NVIDIA DOCA GPU Packet Processing Application Guide

Application Guide

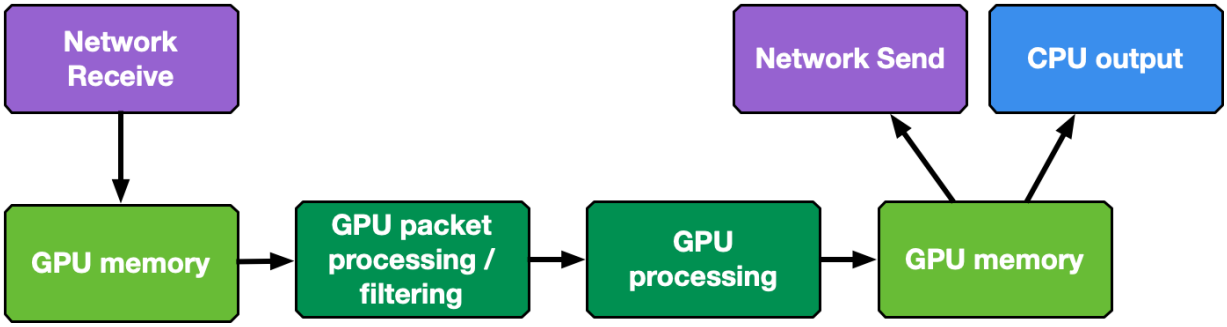
MLNX-15-060588 _v2.0.2 | May 2023

Table of Contents

Chapter 1. Introdution

Note: This application is currently supported at beta level.

Real-time GPU processing of network packets is a technique useful to several different application domains, including signal processing, network security, information gathering, and input reconstruction. The goal of these applications is to realize an inline packet processing pipeline to receive packets in GPU memory (without staging copies through CPU memory), process them in parallel with one or more CUDA kernels, and then run inference, evaluate, or send the result of the calculation over the network.

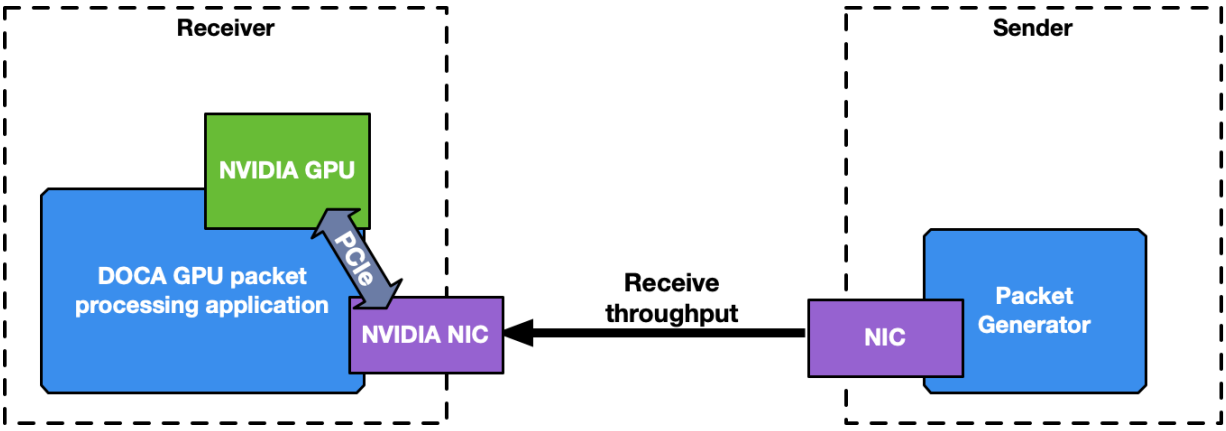


The type of data processing heavily depends on the use case. The goal of this application is to provide a basic layout to reuse in the most common use cases of being able to receive, differentiate and manage the following types of network traffic in multiple queues: UDP, TCP and ICMP.

This application is an enhancement of the use cases presented in the NVIDIA Blog post about DOCA GPUNetIO.

