Offloading P4 Stages on GPU

A Research and Development Project Report
by

Huzefa Chasmai (15D170013)
15D170013

under the guidance of
Prof. Purushottam Kulkarni

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076 (India)
2018

Abstract

P4 is a domain specific programming language for programming packet forwarding planes of software network switches. Software Based Routers give the administrators the flexibility to redefine rules thus changing the priorities/blocking behaviours of certain packets when necessary. This feature can be used to improve the network performance and monitoring. Traditionally the packet processing is implemented in hardware and with good reason too, since the line rates achievable are high. Achieving such high forwarding rates comparable to hardware devices is challenging for software defined networks. Given the inherently parallel nature of packet processing, a GPU-based software router should be able to exploit this parallelism to increase the cumulative forwarding rates. P4 is target independent. So in general we can have a Higher Level P4 code for any software based router. In this report, we explore the benefits of offloading of stages of P4 compilation, which are parallelizable, to a GPU-based software router. We perform experiments taking the use case of longest prefix matching and compare the performances of a CPU based SDN and a GPU based SDN.

Acknowledgements

I wish to sincerely thank my guide Prof. Purushottam Kulkarni as well as PhD students Rinku Shah and Anshuj Garg for their guidance and support, without which this endeavor would have remained fruitless.

Huzefa Chasmai (15D170013)
IIT Bombay
10 May 2018

Table of Contents

Chapter 1

Introduction

Packets are the basic units of communication in packet-switched networks. A packet contains control information like the headers and also the user data called the payload. A network is build up of links called the routers which upon receiving a packet parse the packet headers and forward the packets therein to a corresponding network. The first part is called routing. There are two different ways of implementing the packet processing functionality in routers: either burn the logic into hardware or write it in software.

1.1 Software Routers

The hardware based approach has a huge advantage in terms of speed of forwarding. Typical speeds for a core router is around 40Gbps with fastest being around a 100Gbps[5]. Comparatively the other approach of software defined routers have the advantage of programmability. These routers also have the advantage of scalability and robustness since they can be reprogrammed with support for additional functionalities. These functions make software based networking an appealing option.

As of today most of the routers in use are hardware defined since the speed cost far outweighed the programmability benefits of their software based counterparts. Recently however the software based routers have gained attention because of two reasons :

1. With the advent of GPUs and improvement in the computational speeds, the software based routers running on such hardware have seen an increase in the line rates, PacketShader a Software Router implemented using GPUs has achieved line rates of about 40Gbps[3]

2. Complex use cases are being developed by researchers including applications like cloud computing, wherein approaches like Software Defined Networking provide efficient implementations. With SDN, the administrator can change any network switch's rules when necessary – prioritizing, de-prioritizing or even blocking specific types of packets with a very

granular level of control. Essentially, this allows the administrator to use less expensive commodity switches and have more control over network traffic flow.

Many hardware architectures are being explored for supporting such software based packet processing.

1.2 P4

P4 is a high-level open source language for programming protocol-independent packet processors. P4 provides achieves the following goals:[1]

1. Reconfigurability. The controller should be able to redefine the packet parsing and processing in the field.

2. Protocol independence. The switch should not be tied to specific packet formats. Instead, the controller should be able to specify a packet parser for extracting header fields with particular names and types and a collection of typed match + action tables that process these headers.

3. Target independence. The programmer need not know the details of the underlying switch. Instead the compiler takes into account the switch's capabilities when turning a target independent description into a target dependent one.

The last goal of target independence is what we will be exploring in this project. We take the use case of the underlying switch hardware being a GPGPU (General Purpose GPU + CPU framework).

1.3 Abstract Forwarding Model[1]

In the abstract model Figure [1.2] switches forward packets via a programmable parser followed by multiple stages of match + action, arranged in series, parallel, or a combination of both.[1]

The forwarding model is controlled by two types of operations: Configure and Populate. Configure operations program the parser, set the order of match + action stages, and specify the header fields processed by each stage. Configuration determines which protocols are supported and how the switch may process packets. Populate operations add (and remove) entries to the match + action tables that were specified during configuration. Population determines the policy applied to packets at any given time.

