Table of contents

Introduction
Prerequisites
Environment
Architecture
Objects
Device and Representor
Memory Buffers
Configuration Phase
Configurations
Mandatory Configurations
Device Support
Buffer Support
Execution Phase
Tasks
SHA Task
Partial-SHA Task
Events
State Machine
Idle
Starting
Running
Stopping

Alternative Datapath Options

DOCA SHA Samples

Running the Samples

Samples

SHA Create

SHA-Partial Create
This guide provides instructions on building and developing applications that calculate message digest using the SHA1, SHA2-256 or SHA2-512 algorithms.

**Introduction**

![Note]

The DOCA SHA library is currently supported at alpha level.

The library provides an API for executing SHA operations on DOCA buffers, where the buffers reside in either local memory (i.e., within the same host) or host memory accessible by the NVIDIA® BlueField®-2 device (remote memory). Using DOCA SHA, complex cryptographic hash operations can be easily executed in an optimized, hardware-accelerated manner.

![Note]

NVIDIA® BlueField®-3 does not support this library because it has no SHA acceleration engine.

This document is intended for software developers wishing to accelerate their applications' SHA calculations typically used in digital signature schemes or hash-based message authentication code calculations.

**Prerequisites**

This library follows the architecture of a DOCA Core context, it is recommended to read the following sections before:

- [DOCA Core Execution Model](#)
- [DOCA Core Device](#)
### Environment

DOCA SHA-based applications can run either on the host machine or on the BlueField-2 DPU target.

DOCA SHA calculations from the host to BlueField and vice versa can only be run when the DPU is configured in **DPU mode**.

### Architecture

DOCA SHA is a **DOCA Core Context**. This library leverages the DOCA Core architecture to expose asynchronous tasks/events offloaded to hardware.

SHA can be used to calculate message digest as illustrated in the following diagrams:

- SHA from local memory to local memory:

  ![SHA from local memory to local memory](image)

- Using the DPU to do SHA using the memory between the host and the DPU:
Objects

Device and Representor

The library requires a DOCA device to operate. The device is used to access memory and perform the actual SHA calculation. See [DOCA Core Device Discovery](#).

For the same BlueField DPU, it does not matter which device is used (i.e., PF/VF/SF) as these devices utilize the same hardware component. If there are multiple DPUs, then it is possible to create a SHA instance per DPU, providing each instance with a device from a different DPU.
To access non-local memory (i.e., from the host to DPU or vice versa), the DPU side of the application must choose a device with an appropriate representor (see **DOCA Core Device Representor Discovery**). The device must stay valid for as long as the SHA instance is not destroyed.

**Memory Buffers**

The SHA task requires at least two DOCA buffers containing the destination and the source. Depending on the allocation pattern of the buffers, refer to the DOCA Core Inventory Types table.

Buffers must not be modified or read during the SHA operation. For information on what kind of memory is supported, refer to the table in section "Buffer Support".

**Configuration Phase**

To start using the library, users must go through a configuration phase as described in **DOCA Core Context Configuration Phase**.

This section describes how to configure and start the context to allow the execution of tasks and retrieval of events.

**Configurations**

The context can be configured to match the application use case.

To find if a configuration is supported or its min/max value, refer to section "Device Support".

**Mandatory Configurations**

These configurations must be set by the application before attempting to start the context:

- At least one task/event type must be configured. See configuration of tasks and/or events in sections "Tasks" and "Events" respectively for information.
A device with appropriate support must be provided upon creation

Device Support

DOCA SHA requires a device to operate. For information on choosing a device, see DOCA Core Device Discovery.

As device capabilities may change in the future (see DOCA Core Device Support) it is recommended to select your device using the following methods:

- doca_sha_cap_task_hash_get_supported
- doca_sha_cap_task_partial_hash_get_supported

Some devices can allow different capabilities such as:

- The maximum number of tasks
- The maximum source buffer size
- The minimum destination buffer size
- The maximum supported number of elements in DOCA linked-list buffer
- Check whether SHA1, SHA2-256 or SHA2-512 is supported

Buffer Support

Tasks support buffers with the following features:

<table>
<thead>
<tr>
<th>Buffer Type</th>
<th>Source Buffer</th>
<th>Destination Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local mmap buffer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mmap from PCIe export buffer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mmap from RDMA export buffer</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Buffer Type</td>
<td>Source Buffer</td>
<td>Destination Buffer</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Linked list buffer</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

## Execution Phase

This section describes execution on the CPU using DOCA Core Progress Engine.

## Tasks

DOCA SHA exposes asynchronous tasks that leverage DPU hardware according to DOCA Core architecture.

### SHA Task

The SHA task `doca_sha_task_hash` allows one-shot SHA calculation using buffers as described in section "Buffer Support". One-shot means that the source buffer is used as a whole input, therefore, the SHA operation is completed after this task completion event arrives.