Chapter 2. System Design

This is a receive-and-process DOCA application, so a packet generator sending packets is required to test the application.



To launch the application, the PCIe address of the GPU and NIC are needed.

Before running the application, ensure that configurations explained in NVIDIA DOCA GPUNetIO Programming Guide and NVIDIA DOCA Flow Programming Guide are respected. For example, make sure to configure huge pages of 1GB (DOCA Flow and DPDK requirement):

```
$ sudo vim /etc/default/grub

# Add hugepages size and number along with other options you have in the CMDLINE
# GRUB_CMDLINE_LINUX_DEFAULT="default_hugepagesz=1G hugepagesz=1G hugepages=4"
$ sudo update-grub
$ sudo reboot

# After rebooting, check huge pages info
$ grep -i huge /proc/meminfo
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
HugePages_Total:      16
HugePages_Free:       15
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:    1048576 kB
Hugetlb:        16777216 kB
```

List of items to check before running the application:

- CUDA Toolkit is installed at 12.1 version or newer
- `doca-gpu` and `doca-gpu_dev` packages are installed
- `gdrcopy` installed in `/opt/mellanox/gdrcopy`:
  - `export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/opt/mellanox/gdrcopy/src`
- `gdrdrv` kernel module is active and running on the system
- `nvidia-peermem` kernel module is active and running on the system
- DOCA and DPDK library path are preset in the `LD_LIBRARY_PATH` environment variable:
  - `export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/opt/mellanox/dpdk/lib/x86_64-linux-gnu:/opt/mellanox/doca/lib/x86_64-linux-gnu`
- Network card is correctly configured and has a direct PCIe connection with the GPU

If the application is running in a multi-GPU environment, either choose the GPU to use by setting the `CUDA_VISIBLE_DEVICES` environment variable, or add this simple piece of code in the `gpu_packet_processing.c` file in the main function right after the `doca_argp_start` function:

```
int cuda_id;
cudaDeviceGetByPCIBusId(&cuda_id, app_cfg.gpu_pcie_addr);
cudaFree(0);
cudaSetDevice(cuda_id);
```

Chapter 3. Application Architecture

The application manages different types of traffic differently, dedicating up to 4 receive queues to each one using DOCA Flow with RSS mode to assign each packet to the right queue.

The more queues the application uses, the higher is the degree of parallelism in how receive data is processed and how long it takes.

Tip: It is highly recommended to use more than one receive queue for 100Gb/s or higher network traffic throughput.

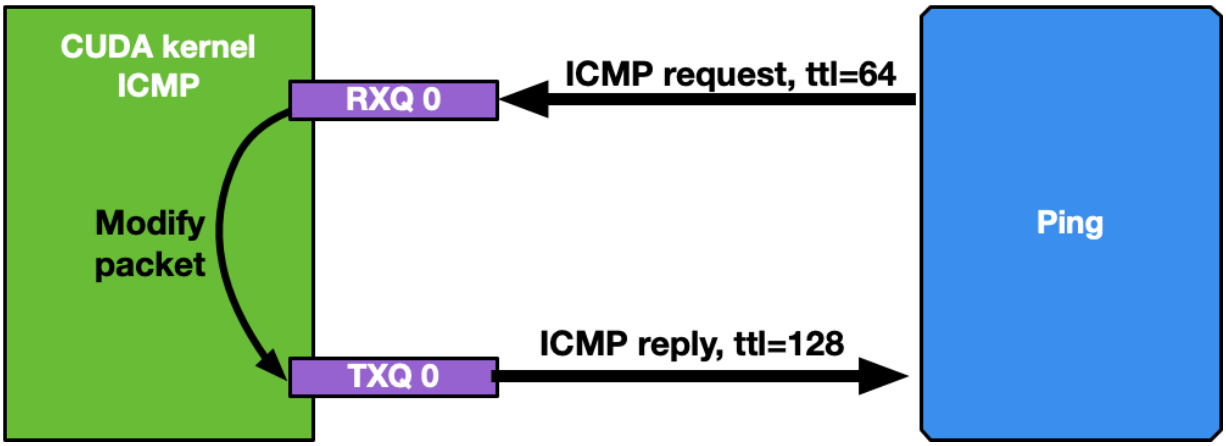
3.1. ICMP Network Traffic

If the network interface used for the application has an IP address, it is possible to ping that interface. ICMP packets are received by a dedicated CUDA kernel (file `gpu_kernels/receive_icmp.cu`) which:

