P4GPP: A GPU Accelerated P4 Packet Processing

Yongjeong Kim
Brown University

Abstract
Recent works on P4 architectures such as programmable switches and SmartNICs face challenges in meeting the resource constraint of the devices, which interrupts the P4 program design innovation for numerous applications. In this paper, we explore GPU as the next P4 architecture target. GPU contains a rich memory that can hold large register arrays and tables. Given P4’s parallelism on packet-level and program-level granularities, GPU’s massive concurrent computations units can accelerate P4 programs when compiled optimally. However, developers need GPU expertise to avoid kernel launches and branch blocking overheads.

We present P4GPP, a GPU packet processing framework with P4 abstraction. Developers can deploy P4 programs to P4GPP with ease without concerning the underlying GPU details. P4GPP compiler analyzes dependencies of P4 programs and generates a set of possible execution path graphs given the network policy. Then, it selects a graph with the least dynamic parallelism overhead and compiles the graph to optimized CUDA source codes that can execute both packets and programs concurrently with recent CUDA features. Our preliminary evaluations illustrate that P4GPP outperforms the CPU software router and both serialized and naively-implemented dynamic parallelism GPU programs.

1 Introduction
As the end of Moore’s Law suggests, the end devices’ CPUs fail to meet the ever-increasing service demands at modern data centers. In order to increase the service availability at end devices, recent efforts focus on offloading applications to either programmable network devices or accelerators. While machine learning and graphic processing applications have been offloaded to GPU at the servers, network programs are compiled to programmable data planes at the network. A programmable data plane’s pipeline is reconfigurable to add new features without vendor support and an expensive development cycle, attracting interest in offloading the number of applications to reserve server resources for other valuable computations.


However, dynamic traffic patterns perplex resource management on programmable network devices and selecting target applications to offload. Studies [4, 11] attempt to mitigate these issues by dynamically distributing offloaded program segments across the network, given each device’s resource constraint and the program’s performance requirement. There has been research on FPGA and ASIC as target accelerators for P4 programs such as P4FPGA [16] on the network path. We believe the host can take a role in distributed P4 programs context in a software-defined way by utilizing a GPU accelerator, while not significantly affecting the host’s CPU utilization.

Compiling P4 programs to CUDA C++ source codes for Nvidia GPU raises challenges due to overheads caused by the heterogeneous architecture between CPU and GPU: kernel launching delay, reordering outgoing packets, and data copying overhead from the host memory to the device memory. Recent Nvidia GPUs contain streams to hide the overhead by overlapping kernel executions and increasing GPU utilization, and also support the dynamic parallelism [1] to enable kernel launching within a device; however, due to the physical limit on the number of supported streams and launching overheads from the device, naive optimization algorithm with these features can degrade the performance.

This paper revisits GPU as a packet processing accelerator for P4 programs in a context of functional-level and data-level parallelisms; then, it seeks the best methodology to optimally translate the P4 pipeline to CUDA codes for Nvidia GPUs. It aims to realize the optimizations by applying state-of-art techniques introduced in literature and concurrent executions.
of multiple programs. It achieves functional parallelism with the dynamic parallelism and carefully crafts kernels launching and synchronizing codes during the compilation process given the kernel dependency graph, kernel weight information, and the physical limit of the underlying GPUs.

The contributions of this paper are the following:

- Analyzing previous research works on GPU network packet processing accelerators and their applicability to P4’s match-action pipeline.
- Evaluating the P4GPP framework with GPU optimizations that supports multiple stateful and stateless P4 programs.
- Proposing P4GPP run-time that can be controlled by SDN controller.

2 Motivation And Background

2.1 GPU Packet Processing Accelerator

GPUs contain hundreds to thousands of compute cores that can execute shared instructions parallel. GPUs are widely deployed in the data centers for numerous applications, including AI and graphic processing, given their components and functions with strong parallelism. A network packet processing application is suitable for the GPU target in packet-level granularity, where a series of packets can share a network functions execution path. Multi-core CPU’s core counts limit its capability in realizing the parallelism; however, GPU offers an affordable solution with its massive core count that can compute the same operation on multiple packets lightly, as illustrated in Table 1 and Table 2. While there is the launching overhead for GPUs, our evaluation in Figure 6a shows that the overhead is stable across different workloads. At the same time, the GPU booster has lower overall latency than that of the CPU booster as workloads get heavier.