A single:

```
doca_sha_task_hash
```

## Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_sha_task_hash_set_conf</code></td>
<td><code>doca_sha_cap_task_hash_get_supported</code></td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_sha_task_hash_set_conf</code></td>
<td>-</td>
</tr>
<tr>
<td>Maximal source buffer size</td>
<td>-</td>
<td><code>doca_sha_cap_get_max_src_buf_size</code></td>
</tr>
<tr>
<td>Maximum source buffer list size</td>
<td>-</td>
<td><code>doca_sha_cap_get_max_list_buf_num_elem</code></td>
</tr>
</tbody>
</table>
Task Input

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source buffer</td>
<td>Buffer pointing to the memory to be used for SHA calculation</td>
<td>Only the data residing in the data segment is to be used</td>
</tr>
<tr>
<td>Destination buffer</td>
<td>Buffer pointing to the memory used for writing the SHA calculation result</td>
<td>The SHA result is appended to the tail segment</td>
</tr>
<tr>
<td>SHA algorithm type</td>
<td>SHA algorithm to be used in SHA calculation</td>
<td>Must be one of DOCA_SHA_ALGORITHM_SHA1, DOCA_SHA_ALGORITHM_SHA256, DOCA_SHA_ALGORITHM_SHA512</td>
</tr>
</tbody>
</table>

Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The SHA calculation of data from the source buffer is successfully completed and the result is written to the destination buffer
- The destination buffer data segment is extended to include the SHA result data
Task Completion Failure

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects are not modified
- The destination buffer contents may be modified

Task Limitations

- The operation is not atomic
- Once the task is submitted, the source and destination should not be read/written to
- Other limitations are described in DOCA Core Task

Partial-SHA Task

The partial-SHA task `doca_sha_task_partial_hash` allows stateful SHA calculation for a collection of messages. Using buffers as described in section "Buffer Support".

Stateful means that the input data is composed of many segments (may be spatial or timely non-consecutive), therefore, its SHA calculation requires more than one one-shot SHA operation to finish. During any stateful operation, other independent SHA tasks can also be executed.
**Task Configuration**

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_sha_task_partial_hash_set_conf</code></td>
<td><code>doca_sha_cap_task_partial_hash_get_supported</code></td>
</tr>
<tr>
<td>Number of tasks</td>
<td><code>doca_sha_task_partial_hash_set_conf</code></td>
<td></td>
</tr>
<tr>
<td>Maximal source buffer size</td>
<td>-</td>
<td><code>doca_sha_cap_get_max_src_buf_size</code></td>
</tr>
<tr>
<td>Maximum source buffer list size</td>
<td>-</td>
<td><code>doca_sha_cap_get_max_list_buf_num_elem</code></td>
</tr>
<tr>
<td>Minimum destination buffer size</td>
<td>-</td>
<td><code>doca_sha_cap_get_min_dst_buf_size</code></td>
</tr>
<tr>
<td>SHA block size</td>
<td></td>
<td><code>doca_sha_cap_get_partial_hash_block_size</code></td>
</tr>
</tbody>
</table>
Task Input

Common input as described in DOCA Core Task.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source buffer</td>
<td>Buffer pointing to the memory to be used for SHA calculation</td>
<td>Only the data residing in the data segment is to be used. And the data length for the non-last data segment must be multiple of the SHA block size queried by doca_sha_cap_get_partial_hash_block_size</td>
</tr>
<tr>
<td>Destination buffer</td>
<td>Buffer pointing to the memory is used for writing the SHA calculation result</td>
<td>The SHA result is appended to the tail segment. During the whole calculation process, this buffer cannot be modified.</td>
</tr>
<tr>
<td>SHA algorithm type</td>
<td>SHA algorithm to be used in SHA calculation</td>
<td>Must be one of DOCA_SHA_ALGORITHM_SHA1, DOCA_SHA_ALGORITHM_SHA256, DOCA_SHA_ALGORITHM_SHA512</td>
</tr>
<tr>
<td>Whether the current source buffer is the last segment</td>
<td>Indicate whether the current source Buffer is the last segment data to be used for partial-SHA calculation</td>
<td>Use doca_sha_task_partial_hash_set_is_final_buf to set this property</td>
</tr>
<tr>
<td>Set source buffer</td>
<td>Use to set the subsequent source segment buffer after the initial</td>
<td></td>
</tr>
<tr>
<td></td>
<td>doca_sha_task_partial_hash_task is allocated</td>
<td></td>
</tr>
</tbody>
</table>

Task Output

Common output as described in DOCA Core Task.

Task Completion Success
After the task completes successfully, the following happens:

- The SHA calculation of data from the source buffer is successfully completed and the result is written to the destination buffer
- The destination buffer data segment is extended to include the SHA result data

**Task Completion Failure**

If the task fails midway:

- The context may enter stopping state if a fatal error occurs
- The source and destination `doca_buf` objects is not modified
- The destination buffer contents may be modified

**Task Limitations**

- The operation is not atomic
- Once the task is submitted, the source and destination should not be read/written to
- Other limitations are described in DOCA Core Task

**Events**

DOCA SHA exposes asynchronous events to notify about changes that happen unexpectedly according to the DOCA Core architecture.