Arriving packets are first handled by the parser. The packet body is assumed to be buffered separately, and unavailable for matching. The parser recognizes and extracts fields from the header, and thus defines the protocols supported by the switch. The extracted header fields are

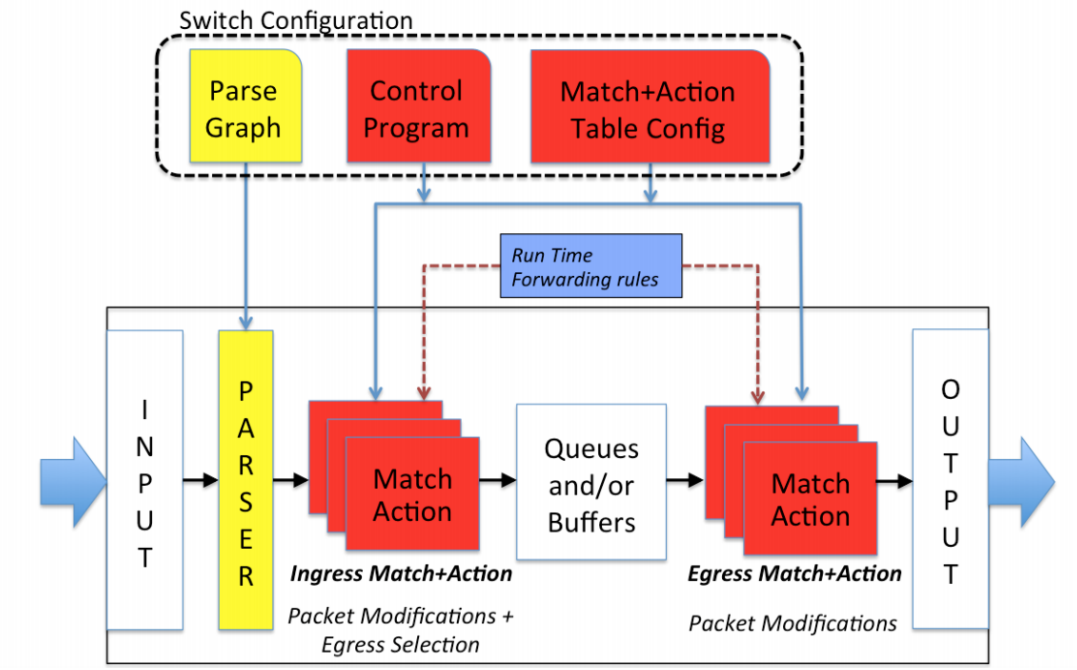

Figure 1: Abstract Forwarding Model

Figure 1.1: The Abstract Forwarding Model of P4 (Credits : p4 workshop 2015 github)

then passed to the match + action tables. According to the results of the ingress match-action tables the packet is then forwarded to the egress match action tables and eventually to the egress port with a series of header modifications.

So essentially P4 allows programmers to express packet processing logic in an imperative language representing the control flow. The control flow is implemented using match-action table pairs. These match action table pairs are then translated by the compiler to a Table Dependency Graph (TDG) which can then be analysed to check for dependencies. Finally the compiler maps the corresponding TDG on to a specific target platform.

1.4 GPGPU

General-purpose computing on graphics processing units (GPGPU) make use of a graphics processing unit (GPU) along with the CPU to handle applications. A GPU has multiple cores where each of the cores can execute instructions simultaneously. Hence GPUs are very suitable for SIMD (Single Instruction Multiple Data) applications. The dominant proprietary framework for a GPGPU computing language is Nvidia CUDA. GPGPUs typically employ an architecture where the GPU is viewed as a coprocessor to the CPU (i.e. a heterogeneous architecture)

According to Nvidia's Architecture the CPU is called the host and the GPU is called the device. We can write a C/C++ code which has instructions that run on the CPU (parsed by standard