GPU network packet processing accelerators on the network function virtualization (NFV) environment have been already studied in the literature for both general packet processing such as TCP/IP header updates and other network functions such as firewall and VPN and show a performance gain over CPU packet processing programs in CPU-GPU heterogeneous architecture. GASPP [15] processes stateful packet processing, including TCP operations, running purely on GPU. APUNET [5] explores the integrated GPU architecture to share the memory between CPU and GPU to avoid memory-copying. G-NET [18] supports the dynamic SM distribution among network functions and batch sizing based on network function performance.

Nonetheless, there is no extensive study on GPU as a P4 compilation target in the context of the latest CUDA features, such as the dynamic parallelism that allows launching kernels from the device for the concurrent programs execution.

Figure 1: V1Model Architecture

P4GPU [9] compiles one P4 program into its heterogeneous CPU-GPU architecture and applies its load-balancer component to distribute workloads to either CPU and GPU. However, it does not provide an insight into multiple P4 programs environment and the dynamic parallelism feature; its distributed workload on the CPU increases the CPU utilization that can be reserved for other services at the server.

2.2 P4 Abstraction and Pipeline

CUDA C++ is a high-level programming framework with a rich set of APIs that one can utilize to implement network functions manually; however, it requires GPU expertise to optimize the implementation. The general-purpose language complicates optimizing the domain-specific program and may increase latency when there are heavy computations or incorrect use of features. P4 is designed as a lightweight programming language for programmable network devices, which can concisely describe network functionality with restrictions on operations. P4’s V1Model pipeline is composed of parser, ingress, egress, and deparser stages, where parser and deparser are used for assembling and disassembling packet headers, and ingress and egress are used for match-action stages for updating packet values or states, as illustrated in Figure 1.

2.2.1 Parser and Deparser

In order to ensure low latency, the scope of packets for ingress and egress stages is limited to headers in P4. The data isolation process is done during the parser and deparser stages: the packet headers are decoupled from the payload during the parser stage with conditional states and assembled back during the deparser stage. Some of the parsed headers can be omitted. P4 headers are composed of customized headers, including metadata and standard metadata. The standard metadata structure is used for statistical purposes and accessing or updating information regarding ingress and egress ports. As illustrated, the limited data scope on the packet makes it suitable for a GPU accelerator where accessing packets from the device requires data-copying.
<table>
<thead>
<tr>
<th>Product</th>
<th>Number of Cores</th>
<th>Memory Size</th>
<th>Memory Bandwidth</th>
<th>Architecture</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>V100S PCIe</td>
<td>5760</td>
<td>32 GB</td>
<td>1134 GB/s</td>
<td>Volta</td>
<td>$10,669</td>
</tr>
<tr>
<td>Quadro RTX 6000</td>
<td>5256</td>
<td>24 GB</td>
<td>672 GB/s</td>
<td>Turing</td>
<td>$4,000</td>
</tr>
<tr>
<td>Quadro RTX 8000</td>
<td>5256</td>
<td>48 GB</td>
<td>672 GB/s</td>
<td>Turing</td>
<td>$5,500</td>
</tr>
<tr>
<td>T4</td>
<td>2880</td>
<td>16 GB</td>
<td>320 GB/s</td>
<td>Turing</td>
<td>$3,904</td>
</tr>
</tbody>
</table>

Table 1: High Performance GPU Products

<table>
<thead>
<tr>
<th>Product</th>
<th>Core Count</th>
<th>Max Turbo Frequency</th>
<th>Cache Size</th>
<th>Memory Bandwidth</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon Platinum 8180</td>
<td>28</td>
<td>3.80 GHz</td>
<td>38.5 MB</td>
<td>2666 MHz</td>
<td>$8,950</td>
</tr>
<tr>
<td>Intel Xeon Platinum 8170</td>
<td>26</td>
<td>3.70 GHz</td>
<td>35.75 MB</td>
<td>2666 MHz</td>
<td>$9,349</td>
</tr>
<tr>
<td>Intel Xeon Platinum 8160</td>
<td>24</td>
<td>3.70 GHz</td>
<td>33 MB</td>
<td>2666 MHz</td>
<td>$4,825</td>
</tr>
</tbody>
</table>

Table 2: High Performance CPU Products

2.2.2 Ingress and Egress Pipeline

After parsing the packet headers, the following pipeline is composed of ingress and egress processes, which are a series of actions and table look-ups. The action may update the header or register values. P4 tables contain keys for each entry and its corresponding run-time action, which may execute one or more other actions. Some of the standard external action, such as dropping, allows marking a packet dropped in its metadata to avoid further execution. The table look-up process does not create a race condition, which can be executed concurrently in the context of GPU. The egress port information in the standard metadata is updated during ingress or egress to ensure routing at the end of the P4 pipeline.