The only events SHA exposes are common events as described in DOCA Core Event.
State Machine

The DOCA SHA library follows the context state machine as described in DOCA Core Context State Machine.

The following section describes moving states and what is allowed in each state.

Idle

In this state, it is expected that the application either:

- Destroys the context
- Starts the context

Allowed operations:

- Configuring the context according to section "Configurations"
- Starting the context

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Create the context</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop after making sure all tasks have been freed</td>
</tr>
<tr>
<td>Stopping</td>
<td>Call progress until all tasks are completed and freed</td>
</tr>
</tbody>
</table>

Starting

This state cannot be reached.

Running
In this state, it is expected that the application:

- Allocates and submits tasks
- Calls progress to complete tasks and/or receive events

Allowed operations:

- Allocating previously configured task
- Submitting a task
- Calling stop

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Call start after configuration</td>
</tr>
</tbody>
</table>

**Stopping**

In this state, it is expected that the application:

- Calls progress to complete all inflight tasks (tasks complete with failure)
- Frees any completed tasks

Allowed operations:

- Calling progress

It is possible to reach this state as follows:

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>Call progress and fatal error occurs</td>
</tr>
<tr>
<td>Running</td>
<td>Call stop without freeing all tasks</td>
</tr>
</tbody>
</table>
Alternative Datapath Options

DOCA SHA only supports datapath on the CPU. See section "Execution Phase".

DOCA SHA Samples

This section describes DOCA SHA samples based on the DOCA SHA library.

The samples in this section illustrate how to use the DOCA SHA API to do the following:

- Do SHA calculation of contents of a buffer, and write result to another buffer
- Chop the contents of a buffer into a collection of segments, and do partial-SHA calculation of this collection of segments, and write result to another

Info

All the DOCA samples described in this section are governed under the BSD-3 software license agreement.

Running the Samples

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 samples.

2. To build a given sample:

   ```bash
cd /opt/mellanox/doca/samples/doca_sha/<sample_name>
meson/tmp/build
```
3. Sample (e.g., doca_sha_create) usage:

Usage: doca_sha_create [DOCA Flags] [Program Flags]

DOCA Flags:
-h, --help Print a help synopsis
-v, --version Print program version information
-l, --log-level Set the (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
--sdk-log-level Set the SDK (numeric) log level for the program <10=DISABLE, 20=CRITICAL, 30=ERROR, 40=WARNING, 50=INFO, 60=DEBUG, 70=TRACE>
-j, --json <path> Parse all command flags from an input json file

Program Flags:
-d, --data user data

4. For additional information per sample, use the -h option:

/tmp/build/doca_<sample_name>-h

Samples

SHA Create

This sample illustrates how to perform SHA calculation with DOCA SHA.
The sample logic includes:

1. Locating DOCA device.
2. Initializing required DOCA Core structures.
3. Setting the task_pool configuration for doca_sha_task_hash.
4. Populating DOCA memory map with two relevant buffers.
5. Allocating element in DOCA buffer inventory for each buffer.
6. Allocating and initializing a doca_sha_task_hash.
7. Submitting the task.
8. Retrieving task result once it is done.

Reference:

- /opt/mellanox/doca/samples/doca_sha/sha_create/sha_create_sample.c
- /opt/mellanox/doca/samples/doca_sha/sha_create/sha_create_main.c
- /opt/mellanox/doca/samples/doca_sha/sha_create/meson.build

**SHA-Partial Create**

This sample illustrates how to perform partial-SHA calculation for a collection of data segments with DOCA SHA.

The sample logic includes:

1. Locating DOCA device.
2. Initializing the required DOCA Core structures.
3. Setting the task_pool configuration for doca_sha_task_partial_hash.
4. Chopping the source data into a collection of data segments according to the selected SHA algorithm's block size
5. Populating DOCA memory map with needed buffers for all source data segments and destination buffer.

6. Allocating element in DOCA buffer inventory for the first source buffer and destination buffer.

7. Allocating and initializing a `doca_sha_task_partial_hash` with the first source buffer and the destination buffer.

8. Iteratively repeating the following sub-steps until all data segments are consumed:
   1. Submitting the `doca_sha_task_partial_hash`.
   2. Waiting for the submitted task to finish.
   3. Allocating a `doca_buf` for the next source segment and use `doca_sha_task_partial_hash_set_src` to set it as source buffer of the above allocated task.
   4. If it is the final segment, use `doca_sha_task_partial_hash_set_is_final_buf` to mark it in the allocate task.

9. Retrieving the result of the final iteration in the destination buffer as the full partial-SHA calculation result.

10. Destroying all SHA and DOCA Core structures.

Reference:

- `/opt/mellanox/doca/samples/doca_sha/sha_partial_create/sha_partial_create_sample.c`
- `/opt/mellanox/doca/samples/doca_sha/sha_partial_create/sha_partial_create_main.c`
- `/opt/mellanox/doca/samples/doca_sha/sha_partial_create/meson.build`