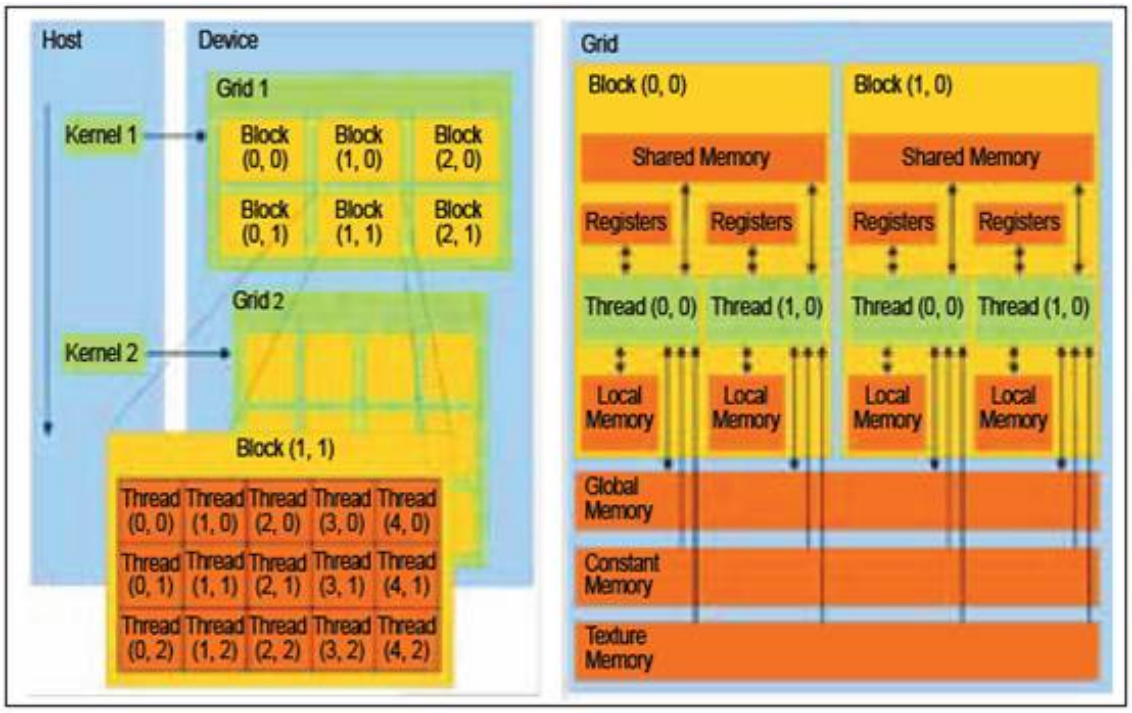

Figure 1.2: Basic CUDA Architecture [Nvidia CUDA]

compilers like g++) and at the same time create kernel functions which will run on the device (parsed using a special CUDA compiler (nvcc)). A kernel is executed as a group of thread blocks. A thread block is a batch of threads which can cooperate with each other through synchronization and block level memory sharing. Each of the threads executes the kernel function simultaneously in a block giving rise to the parallelism. Hence the GPUs are most suited for SIMD applications. There are specific function calls for copying memory between the host and the device and this is how the CPU and the GPU share data.

1.5 Report Outline

The rest of this report is organized as follows. In Chapter 2 we outline the existing methods related to our work. Chapter 3 describes the details of our approach. Our implementation setup details are presented in Chapter 4. Chapter 5 documents the results of our experiments and draws some inference. Finally, the report is concluded in Chapter 5.

Chapter 2

Related Work

There is a lot of work going on in the P4 community with respect to the different targets on which the P4 code can be burned. Currently, P4 can be programed onto a variety of targets including software switches, NPUs, and FPGAs to name a few. Also we aren't the first one's to explore GPUs as a target for the P4. The following work has already been done in this domain.

2.1 P4GPU

Li and Luo[4]
This paper has proposed a toolset called P4GPU which maps P4 defined data planes onto a heterogeneous CPU/GPU architecture. This toolset parses the P4 code and generates the corresponding match-action engine kernels. This is achieved by parsing the IR code representation and the Table dependency graphs. It also discusses the idea of a complete P4-to-GPU compiler. The paper specifies that most of the match action tables fall under two main categories and it proposes solutions to a generic kernel for the two categories : IP Lookup and Classification. For LPM the paper refers to an efficient trie-to-array vectorization technique to convert trie-based data structures into array-like format in addition to a path compression technique to reduce the memory requirements. Further on the paper also discusses GPU optimization techniques to hide the latency like multithreading. They test their implementation for a simple router use case.