2.3 GPU Network Processing Challenges

2.3.1 Kernel Launching Delay

To assign jobs to the device (GPU), the host (CPU) needs to launch a kernel with a batch of threads. These threads share the same execution unit. The batch data should be copied from the host memory to the device memory along with the kernel launching. When the device finishes its execution, the data is copied back from the device memory to the host memory.

There are multiple techniques to hide this memory copying overhead, where some of them are hardware-dependent: 1) HyperQ [3] with the multiple streams (buffers) to overlap the copying 2) zero-copying approach to copy data on demand 3) the unified memory [6] 4) integrated GPU and 5) GPU Direct RDMA [14]. In the integrated GPU, the host and the device share the memory region, so memory copying is not required upon kernel launching. In the direct access from GPU to NIC, the device can bypass the host for the packet data access. However, these techniques are hardware-dependent and not a universal solution for GPU accelerators. HyperQ is a technique to hide the data transfer overhead by using multiple streams to launch kernels, overlapping the memory copying. In the zero-copying approach, the data at the host memory is copied to the device memory on demand. It is beneficial if a small amount of the data is needed spontaneously. The unified memory allows a unified virtual address space between the host and the device and transfers data on demand. However, it does not improve the overall performance regarding the data transfer delay for the GPU packet accelerator.

2.3.2 GPU Utilization

There are two main types of built-in support for CUDA-capable Nvidia GPU for efficient utilization: 1) HyperQ and 2) MPS [10]. With multiple streams, HyperQ allows executing multiple kernel launches from the host concurrently within a context. MPS extends HyperQ by allowing kernel launches in different contexts to run concurrently. The drawback of these supports is that they are hardware-dependent solutions.

Batching packets is another way to optimize GPU utilization. Launching a kernel is an expensive operation that leads to data copying latency and a kernel setup. Batching for each kernel launch is required to efficiently hide these latencies to assign jobs to multiple threads, increasing GPU workloads and overlapping launches. However, choosing the right batch size for each application is challenging since the GPU computation time may differ. Based on the evaluation from this paper, increasing the batch size reduces the overall latency, but eventually, the gain is negligible due to batching packets overhead.

The dynamic parallelism allows executing multiple kernels in a single launch from the host side. Using different streams for each dynamic kernel call within the main kernel function enables concurrent executions of the kernels in GPU. As noted from NFP [12], stateless network functions can be executed in parallel, as well as some stateful network functions with minor synchronization processes. Nonetheless, our evaluation illustrates that light child kernel launches result in a minor performance gain or loss due to additional launching overhead within GPU itself and the blocking on the caller for synchronizing. The overheads [13] lead to even more latency,
including the hardware limit on the number of streams, light child kernels that do not hide child kernel launching overhead, and the heavy child kernels that cause the parent kernel idle.

2.3.3 Preserving Packet Order

The order of outgoing packets is preserved within a batch of packets that are launched together since the implicit synchronization is done when the parent kernel returns; however, the order of outgoing batches is not guaranteed even if the same kernel is launched since they do not share the execution unit and branch blocking may occur differently among batches.

While sending out-of-order batches to the network may increase throughput, it may result in incorrect network behavior. There should be an explicit ordering mechanism within a host to balance efficiency between GPU utilization and outgoing packet ordering. As observed from BLOP [19], kernels have minor execution time differences for network functions, and these can be reordered with a delicate operation such as batch id.

2.3.4 Data-Level Parallelism

Stateful packet processing is not preferred in network packet processing since it may increase latency when serial access on a shared object is required. In GPU, a batch of threads in a warp share the same execution units, so concurrent data access on a shared object may raise a race condition, and serial data access on the object can block other threads.

While the stateful processing may cause serial executions on GPU, stateless functions illustrate the data-level parallelism on network packets. In the context of P4, packets’ headers are processed through match-action pipelines with one or more table lookups whose pipeline can be executed as a batch of packets in parallel and overlapped with each other with streaming of GPU. Along with the function-level parallelism, applying the data-level parallelism may significantly reduce the latency with GPU.