1. Receives packets using the DOCA GPUNetIO CUDA warp-level function `doca_gpu_dev_eth_rxq_receive_warp`.
2. Checks if the packet is an ICMP echo request.
3. Forwards the same packet, modifying some header info (e.g., swapping MAC and IP addresses, changing ICMP packet type).
4. Pushes the modified packet into the send queue using the DOCA GPUNetIO thread-level function `doca_gpu_dev_eth_txq_send_enqueue_strong`.
5. Sends the packet using the DOCA GPUNetIO thread-level functions `doca_gpu_dev_eth_txq_commit_strong` and `doca_gpu_dev_eth_txq_push`.

Note: This is not a compute intensive use case so a single CUDA warp with only one receive queue and one send queue is enough to keep up with a decent latency.

By default, the OS CPU ping TTL is set to 64. Therefore, to be sure the GPU is actually replying to ICMP ping requests, TTL is set to 128 in this application.



Motivations for this use case:

- Provide an easy tool to check connectivity between packet the generator machine and the DOCA application machine
- Have a sense of network latency between the two machines using a well-known tool like ping
- Show an easy way to receive and forward modified packets
- Provide a warp-level implementation of a CUDA kernel receiving and forwarding traffic

Assuming the network interface to ping has the IP address `192.168.1.1`, this is the expected output:

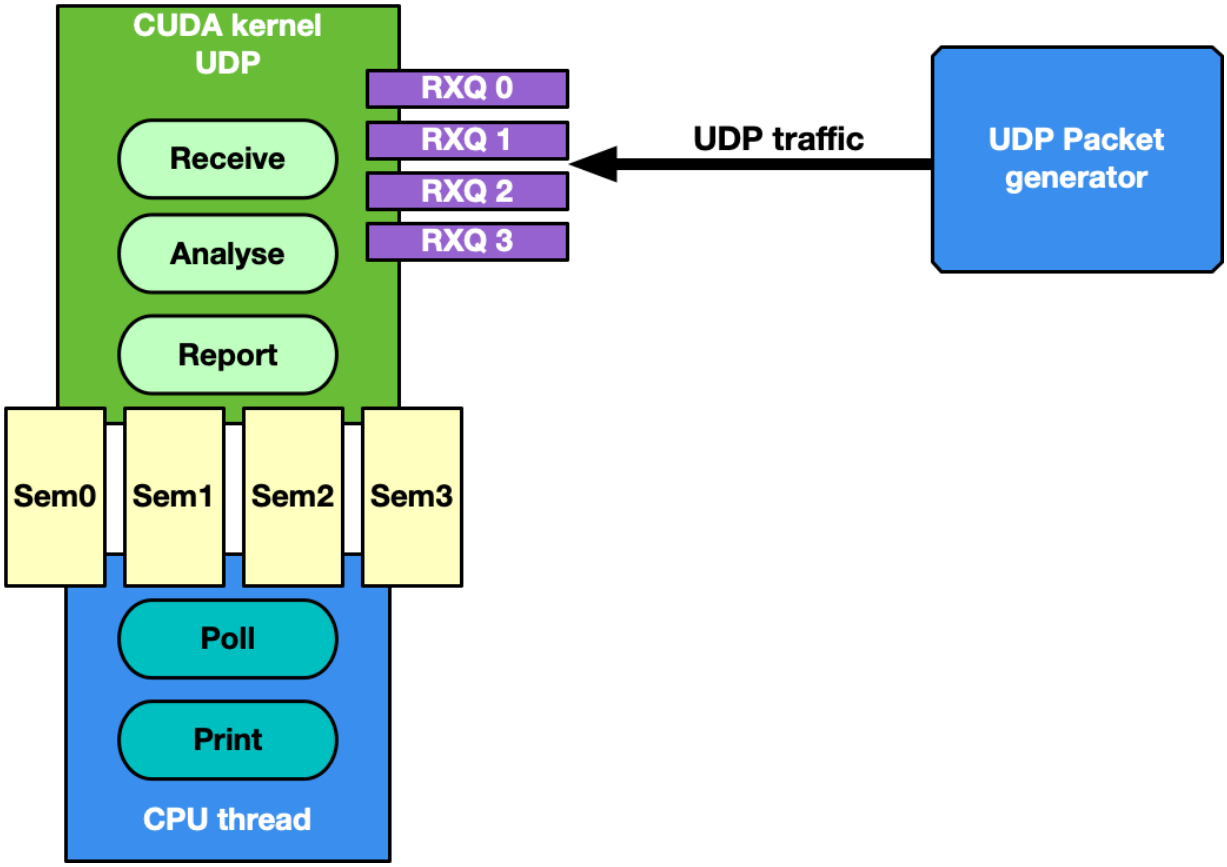
```
$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.324 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=64 time=0.332 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=64 time=0.299 ms
64 bytes from 192.168.1.1: icmp_seq=4 ttl=64 time=0.309 ms
64 bytes from 192.168.1.1: icmp_seq=5 ttl=64 time=0.323 ms
64 bytes from 192.168.1.1: icmp_seq=6 ttl=64 time=0.300 ms
64 bytes from 192.168.1.1: icmp_seq=7 ttl=64 time=0.274 ms
64 bytes from 192.168.1.1: icmp_seq=8 ttl=64 time=0.314 ms
64 bytes from 192.168.1.1: icmp_seq=9 ttl=64 time=0.327 ms
64 bytes from 192.168.1.1: icmp_seq=10 ttl=64 time=0.384 ms
# At this point, the DOCA application has been started on the 192.168.1.1 interface
# TTL becomes 128 as it's the GPU replying to ICMP requests now instead of the OS
64 bytes from 192.168.1.1: icmp_seq=11 ttl=128 time=0.346 ms
64 bytes from 192.168.1.1: icmp_seq=12 ttl=128 time=0.274 ms
64 bytes from 192.168.1.1: icmp_seq=13 ttl=128 time=0.294 ms
64 bytes from 192.168.1.1: icmp_seq=14 ttl=128 time=0.240 ms
64 bytes from 192.168.1.1: icmp_seq=15 ttl=128 time=0.273 ms
64 bytes from 192.168.1.1: icmp_seq=16 ttl=128 time=0.238 ms
64 bytes from 192.168.1.1: icmp_seq=17 ttl=128 time=0.252 ms
64 bytes from 192.168.1.1: icmp_seq=18 ttl=128 time=0.232 ms
64 bytes from 192.168.1.1: icmp_seq=19 ttl=128 time=0.278 ms
64 bytes from 192.168.1.1: icmp_seq=20 ttl=128 time=0.276 ms
64 bytes from 192.168.1.1: icmp_seq=21 ttl=128 time=0.294 ms
......
```

3.2. UDP Network Traffic

This is the most generic use case of receive-and-analyze packet headers. Designed to keep-up with 100Gb/s of incoming network traffic, the CUDA kernel responsible for the UDP traffic dedicates one CUDA block of 512 CUDA threads (file `gpu_kernels/receive_udp.cu`) to a different Ethernet UDP receive queue.

The data path loop is:

1. Receive packets using the DOCA GPUNetIO CUDA block-level function `doca_gpu_dev_eth_rxq_receive_block`.
2. Each CUDA thread works on a subset of received packets.
3. DOCA buffer containing the packet is retrieved.
4. Packet payload is analyzed to differentiate between DNS packets from other UDP generic packets.
5. Packet payload is wiped-out to ensure that old stale packets are not analyzed again.
6. Each CUDA block reports to the CPU thread statistics about types of received packets through a DOCA GPUNetIO semaphore.
7. CPU thread polls on semaphores to retrieve and print the statistics to the console.