2.2 PacketShader

Han *et al.*[3]
PacketShader is a software switch based on GPUs and is the first of its kind which is able to achieve throughputs in the range of 40Gbps. This is achieved by using Nvidia CUDA framework with optimizations like batch processing, concurrent copy and execution as well as optimizing the packet I/O engine. They performed further optimizations to increase the speedup by allocating memory in the NIC driver for batches of packets together rather than individual packets and also

removed unnecessary meta-data fields to achieve much higher forwarding speeds. Due to time constraints we will not be doing the same optimizations in our implementation.

2.3 Packet Processing on the GPU

Mukerjee *et al.*[5]

This paper discusses implementation challenges of implementing a software based router on the GPU. They discuss the GPU design issues with respect to pipelining and batching and their corresponding benefits. They discuss the details of a few optimization techniques in order to increase the throughput of their router. They also present results for the speedups given by the different optimizations in stages on their router implementation.

Chapter 3

Approach

In this chapter we will formulate an approach on offloading the P4 parallelizable stages on the GPU as well as lay down a basic use case to compare the speedups obtained for the basic match-action table kernel based implementations.

3.1 P4 Match Action Tables

The match action tables in the P4 represent the crux of the P4 control logic. Also packet classification and table matching are the main bottlenecks with respect to packet forwarding.[4] Most of the match+action tables fall under two types of categories :

3.1.1 LPM (Longest Prefix Matching)

Each router has its own forwarding table, where the different entries specify the different sub-networks with their corresponding masks and the ports to forward to. Now given a packet, based on its IP we need to select an entry of the forwarding table and accordingly decide the port on which it is to be forwarded. There may be multiple entries in the forwarding table matching the current IP, we need to select the one with the longest subnet mask.

3.1.2 Classification

Exact and wildcard classification are the two heuristics defined in the P4 language specification. Based on the header fields and some priority rules along with some regexps this type of table classifies the packets and accordingly modifies the header fields or routes the packets.

The P4 match action tables are concisely represented in the IR (Intermediate Representation) generated at runtime and a step towards building a parser that generates code for the GPU would be to generate kernel code corresponding to the match-action table specification mentioned in the IR. This IR is currently generated by P4 and is represented as a dictionary with various fields corresponding to the different parameters like headers to pass, type of table etc are specified.

3.2 GPU kernels for the above match action tables

In order to make a P4 compiler than reads in a P4 code and based on the match action specification generates a code that maps onto a kernel we need to build configurable kernels that take in IR representation of the match action tables and output respective kernel functions. This is possible by making generic configurable kernels corresponding to the two types of match-action tables mentioned before. In this project we take up the first examples as our use case. We show that a generic LPM kernel can be made and also show that GPUs show configurable speedup doing this processing as compared to a CPU based implementation.

3.3 Our Use case

As mentioned, we take up the use case of LPM kernel to demonstrate the speedup achieved by the GPU based implementation as opposed to the same based on a CPU. Also since this LPM kernel can be plugged in for P4 match action tables corresponding to LPM's we also demonstrate the flexibility and configurability of the GPU kernel designs. There are many ways of implementing a LPM, with Patricia tries being a very common method. But GPUs do not support applications which make use of pointers which was what the Patricia Trie Implementation had and the same using arrays wasted a lot of memory. Hence we decide to use the Linear Search kernel and correspondingly a linear search CPU code to compare the speedups. Due to time and complexity constraints we show the comparison of the speedups for the match action table implementations in the GPU based kernels and are unable to work on the part of making the a P4 code parser as mentioned above.

Chapter 4

Implementation Aspects

4.1 Experimental Platform

Device Used to do the experimenting has the following specifications:
CPU Specs :
i5 with 4 cores
6GB RAM

GPU Specs:
GeForce GTX 970
1664 Cuda cores
4GB Global Memory

4.2 DataSet

We use publicly available datasets to evaluate the system performance. We apply the largest available prefix dataset, CAIDE RouteView[2] (January 2015), consisting of approx- imately 550,000 IPv4 entries and approximately 20,000 IPv6 entries in order to evaluate the GPU lookup engines.