2.3.5 Function-Level Parallelism

While previous works use GPU to execute network functions on a batch of packets to hide the overheads, our work uses the dynamic parallelism feature to enable functional parallelism. NFP (Network Function Parallelism) realizes the functional parallelism on the chained network functions in a virtualization environment with a multi-core CPU and DPDK. It executes network functions serially or parallels with minor copying overhead if the functions can be merged later based on a priority chart. Nonetheless, NFP is not concerned with global and per-packet states; network functions do not share metadata but forward a processed packet.

The simple modification of the algorithm with a shared global object cannot be directly applied to GPU accelerators due to its heterogeneous CPU-GPU fabric requirement, where its unique challenges, including launching delay and branch blocking, exist. The naive implementation of NFP’s algorithm with the dynamic parallelism results in the additional overheads [13] on GPU due to hardware limitations on the number of streams.

3 P4GPP Framework Workflow

This section describes the workflow for P4GPP framework packet processing and table management. P4GPP Framework has the main program and run-time component as illustrated in Figure 2. These components are generated during the compilation process that consists of three stages. P4 compiler compiles the merged P4 program that consists of multiple P4 programs as control blocks. Given the JSON representation of the P4 pipeline and network configurations, the P4GPP compiler generates appropriate CUDA source codes for the main framework and P4GPP run-time. Then, the programs compiled by the CUDA compiler are deployed to the host CPU and interact with its NIC and GPU for the packet processing. Given the network configuration and P4info file, P4GPP run-time interacts with the network controller for SDN control and inserts tables accordingly to the main program.

With the router abstraction of the server, we enable fine-grained traffic control with the network controller. The SDN controller manages tables across switches and P4GPP with network applications. The controller interacts with the runtime component through the P4GPP driver. Given the local communication channel with the main framework, the runtime installs table entries to the framework, manages bandwidths, and shares the statistical data with the SDN controller. The abstraction provides a few advantages over standalone local controllers within the framework: 1) centralized control with SDN controller and switches, 2) compatibility with existing tools, and easy deployment. In the future, we plan to optimize the communication channel between P4GPP runtime and the network controller.
4 Code Translation

P4 syntax resembles C/C++ syntax such as data types, basic operators, and if-blocks. Header definitions contain bit or int type members, and actions (P4 functions) take registers, headers, or run-time parameters to compute operations, including assignment, addition, and subtraction. Nonetheless, there is a unique challenge to translating the P4 pipeline to CUDA codes to realize GPU optimizations such as branch blocking, memory copying, and serial execution overheads. In this section, we describe how to compile the P4 program into the P4GPP framework efficiently with ddosd [8] P4 program as an example.

4.1 Headers

Following is the P4 headers of ddosd.p4:

```p4
typedef struct header_t {
    uint8_t *ethernet;
    uint8_t *ddosd;
    uint8_t *ipv4;
    uint8_t *metadata;
    uint8_t *standard_metadata;
    uint8_t ethernet_valid;
    uint8_t ddosd_valid;
    uint8_t ipv4_valid;
} header_t;

header_t hdrs[QUEUE_SIZE];
```

CUDA framework contains a rich set of APIs to support the zero copying: `cudaHostAlloc` allows allocating memory to store the original packet header that is accessible by both the host and the device, and `cudaHostGetDevicePointer` returns a device pointer that the device can use to access the data.

To support header members that is not evenly divisible by 8, we utilize bit-level operators to fulfill the alignment requirement. For instance, to modify IPv4 ECN value to 3, the translator will first compute the byte offset of ECN from IPv4 header (one byte) then apply & and | operators to modify dedicated bits.

```p4
uint16_t ipv4[1] = (hdrs[index].ipv4[1] & 0xfc) | 0x3;
```

Such operations come with the cost, so that, in order to fully optimize the header access, members of the type that is evenly divisible by eight are always preferred.

P4 supports the valid function for each header to indicate the header validity, which needs to be implemented as an extern function. To translate the P4 header validation, each header carries a validity byte in each structure, where the value of 1 indicates the header is valid. The P4 main header carries a metadata structure to record debugging information and transfer packet-specific information among stages. To avoid the massive memory copying with the zero-copying approach, these members are stored as pointers to specific data regions of packets, and GPU copies the memory segment on demand.

4.2 Registers

In P4, registers are crucial to implementing stateful network functions. The register can be declared as an array of bit or int-type entries. The following shows the example of register declaration with ten entries whose type size is 7 bits in P4:

```p4
register <bit<7>>(10) sample_register;
```

Unlike switch fabrics, GPU contains GBs of device memory with hundreds of GB/s memory bandwidth as noted from table1. Taking advantage of this rich memory and optimizing data accesses, P4GPP expects a register of data types evenly divisible by a byte; else, it programatically converts the bit-type variable type to byte type variable.

