmic analysis. In *Information Assurance and Security, 2009. IAS '09. Fifth International Conference on*, volume 2, pages 149–152, Aug. 2009.

[46] M. Yu, A. Fabrikant, and J. Rexford. Buffalo: bloom filter forwarding architecture for large organizations. In *CoNEXT '09: Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 313–324, New York, NY, USA, 2009. ACM.