4.3 Problem Instance

4.3.1 Forwarding Table

We consider a forwarding table of size 50 table entries of ipv4 subnet IPs and their corresponding masks and forwarding ports set to the AS numbers as available to us based on the data set. We generate a forwarding table as such and maintain the same in the a file.

4.3.2 Query Packets IP addresses

We then generate files which store IP addresses again picked up from the dataset. We vary the size of the Query packets from 5K to 1M in representative steps and perform our analysis based on these different query data. The query data represent the IPs corresponding to the packets received by a switch at a time and we can safely assume that switches connected to busy servers receive such amounts of requests per second. We read input from the file line by line simulating receiving packets and processing them from the queue one by one.

4.4 Data Structure Used

We use the following structure to store the forwarding table entry:

```
struct table_entry
{
        unsigned long p_key;      /* Node key */
        struct mask p_m;          /* Node masks */
        int fwdport;              /* Forwarding Port */
};
```

This structure is used to store the information for each forwarding table entry mainly the p_key (corresponding to subnet IP), the mask struct object p_m and the fwdport.

```
struct mask {
        unsigned long pm_mask;
        struct MyNode *pm_data;
        int mask_len;
};
```

The mask structure has a pm_mask which is a number representing the masks so that when anded with the IP representation key it produces the same effect as logical ANDing with the Mask. Also we have our own struct called MyNode which is a dummy struct storing additional variables we might need for other match-action table entries and other additional information which may be needed to change the headers of the input packet showing flexibility of the approach.

We maintain a linear list in the form of an array of these data objects and this represents the forwarding table data structure in the code.

4.5 Different Approaches of LPM

4.5.1 Sequential CPU code

We first write a code that is based solely on the CPU and sequentially applies the linear LPM algorithm on the input IPs.

4.5.2 Parallel GPU Code

We then write a parallel using CUDA by defining a kernel function which takes in IPs in batches and runs the LPM on each IP of the particular batch. So we spawn a kernel with number of threads equal to the batch size, i.e. each IP in the batch has its own thread in the kernel which performs the same linear LPM. So the CPU keeps on collecting the data and then once a batch amount of data is collected a kernel is called whose task is to process this batch of data. Further optimization is done by means of pthreads. We spawn two threads. While one thread creates a batch the other processes the kernel function on the previous batch and vice versa. Used the pthread library for spawning of the threads and semaphores for synchronization so that both the threads do not read the same input IP. Now the batch size is kept as a parameter and we experiment with different batch sizes.

4.6 Evaluation Metrics

We measure the mean total time taken for evaluating all the queries as an evaluation metric for the algorithms. This time also includes the time taken to setup the CUDA memory and also transfer the data from the host to the device. The time taken is calculated for 10 different runs and the mean time is noted in order to avoid system dependencies on the time. The time reported are measured in milliseconds(ms). The speedup is calculated for the parallelizable code w.r.t. to the sequential code and is reported for comparison.

4.7 Libraries Used

We use CUDA and g++ as compilers for our programs. Other than that the libraries used are pthread for multithreading, netinet/in.h and arpa/inet.h for handling the IP address format conversion to a unique key and various file stream libraries for file input output operations.

Chapter 5

Results

5.1 LPM Queries vs SpeedUp

The data in the table contains the CPU time taken and also the GPU time taken for a particular query given batch size equal to number of queries. As seen from the graph 5.1 and the data in the table 5.1 the speedup goes on increasing with the number of queries processed. This shows that for higher amounts of packets the GPU based approach performs better than the Sequential CPU based approach. The time taken for the CPU sequential program is directly proportional to the queries processed. This was expected since the algorithm essentially performs a constant time operation per query along with some constant time initialization operations. The time taken by the GPU however does increase with the number of queries but not exactly linearly. This results in the speedups being not proportional. Now the reason for the increase in the time is the time for CUDA memory allocation and CUDA memory copy overheads. The total number of cores of our GPU is 1664. So once the batch size increases beyond this number all the threads cannot run simultaneously. Still the trend in the graph is followed since the internal CUDA scheduling