In order to allow global access to registers across all threads in different kernels, registers need to be stored in the global memory. For this reason, we store registers in the device memory and access them by threads in GPU kernels. While storing and sorting packets at the host side is possible as that of GPUNFV [17], such a mechanism leads to a sorting burden on the host side. It requires register transfer between the host
and the device memory and the synchronization cost to merge updates on the register at the host side. The cost of such an operation gets expensive as the size of the register increases. Following is the register declaration in CUDA for the example above:

```c
__device__ uint8_t sample_register[10];
```

CUDA supports atomic operations on the global device memory for applications that require strong consistency on the register. However, writing operations on the shared global memory for every possible kernel launching is not preferred since such implementation may lead to constant blocking. While such phenomena are universal across all accelerators, the GPU accelerator will suffer from the serial execution due to threads blocking, failing to hide its other overheads with its paralleled execution units. We are planning on exploring a flow-sorting approach with dedicated cloned register memory to mitigate the overhead in the future.

### 4.3 Parser and Deparser

P4 parser is responsible for parsing the incoming packet into its headers and payload. The parser is composed of one or more states, where each state is responsible for parsing a specific header. It determines the next state after extracting the header and matching the header type. There are possible actions after matching the type: 1) transition to its next step, 2) drop the packet, and 3) accept the packet. Following is the parser from the ddos P4 program:

```c
parser Parser (...)
{
    state start {
        pkt.extract (hdr.ethernet);
        transition select (hdr.ethernet.ether_type) {
            ETHERTYPE_DDOSD: parse_ddosd;
            ETHERTYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }
    state parse_ddosd {
        pkt.extract (hdr.ddosd);
        transition select (hdr.ddosd.ether_type) {
            ETHERTYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }
    state parse_ipv4 {
        pkt.extract (hdr.ipv4);
        transition accept;
    }
}
```

The parser is translated into a simple, lightweight C++ function that executes on the CPU. Each state is implemented as a separate parser function and calls another state for the next transition or returns an error for packets to be dropped. We use the zero-copying approach by using pointers instead of copying the header contents. After extracting the header, its pointer is passed to other components for ingress and egress processing, while its payload is stored in the host memory to avoid additional data transfer between the host and the device memories.

P4 deparser is responsible for assembling the packet back with its payload and header. If there is no change in the header order, it does not require an additional operation. However, if the order of headers changes due to the deparsing operation, packet copying is required to reassemble accordingly. The deparser can be translated to bypass the process if the copying is not required or to modify the outgoing packet header if reordering is needed.

### 4.4 Match-Actions Tables

P4 tables contain one or more keys for each entry and its corresponding run-time action. When there is no matching, a default action is taken if defined. In CUDA, the table can be represented as an array of entries. Each entry contains one or more key members, an action number indicating which action to take, and action parameter values. These arrays are stored in the device memory to be accessed globally across streaming multiprocessors. The fine-grained dynamic parallelism can be applied to each run-time action; however, light actions may fail to hide the launching overhead. The hardware limit on the number of streams does not realize this optimization as the batch size increases.

While the tables can be stored in a cached memory with a constant specifier, our evaluation shows that using the constant memory for the table entries for caching does not provide many benefits for network packet processing. Due to the limit on the number of bytes for each cache for each streaming multiprocessor, the table entries do not fit well to the cache memory, as shown in Table 3. For the network functions with packets that access specific similar entries, the device memory entries achieve a similar performance as that of the constant memory table entries. We plan to evaluate fine-grained dynamic parallelism on hash algorithms best optimized for network tables in the future.

### 4.5 P4 Programs

We assume that the user provides a merged P4 program to the compiler, where each P4 program is encoded as a separate control block as illustrated below:

```c
control ddosd (...) [⋯]
c ontrol firewall (...) [⋯]
c ontrol load_balancer (...) [⋯]
c ontrol ipv4_routing (...) [⋯]
c ontrol ingress {
    table ddosd_syn [⋯]
}
```

<table>
<thead>
<tr>
<th>P4 programs</th>
<th>Number of Tables</th>
<th>Number of Entries</th>
<th>Register Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firewall</td>
<td>2</td>
<td>2048</td>
<td>512 B</td>
</tr>
<tr>
<td>ddosd</td>
<td>3</td>
<td>1024</td>
<td>62503 B</td>
</tr>
<tr>
<td>IPv4 Routing</td>
<td>1</td>
<td>3072</td>
<td>0 B</td>
</tr>
</tbody>
</table>

Table 3: Modified P4 Programs for GPU Target
After processing all kernels, the parent kernel synchronizes, which is called by the host.