The motivation for this use case is mostly to provide an application template to:

- Receive and analyze packet headers to differentiate across different UDP protocols
- Report statistics to the CPU through the DOCA GPUNetIO semaphore

Several well-known packet generators can be used to test this mode like T-Rex or DPDK testpmd.

3.3. TCP Network Traffic and HTTP Echo Server

By default, the TCP flow management is the same as UDP: Receive TCP packets and analyze their headers to report to the CPU statistics about the types of received packets.

This is good for passive traffic analyzers or sniffers but sometimes a packet processing application needs to receive packets directly from TCP peers which implies the establishment of a TCP-reliable connection through the 3-way handshake method. Therefore, it is possible to enable TCP "server" mode through the `-s` command-line flag which enables an "HTTP echo server" mode where the CPU and GPU cooperate to establish a TCP connection and process TCP data packets.

Specifically, in this case there are two different sets of receive queues:

- CPU DPDK receive queues which receive TCP "control" packets (e.g. SYN, FIN or RST)
- DOCA GPUNetIO receive queues to receive TCP "data" packets

This distinction is possible thanks to DOCA Flow capabilities.

The application's flow requires CPU and GPU collaboration as described in the following subsections.

3.3.1. Step 1: TCP Connection Establishment

A CPU thread through DPDK queues receives a TCP SYN packet from a remote TCP peer. The CPU thread establishes a TCP reliable connection (replies with a TCP SYN-ACK packet) with the peer and uses DOCA Flow to create a new steering rule to redirect TCP data packets to one of the DOCA GPUNetIO receive queues. The new steering rule excludes control packets (e.g., SYN, FIN or RST).

3.3.2. Step 2: TCP Data Processing

The CUDA kernel responsible for TCP processing receives TCP data packets and performs TCP packet header analysis. If it receives an HTTP GET request, it stores the relevant packet's info in the next item of a DOCA GPUNetIO semaphore, setting it to `READY`.

3.3.3. Step 3: HTTP Echo Server

A second CUDA kernel responsible for HTTP processing polls the DOCA GPUNetIO semaphore. Once it detects the update of the next item to `READY`, it reads the HTTP GET packet info and crafts an HTTP response packet with an HTML page.