#LPM Queries	CPU proc time (msec)	Offloaded GPU proc time (msec)	Speedup
5000	13.405106	2.959719	4.5291819933
10000	22.79424	4.433407	5.1414724612
20000	40.81666	6.614689	6.1706090793
50000	88.28579	13.125997	6.7260254592
100000	172.2527	21.24535	8.1077835856
500000	826.9323	83.44099	9.9103845724
1000000	1645.806	153.9176	10.6927732761

Table 5.1: Comparing the CPU time, GPU and the speedup for various values of number of queries

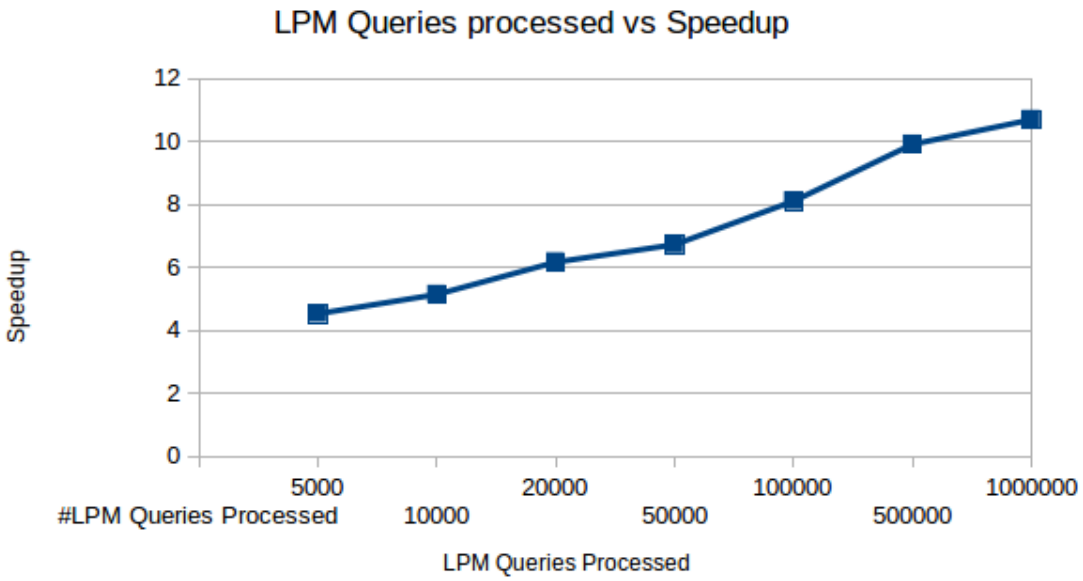

Figure 5.1: Varying the LPM Queries vs Speedup Achieved.

is highly optimized since it is hardware based thread scheduling which results in high effective speedup even in these cases.

5.2 Batch Size vs Actual Time and Speedup

For number of LPM queries = 1000000.

As we see from the figure 5.2, an increase in the batch size results in reduction in the time taken to process the queries, since queries of a single batch are processed simultaneously, so the more the size of the batch the faster the queries are resolved. But there is another factor as seen resulting from the peak obtained at batch size of 1024. For different values of the number of queries, close to the number of cores in the GPU, we see that there is a peak in the speedup / trough in the actual processing time. We suspect that this is due to the effect of the batch generation time as well as the overheads involved in the memory allocation and memory transfers to and from the kernel as well as the kernel launch overhead. For small values the batch formation time and the other overheads do not have a huge difference and the parallelising part dominates and hence the actual time goes on decreasing till 1024. At around 1024 and above, as the number of cores get exceeded these overheads increase more and probably the benefit by the parallelisation is shadowed by the increase in these overheads. A deeper analysis in the actual cause is needed.