Given the control blocks, P4GPP compiler applies the algorithm to generate a parallel execution path. We will discuss the algorithm in details in section 6.1. Below is the example of CUDA kernel codes generated by the compiler:

```c
#include <cuda_runtime.h>

// Table to store the actions of the kernel. The default action is NoAction.
#define table mv_32 sync {
  actions = [NoAction;] (NoAction;)
#define table mv_32 sync {
  actions = [NoAction;] (NoAction;)
#define table firewall sync {
  actions = [NoAction;] (NoAction;)
#define table load_balancer sync {
  actions = [NoAction;] (NoAction;)

// Function to apply the actions of the kernel.
void apply {
  dddos.apply(...) ;
  dddos_sync.apply();
  mv_32.apply(...);
  mv_32_sync.apply();
  firewall.apply(...);
  firewall_sync.apply();
  load_balancer.apply(...);
  load_balancer_sync.apply();
}
```

Given the intermediate representation, the compiler translates serialized functions to device kernels and functions to be executed dynamically into global kernels. To avoid redundant kernel launching, only a single thread launches the global kernel using a different stream for the concurrent execution. After processing all kernels, the parent kernel synchronizes the result. `g_network_functions` embodies this kernel handling, which is called by the host.

## 5 Dynamic Parallelism Optimizations

### Algorithm 1 P4GPP Execution Path Algorithm

**Input:**
- `branch_count(sync_{i}, sync_{j})`: it calculates the number of branches between two synchronization points.
- `next_path(service_graph, sync)`: it returns a path at the given sync point from the service graph with the shortest execution time.
- `max(P_{0}, ..., P_{nbranch})`: it returns a branch with the longest serial execution time compared with other individual branches if they are launched in parallel.
- `service_graph`: the service graph by NFP algorithm.
- `nStreams`: the number of streams reserved for each parent kernel.

**Output:**
- `execution_path_graph`: the paralleled P4 programs execution path graph.

```c
1: for each sync do
2:   nbranch ← branch_count(sync prev, sync current)
3:   P_{sub parent} ← max(P_{0}, ..., P_{nbranch})
4:   for each p ∈ P, where p ≠ P_{sub parent} do
5:     if sync ≥ t sync then
6:       merge(P_{sub parent}, P current)
7:       nbranch ← nbranch − 1
8:     end if
9:   end for
10: while nbranch > nStreams do
11:   P current ← next_path(service_graph, sync)
12:   Merge(P_{sub parent}, P current)
13:   nbranch ← nbranch − 1
14: end while
15: if t exec sub parent ≤ launching_overhead then
16:   for each p ∈ P, where p ≠ P_{sub parent} do
17:     Merge(P_{sub parent}, P current)
18:     nbranch ← nbranch − 1
19: end for
20: end if
21: P parent ← P parent + P_{sub parent}
22: end for
```

### 5.1 NF Parallelism Identification Algorithm

Our kernels graph algorithm is based on NFP’s NFPI (NF Parallelism Identification) algorithm for creating parallel service graphs. NFPI checks dependencies among network functions to create an execution path as illustrated in Figure 3. Excluding programs that need to be executed serially due to write-read relationship, NFP executes other programs in parallel with or without packet cloning. For instance, functions with write-write or read-write relationships can be executed.
in parallel with cloning packets and then merge the result.

In order to enable this feature at GPU, the cloned packets need to be stored in the global memory. However, the synchronization comes with a cost for the GPU accelerator. Due to the dynamic parallelism overheads as noted earlier [13], directly applying NFP’s NFPI algorithm to GPU may lead to worse performance: 1) child kernel launches with different streams may exceed the hardware limit on the number of streams, resulting in launch waiting time, 2) light child kernel executions (parallel version) fail to hide the launch overhead, and 3) writing and reading data from global memory causes additional latency. Even if the heavy parent execution time hides the light child kernel overhead, it consumes the number of idle streams for other works such as parent kernel launching.

