DOCA Erasure Coding
## Table of contents

**Introduction**
- Glossary

**Prerequisites**

**Environment**

**Architecture**
- Flows
- Create Redundancy Blocks
- Recover Block
- Objects
- Device and Device Representor
- Memory Buffers

**Configuration Phase**
- Configurations
- Mandatory Configurations
- Device Support
- Buffer Support

**Execution Phase**
- Matrix Generate
- Matrix Type
- Matrix Functionality
- Tasks
<table>
<thead>
<tr>
<th>Task Batching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Galois Mul Task</td>
</tr>
<tr>
<td>Create Task</td>
</tr>
<tr>
<td>Update Task</td>
</tr>
<tr>
<td>Recover Task</td>
</tr>
</tbody>
</table>

**DOCA Erasure Coding Samples**

<table>
<thead>
<tr>
<th>Sample Prerequisites</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running the Sample</td>
</tr>
</tbody>
</table>

**Samples**

<table>
<thead>
<tr>
<th>Erasure Coding Recover</th>
</tr>
</thead>
</table>
This guide provides instructions on how to use the DOCA Erasure Coding API.

Introduction

Note

This library is currently supported at alpha version.

The DOCA Erasure Coding (known also as forward error correction or FEC) library provides an API to encode and decode data using hardware acceleration, supporting both host and NVIDIA® BlueField®-3 (and higher) DPU memory regions.

DOCA Erasure Coding recovers lost data fragments by creating generic redundancy fragments (backup). Each redundancy block that the library creates can help recover any block in the original data should a total loss of fragment occur. This increases data redundancy and reduces data overhead.

The library provides an API for executing erasure coding (EC) operations on DOCA buffers residing in either the DPU or host memory.

This document is intended for software developers wishing to accelerate their application's EC memory operations.

Glossary

Familiarize yourself with the following terms to better understand the information in this document:

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Original data, original blocks, blocks of original data to be protected/preserved</td>
</tr>
<tr>
<td>Coding matrix</td>
<td>Coefficients, the matrix used to generate the redundancy blocks and recovery</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Redundancy blocks</td>
<td>Codes; encoded data; the extra blocks that help recover data loss</td>
</tr>
<tr>
<td>Encoding</td>
<td>The process of creating the redundancy blocks. Encoded data is referred to as the original blocks or redundancy blocks.</td>
</tr>
<tr>
<td>Decoding</td>
<td>The process of recovering the data. Decoded data is referred to as the original blocks alone.</td>
</tr>
</tbody>
</table>

**Prerequisites**

DOCA Erasure Coding library follows the architecture of a DOCA Core Context, it is recommended read the following sections before:

- [DOCA Core Execution Model](#)
- [DOCA Core Device](#)
- [DOCA Core Memory Subsystem](#)

**Environment**

DOCA Erasure Coding-based applications can run either on the host machine or on the DPU target (NVIDIA® BlueField®-3 and above).

Erasure Coding can only be run with DPU configured in DPU mode as described in [NVIDIA BlueField Modes of Operation](#).

**Architecture**

DOCA Erasure Coding is a [DOCA Context](#) as defined by DOCA Core. This library leverages the DOCA Core architecture to expose asynchronous tasks/events that are offloaded to hardware.

The following diagram presents a high-level view of the EC transmission flow:
1. \( M \) packets are sent from the source (8 in this case).

2. Before the source send them, the source encode the data by adding to it \( T \) redundancy packets (4 in this case).

3. The packets are transmitted to the destination in UDP protocol. Some packets are lost and \( N' \) packets are received (in this case 4 packets are lost and 8 are received).

4. The destination decodes the data using all the packets available (both original data in green and redundancy data in red) and gets back the \( M \) original data packets.

**Flows**

Regular EC flow consists of the following elements:

1. Creating redundancy blocks from data (EC create).

2. Updating redundancy blocks from updated data (EC update).

3. Recovering data blocks from redundancy blocks (EC recover).
The following sections examine an M:K (where M is the original data and K is redundancy) EC.

**Create Redundancy Blocks**

The user must perform the following:

1. Input M data blocks via `doca_buf` (filled with data, each block size B).
2. Output K empty blocks via `doca_buf` (each block size B).
3. Use DOCA Erasure Coding to create a coding matrix of M by K via `doca_buf`.
4. Use DOCA Erasure Coding Create task to get the K output redundancy blocks.

**Note**

This step can be repeated in a stream use case, as the DPU would not be the recovery or update point.
Recover Block

The user must perform the following:

1. Input M-L original blocks via `doca_buf` (blocks that were not impaired).

2. Input L≤K (any) redundancy blocks via `doca_buf` (redundancy blocks originating from create/update tasks).

3. Input bitmask or array, indicating which blocks to recover.

4. Output L empty blocks via `doca_buf` (same size of data block).

5. Use DOCA Erasure Coding to create a recover coding matrix of M by L via `doca_buf` (unique per bitmask).

6. Use DOCA Erasure Coding Recover task to get the L output recovered data blocks.
Objects

Device and Device Representor

The DOCA Erasure Coding library requires a DOCA device to operate. The device is used to access memory and perform the encoding and decoding operations. See DOCA Core Device Discovery.

For same Bluefield card, it does not matter which device is used (PF/VF/SF), as all these devices utilize the same HW component. If there are multiple DPUs, then it is possible to create an EC instance per DPU, providing each instance with a device from a different DPU. To access memory that is not local (from the host to the DPU and vice versa), the DPU side of the application must pick a device with an appropriate representor. See DOCA Core Representor Device Discovery.