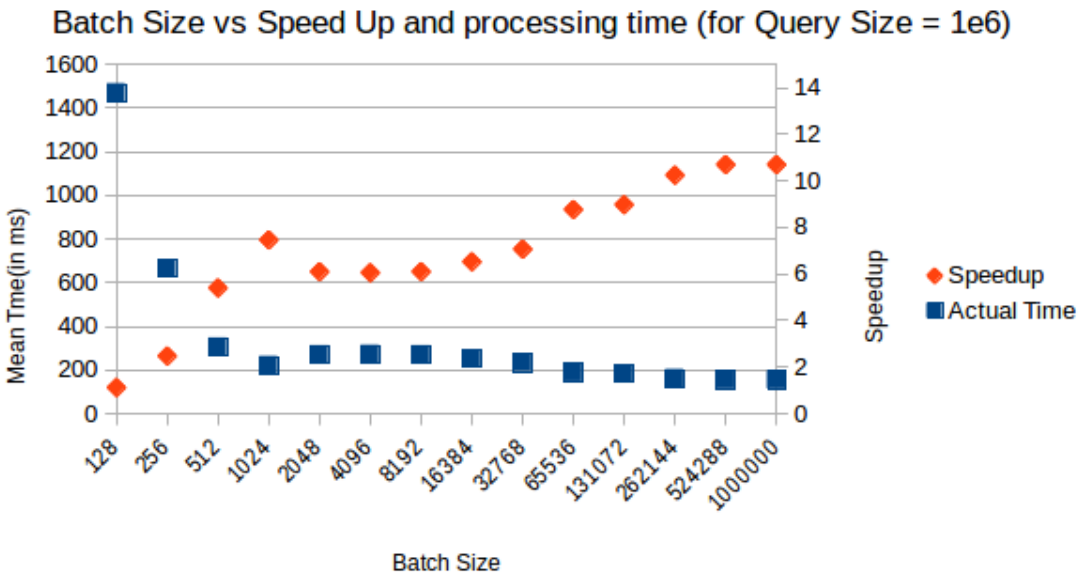

Figure 5.2: Varying the Batch Size vs Speedup and Actual Time

Batch Size	Actual Time	Speedup
128	1469.499	1.119977625
256	664.893	2.4752945211
512	304.7779	5.400017521
1024	220.5937	7.4608023711
2048	269.963	6.0964132122
4096	271.8299	6.0545436687
8192	269.8116	6.0998341065
16384	252.2364	6.5248552548
32768	232.9494	7.0650793692
65536	187.791	8.7640302251
131072	183.4072	8.9735081284
262144	160.68	10.2427557879
524288	153.9176	10.6927732761

Table 5.2: Comparing the CPU time and the speedup for various values of batch size for number of queries = 1000000

Chapter 6

Conclusion and Further Work

6.1 Further Work

Our experiments lacked a lot of GPU optimization as mentioned in[4] and[5]. Using these optimizations another set of experiments needs to be run and the speedups then compared.
Also there are many faster trie based LPM algorithms which should be used to compare the results too[4]. Apart from these we must also test the classification kernel and compare the results there. We are still far away from our goal to make a compiler that generates code for the heterogeneous CPU/GPU architecture. A lot of work can be done in this direction.

6.2 Conclusion

The use of software routers is no doubt going to increase in the future with newer applications demanding its features. The popularity of P4 with lots of papers published in recent years is a good driver for the same. As we have seen from the experiments of PacketShader and our own results of the speedup, GPU based software routers will be essential to achieve comparable line rates with the hardware routers.

6.3 Source and Detailed Experimentation Results

For viewing the source code and detailed explanation of our experiment scripts and results check out our Github page.

References

[1] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., *et al.*, 2014, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review* **44**, 87–95.

[2] CAIDE (Center for applied internet data analysis), 2010, "Routeviews prefix to as mappings dataset[online]," `http://www.caida.org/data/routing/routeviews-prefix2as.xml`

[3] Han, S., Jang, K., Park, K., and Moon, S., 2010, "Packetshader: a gpu-accelerated software router," in *ACM SIGCOMM Computer Communication Review*, Vol. 40 (ACM). pp. 195–206.

[4] Li, P., and Luo, Y., 2016, "P4gpu: Acceleration of programmable data plane using a cpu-gpu heterogeneous architecture," in *High Performance Switching and Routing (HPSR), 2016 IEEE 17th International Conference on* (IEEE). pp. 168–175.

[5] Mukerjee, M., Naylor, D., and Vavala, B., 2015, "Packet processing on the gpu," *Unpublished manuscript. Department of Computer Science, Carnegie Mellon University*