CUDA-enabled GPUs in the data center contain 32 to 64 hardware streams, as noted in Table (need to create the table). Algorithm 1 limits the maximum number of paralleled branches to avoid launch blocking due to streams. Based on the dependency graph generated from the NFPI algorithm, the P4GPP algorithm weights each paralleled execution path, where weight may include execution time, synchronization cost, and launching overhead. Then, it selects a branch with the heaviest execution as a parent kernel execution path to hide the child launching overhead and serialize the child kernel branch if its synchronization cost is higher or equal to its execution time. The algorithm merges light branches to meet the maximum number of paths (branches) based on the number of streams supported by the underlying GPU. After the processing, if the resulting parent’s execution time is shorter or equal to the child’s launching time, it merges the light child branch to increase the parent execution time. It repeats this process for every sync group.

Figure 3 demonstrates the process for generating the execution path graph, where the maximum number of paralleled function execution is set to 3 streams. Given serially chained P4 programs, NFPI generates the network function dependency graph that parallels some of the executions of the program with synchronization copying when needed. The dotted branch represents a child kernel launching; the red branch represents paralleled P4 programs but synchronization is needed; the blue branch implies paralleled P4 program launch without a need for copying synchronization.

The execution path generator selects the most compute-intensive branch as the parent execution path and adds weights to each branch. Assuming that the sum of launching overhead, f4 execution time, and f4 synchronization overhead exceeds the sum of f1 and f2 execution times, it merges the f4 branch since the launch of the kernel results in performance loss. As a result, the algorithm generates an execution path graph where it consumes one less stream and takes less computation time than that of NFPI for the GPU accelerator.

However, multiple variations of the execution path graphs can pass the optimization and the execution order policies. A particular path may perform better than the others in the target deployment environment, which may contain different running services in the hardware. P4GPP optimizer repeats this execution path generation process to create a variation of the optimized paths, which can be tested and selected during the deployment. We leave the automated testing process as future work.

6 P4GPP Framework

The framework is designed to accelerate packet processing with GPU and P4 as its abstraction layer. As illustrated in Figure 4, the CUDA code is generated by analyzing the P4 JSON file from the P4 compiler. P4GPP compiler analyzes the JSON representation of P4 programs to generate the service.
graph by applying the NFPI algorithm. Then, the execution path generator takes the graph as an input to generate the execution path intermediate representation. The P4-CUDA translator generates the overall framework code, uses this intermediate form to generate kernel codes and takes p4info files from the P4 compiler to generate the tables management component. Finally, the CUDA compiler compiles the CUDA source code to produce the executable file.

6.1 P4GPP Execution Path Algorithm

Figure 5 shows the framework design for P4GPP. There are four worker threads at the host side for reading I/O, writing I/O, batching, and runtime supports. The read I/O thread is responsible for reading and parsing incoming packets from the network interface card to their headers and payload and placing the packets in the packet pool queue and the parsed headers to the incoming queue. Then the batcher thread reads packets from the incoming queue, creates a batch of packets with a batch id tag, and launches the batch to the device. It utilizes multiple streams to concurrently launch each batch and monitors each stream to synchronize the process. After the kernel execution is completed, the thread inserts the batch to the appropriate outgoing queue position according to their IDs to preserve the outgoing packet order as that of BLOP [19]. The write I/O thread then deparses the packets in a batch and sends them to their designated egress port number. The runtime thread handles runtime requests such as statistics and table entry queries, and table management.

6.2 Memory Copying Optimizations

P4GPP avoids memory copying overheads in its framework at the host side. Incoming queue and outgoing queue arrays are pointers to the packet buffer pool, and batches are an array of the header pointers to the packet. The packet buffer memory is allocated during the framework launching with CUDA API, and it passes device pointers to this memory in a batch to the device. This allows the zero-copying approach to leverage the memory copying overhead between the host and the device memories since most network functions do not fully access the header members.

6.3 GPU Utilization Optimizations

P4GPP enables Hype-Q [3] with streams to hide the parent kernel launching by overlapping the launching at the host side. It also uses dynamic parallelism [1] at the device side by launching child kernels from the parent kernel based on the kernel execution path graph generated by the execution path generator algorithm. In order to mitigate additional blocking or waiting overheads due to excessive use of streams, it sets a threshold on the maximum number of streams present in the framework.