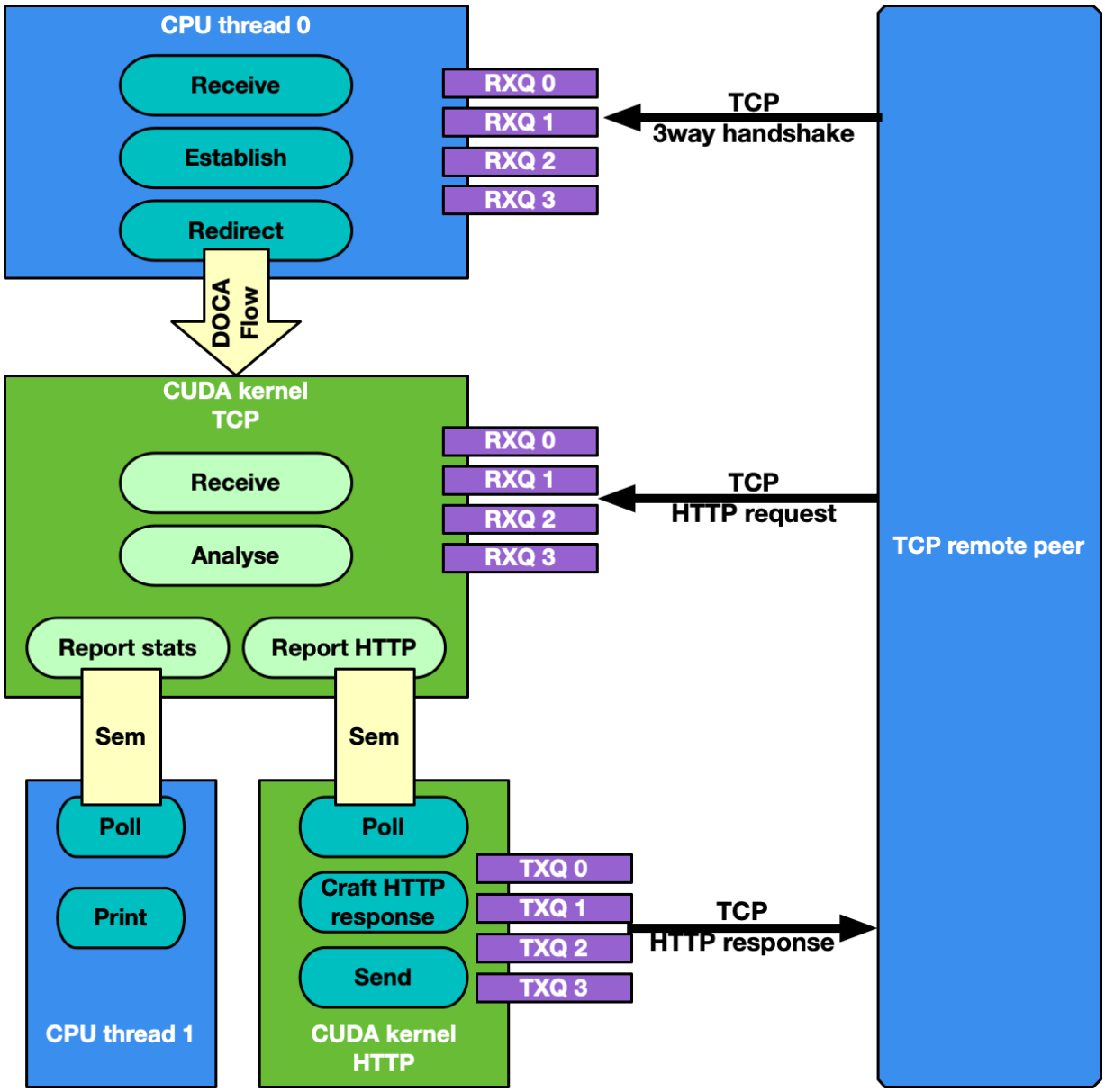
If the request is about `index.html` or `contacts.html`, the CUDA kernel replies with the appropriate HTML page using a `200 OK` code. For all other requests, the it returns a "Page not found" and `404 Error` code.

HTTP response packets are sent by this second HTTP CUDA kernel using DOCA GPUNetIO.

Note: Care must be taken to maintain TCP sequence/ack numbers in the packet headers.

3.3.4. Step 4: TCP Connection Closure

If the CPU receives a TCP FIN packet through the DPDK queues, it closes the connection with the remote TCP peer and removes the DOCA Flow rule from the DOCA GPUNetIO queues so the CUDA kernel cannot receive anymore packets from that TCP peer.



Motivations for this use case:

- Receiving and analyzing packet headers to differentiate across different TCP protocols
- Processing TCP packets on GPU in passive mode (sniffing) and active mode (reliable connection)
- Having a DOCA-DPDK application able to establish a TCP reliable connection without using any OS socket and bypassing kernel routines
- Having CUDA-kernel-to-CUDA-kernel communication through a DOCA GPUNetIO semaphore
- Showing how to create and send a packet from scratch with DOCA GPUNetIO

Assuming the network interface used to run the application has the IP address `192.168.1.1` , it is possible to test this HTTP echo server mode using simple tools like `curl` or `wget`.