7 Evaluation

The framework is tested with Intel(R) Core i7-7700HQ CPU, NVIDIA GeForce GTX 1050, and an external Samsung SSD T7 SCSI Disk Device. The GPU device has Pascal architecture with 640 cores. The framework uses history-based knowledge to weight P4 programs for creating the execution
path graph, which is inserted by the user at the compile time. Our framework preserves 100 percent outgoing packet order while outperforming our implementation of CPU software router. For the preliminary evaluation of the GPU packet processing accelerator, we compile firewall, IPv4 forwarding, and load-balancer P4 programs into our framework and run one million packets to test the performance. The policy requires firewall, IPv4 forwarding, and load-balancing to be executed in that order. To emulate the heavy workloads and P4GPP’s execution path selection algorithm, we generate 11 toy P4 programs where we tune the implementation to manipulate the synchronization and execution times. The first version executes P4 programs serially, the second version applies the NFP algorithm, and the third version generates a framework that utilizes the P4GPP algorithm with stream limit.

Figure 6 demonstrates our preliminary evaluation of CPU, fully-serialized GPU, and NFP-based GPU implementations. In this test case, we use the small table size to observe the overhead implication. All GPU implementations outperform the CPU implementation in terms of the overall latency. The table lookup process by multiple packets can be improved significantly with the parallelism. However, the serialized ordering had lower latency than that of the NFP with the dynamic parallelism as the batch size increases as shown in Figure 6a. This situation is self-explanatory since there are more packets to synchronize while the total table lookup time is relatively faster as demonstrated from the overall latency for the small batch size (64) in Figure 6b. It was evident that NFP fails to detect the most optimized algorithm based on the launching and synchronization costs.

After the preliminary evaluation, we test the framework with toy P4 programs to evaluate whether P4GPP chooses the most optimized path during the compilation to demonstrate the proof of concept. The framework is tested with 11 light P4 programs with the write-write relationships. NFP executes these programs parallel with cloning (device memory) while P4GPP serializes the execution due to launching overhead and light child and parent execution time. As Figure 7a demonstrates, NFP performs worse than serial executions of the programs while P4GPP chooses an optimized solution along with the serial execution. Then, we inserted policy rules among these programs so that some of these programs can be executed serially in NFP. Nonetheless, NFP executes programs in parallel as many as possible, which results in worse performance for light programs as in Figure 7e. Figure 7d illustrates the one kernel execution timeline for the programs in NFP: most of the parent’s execution time was spent on setting and synchronizing cloned packets and launching child kernels.

Then, we tested 11 heavy toy programs that frequently read and write to a packet with the write-write relationship. As expected, NFP executes all of the programs in parallel with cloning. In Figure 7b, we were able to observe that NFP performs significantly better than serial executions due to heavy parent and child workload and P4GPP followed NFP’s execution path because launching and synchronization costs are effectively hidden. NFP and P4GPP share the same execution paths in a dependency-simulated environment with those programs and outperform the serial execution. In emulated hybrid workload with mostly light and heavy programs, P4GPP serialized light programs while NFP executed all programs in parallel. In Figure 7f, all of the accelerators achieve near-optimized performance because NFP hides its overhead on light programs by executing the heavy program in parallel.

8 Future Works

Some of the target implementations are in process and not included in the evaluation. One exciting aspect we are currently exploring is implementing the SDN controller application, such as those of ONOS, to dynamically run our compiler and
schedule particular P4 programs to the server. The current prototype takes the runtime terminal access to the framework interactions such as policy modifications and the table entries management. Another limitation in the current implementation is missing the http server component. Currently, it only works with the command line input interaction. In the future, we are exploring the http connection for rich set of API-based controls and SDN controller compatibility for the dynamic deployment and management.

While the current evaluation shows the proof-of-concept, it requires real-world use cases for the robust framework testing, where P4 programs are widely used for the basic network functions these days. A lot of related P4 works depends customized externs not written in P4 language. This complicates the deployment and requires compiler add-on functions for co-operations. We are looking at the direction of deploying the NFV P4 functions to our framework with the extern to GPU code translator and testing them in the server rack. While some of the configurations are ready, we leave finding the target functions and deploying them for better real-world related workloads for the evaluation as future work.

Current testing environment uses the external GPU accelerator connected to CPU through PCIe bus. Recent innovations introduce high-performance integrated GPUs with diverse resource constraints, those with RDMA feature, and DPU. It will be insightful to explore the different architecture and integrate the implications to the compiler based on the target hardware.

Lastly, the current implementation searches the match-action table entries linearly. While there have been some discussions of the GPU table lookup in parallel, there are no-built in support for the feature from CUDA. Furthermore, the table key tuple size for P4 is dynamic and reconfigurable so that it requires further investigation to realize the feature.

References