Example with `curl`:

```
$ curl http://192.168.1.1/index.html -ivvv

*   Trying 192.168.1.1:80...
* Connected to 192.168.1.1 (192.168.1.1) port 80 (#0)
> GET /index.html HTTP/1.1
> Host: 192.168.1.1
> User-Agent: curl/7.81.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
HTTP/1.1 200 OK
< Date: Sun, 30 Apr 2023 20:30:40 GMT
Date: Sun, 30 Apr 2023 20:30:40 GMT
< Content-Type: text/html; charset=UTF-8
Content-Type: text/html; charset=UTF-8
< Content-Length: 158
Content-Length: 158
< Last-Modified: Sun, 30 Apr 2023 22:38:34 GMT
Last-Modified: Sun, 30 Apr 2023 22:38:34 GMT
< Server: GPUNetIO
Server: GPUNetIO
< Accept-Ranges: bytes
Accept-Ranges: bytes
< Connection: keep-alive
Connection: keep-alive
< Keep-Alive: timeout=5
Keep-Alive: timeout=5

<
<html>
  <head>
    <title>GPUNetIO index page</title>
  </head>
  <body>
    <p>Hello World, the GPUNetIO server Index page!</p>
  </body>
</html>

* Connection #0 to host 192.168.1.1 left intact
```

Chapter 4. DOCA Libraries

This application leverages the following DOCA libraries:

- DOCA GPUNetIO
- DOCA Ethernet
- DOCA Flow

Chapter 5. Configuration Flow

The following explains the application's flow, highlighting main code blocks and functions:

1. Parse application argument.
```
doca_argp_init();
register_application_params();
doca_argp_start();
```
2. Initialize network device as DOCA device, initialize DPDK, and get device DPDK port ID.
```
init_doca_device();
```
Calls `rte_eal_init()` with empty flags to initialize EAL resources.
3. Initialize a GPU device, creating a DOCA GPUNetIO handle for it.
```
doca_gpu_create();
```
4. Initialize DOCA Flow, starting the DPDK port.
```
init_doca_flow();
```
Flags to initialize DOCA Flow are VNF, HW steering, and isolated mode (to prevent the default RSS flows from interfering with the GPUNetIO queues).
5. Create RX and TX queue related objects (i.e., Ethernet handlers, GPUNetIO handlers, flow rules, semaphores) to manage UDP, TCP and ICMP flows.
```
create_udp_queues();
create_tcp_queues();
create_icmp_queues();
/* Depending on TCP mode (HTTP server or not) properly connect different DOCA
 Flow pipes */
create_root_pipe();
```
6. Allocate generic exit flag. All CUDA kernels periodically poll on this flag. If the CPU set it to 1, CUDA kernels exit from their main loop and return.
```
doca_gpu_mem_alloc(gpu_dev, sizeof(uint32_t), alignment, DOCA_GPU_MEM_GPU_CPU,
 (void **)&gpu_exit_condition, (void **)&cpu_exit_condition);
```
7. Launch CUDA kernels, each on a different stream.
```
kernel_receive_udp(rx_udp_stream, gpu_exit_condition, &udp_queues);
kernel_receive_tcp(rx_tcp_stream, gpu_exit_condition, &tcp_queues,
 app_cfg.http_server);
kernel_receive_icmp(rx_icmp_stream, gpu_exit_condition, &icmp_queues);
if (app_cfg.http_server)
    kernel_http_server(tx_http_server, gpu_exit_condition, &tcp_queues,
 &http_queues);
```
8. Launch the CPU thread responsible to poll on DOCA GPUNetIO semaphores and print UDP and TCP stats on the console.
```
rte_eal_remote_launch((void *)stats_core, NULL, current_lcore);
```

9. Launch CPU thread responsible for managing TCP 3-way handshake connections.

```
if (app_cfg.http_server) {
    ...
    rte_eal_remote_launch(tcp_cpu_rss_func, &tcp_queues, current_lcore);
}
```

10. Wait for the user to send a signal to quit the application. When this happens, the signal handler function sets the `force_quit` flag to true which causes the main thread to move forward and set the exit condition to 1.

```
while (DOCA_GPUNETIO_VOLATILE(force_quit) == false);
DOCA_GPUNETIO_VOLATILE(*cpu_exit_condition) = 1;
```

11. Wait for CUDA kernels to exit and finalize all DOCA Flow and GPUNetIO resources.

```
cudaStreamSynchronize(rx_udp_stream);
cudaStreamSynchronize(rx_tcp_stream);
cudaStreamSynchronize(rx_icmp_stream);
if (app_cfg.http_server)
    cudaStreamSynchronize(tx_http_server);
destroy_flow_queue();
doca_gpu_destroy();
```

Chapter 6. Running the Application

1. Refer to the following documents:
   - NVIDIA DOCA Installation Guide for Linux for details on how to install BlueField-related software.
   - NVIDIA DOCA Troubleshooting Guide for any issue you may encounter with the installation, compilation, or execution of DOCA applications.
   - NVIDIA DOCA Applications Overview for additional compilation instructions and development tips of DOCA applications.
2. The `gpu_packet_processing` binary is located under `/opt/mellanox/doca/applications/gpu_packet_processing/bin/doca_gpu_packet_processing`. To build all the applications together, run:
   ```
   cd /opt/mellanox/doca/applications/
   meson build
   ninja -C build
   ```
3. To build only this application:
   a). Edit the following flags in `/opt/mellanox/doca/applications/meson_options.txt`:
      - Set `enable_all_applications` to `false`
      - Set `enable_gpu_support` to `true`
      - Set `enable_gpu_packet_processing` to `true`

   b). Run the commands in step 2.

   Note: `doca_gpu_packet_processing` is created under `./build/gpu_packet_processing/src/`.

   Application usage:
   ```
   Usage: doca_gpu_packet_processing [DOCA Flags] [Program Flags]

   DOCA Flags:
     -h, --help                        Print a help synopsis
     -v, --version                     Print program version information
     -l, --log-level                   Set the log level for the program
    <CRITICAL=20, ERROR=30, WARNING=40, INFO=50, DEBUG=60>

   Program Flags:
     -g, --gpu <GPU PCIe address>      GPU PCIe address to be used by the app
     -n, --nic <NIC PCIe address>      DOCA device PCIe address used by the app
     -q, --queue <GPU receive queues>  DOCA GPUNetIO receive queue per flow
   ```

```
  -s, --httpserver <Enable GPU HTTP server> Enable GPU HTTP server mode
```

4. To run the application on the host, assuming a GPU PCIe address `ca:00.0` and NIC PCIe address `17:00.0` with 2 GPUNetIO receive queues:

```
doca_gpu_packet_processing -n 17:00.0 -g ca:00.0 -q 2
```

Note: Refer to section "Running DOCA Application on Host" in the NVIDIA DOCA Virtual Functions User Guide.

Chapter 7. Arg Parser DOCA Flags

Refer to NVIDIA DOCA Arg Parser Programming Guide for more information.

Flag Type	Short Flag	Long Flag/JSON Key	Description
DPDK flags	`a`	`devices`	Adds a PCIe device into the list of devices to probe.
	`l`	`core-list`	List of cores to run on.
General flags	`l`	`log-level`	Sets the log level for the application: ▶ CRITICAL=20 ▶ ERROR=30 ▶ WARNING=40 ▶ INFO=50 ▶ DEBUG=60
	`v`	`version`	Prints program version information.
	`h`	`help`	Prints a help synopsis.
Program flags	`g`	`gpu`	GPU PCIe address in `<bus>:<device>.<function>` format. This can be obtained using the `nvidia-smi` or `lspci` commands.
	`n`	`nic`	Network card port PCIe address in `<bus>:<device>.<function>` format. This can be obtained using the `lspci` command.
	`q`	`queue`	Number of receive queues to use in the example. Default is 1, maximum allowed is 4.
	`s`	`httpserver`	Enable the TCP HTTP server mode. With

Flag Type	Short Flag	Long Flag/JSON Key	Description
			this flag, TCP packets are not received by GPUNetIO as regular sniffer as it requires a TCP 3-way handshake to establish a reliable connection first.

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation nor any of its direct or indirect subsidiaries and affiliates (collectively: “NVIDIA”) make no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assume no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer (“Terms of Sale”). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, “MATERIALS”) ARE BEING PROVIDED “AS IS.” NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA’s aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA, the NVIDIA logo, and Mellanox are trademarks and/or registered trademarks of Mellanox Technologies Ltd. and/or NVIDIA Corporation in the U.S. and in other countries. The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a world¬wide basis. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
http://www.nvidia.com