The device must stay valid until the EC instance is destroyed.

Memory Buffers

Executing any DOCA EC task requires two DOCA buffers, a source buffer and a destination buffer.

Depending on the allocation pattern of the buffers, refer to the Inventory Types table.
Buffers must not be modified or read during the execution of any task.

**Configuration Phase**

To start using the library, first, you need to 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 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 what the min/max value, please refer to [Device Support](#).

**Mandatory Configurations**

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

- At least 1 task/event type needs to be configured. See configuration of Tasks.
- A device with appropriate support must be provided on creation.

**Device Support**

DOCA Erasure Coding needs a device to operate. For picking a device, see [DOCA Core Device Discovery](#).

Erasure Coding can be used in BlueField-3 with some limitations (see [architecture](#)). Any device can be used PF/VF/SF.
As device capabilities may change in the future, it is recommended to choose your device using the following methods:

- `doca_ec_cap_task_galois_mul_is_supported`
- `doca_ec_cap_task_create_is_supported`
- `doca_ec_cap_task_update_is_supported`
- `doca_ec_cap_task_recover_is_supported`

Some devices can allow different capabilities as follows:

- The maximum buffer list length
- The maximum block size

---

**Note**

Current BlueField-3 limitations:

- Data block count range: 1-128
- Redundancy block count: 1-32
- Block size: 64B-128MB

---

**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>Linked list buffer</td>
<td>Depends on the device; check the <code>max_buf_list_len</code> capability</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>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>
</tbody>
</table>

### Execution Phase

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

### Matrix Generate

All tasks require a coding matrix.

### Matrix Type

DOCA EC provides 2 matrix types which are elaborated on in the following subsections.

**Cauchy**

Cauchy encoding matrix is constructed so that

\[
    a_{ij} = \frac{1}{(x_i + y_j)}
\]

Where:

- \(0 \leq i < \text{number of data blocks}\)
- \(0 \leq j < \text{number of redundancy blocks}\)
- \(x_i = i\)
- \(y_j = j + \text{number of data blocks}\)

**Vandermonde**
Vandermonde encoding matrix is constructed so that $a_{ij} = (i + 1)^j$.

Where:

- $0 \leq i < \text{number of data blocks}$
- $0 \leq j < \text{number of redundancy blocks}$

⚠️ **Warning**

Vandermonde matrix does not guarantee that every submatrix is invertible (i.e., the decode task may fail in some settings).

---

**Matrix Functionality**

**Create**

An encoding matrix is necessary for executing the create task, to create redundancy blocks.

The matrices used for updates and recovery are based on an encoding matrix.

The following subsections describe the available options for creating matrices.

**Generic**

Generic creation, with the `doca_ec_matrix_create()` function, is used for simple setup using one of matrix types provided by the library.

Input:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>One of matrix types provided by the library</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Data block count</td>
<td>The number of original data blocks</td>
</tr>
<tr>
<td>Redundancy block count</td>
<td>The number of redundancy blocks</td>
</tr>
</tbody>
</table>

**Custom**

Custom creation, with the `doxa_ec_matrix_create_from_raw()` function, is used if the desired type of matrix is not provided by the library.

**Input:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>The data of a coding matrix</td>
<td>The size of the data should be <code>data_block_count*rdnc_block_count</code></td>
</tr>
<tr>
<td>Data block count</td>
<td>The number of original data blocks</td>
<td>-</td>
</tr>
<tr>
<td>Redundancy block count</td>
<td>The number of redundancy blocks</td>
<td>-</td>
</tr>
</tbody>
</table>

**Update**

This matrix is necessary for executing the update task, to update the redundancy blocks after a change in the data blocks.

The matrix is created using the `doxa_ec_matrix_create_update()` function.

**Input:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coding matrix</td>
<td>A coding matrix created by <code>doxa_ec_matrix_create()</code> or <code>doxa_ec_matrix_create_from_raw()</code></td>
<td>-</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Update indices    | An array specifying the indices of the updated data blocks                   | • The indices must be in ascending order  
|                   |                                                                              | • The indices should match the order of the data blocks in the matrix creation function |
| Number of updates | The number of updated blocks. The length of the update indices array.        | -                                                                   |

**Recover**

This matrix is necessary for executing the recover task, to recover original data blocks.

The matrix is created using the `doca_ec_matrix_create_recover()` function.

**Input:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coding matrix</td>
<td>A coding matrix created by <code>doca_ec_matrix_create()</code> or <code>doca_ec_matrix_create_from_raw()</code></td>
<td>-</td>
</tr>
</tbody>
</table>
| Missing indices   | An array specifying the indices of the missing data blocks                   | • The indices must be in ascending order  
|                   |                                                                              | • The indices should match the order of the data blocks in the matrix creation function |
| Number of missing | The number of updated blocks. The length of the update indices array.        | -                                                                   |
Tasks

Task Batching

DOCA Erasure Coding supports task batching mode, which is a task submit mode of work that allows aggregating multiple DOCA tasks of the same type and handling them as a single unit.

Info

For more information on task batching, refer to DOCA Core Task.

DOCA Erasure Coding supports the flags DOCA_TASK_SUBMIT_FLAG_FLUSH and DOCA_TASK_SUBMIT_FLAG_OPTIMIZE_REPORTS.

Galois Mul Task

This task executes Galois multiplication between the original blocks and the coding matrix.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>doca_ec_task_galois_mul_set_conf</td>
<td>doca_ec_cap_task_galois_mul_is_supported</td>
</tr>
<tr>
<td>Maximum block size</td>
<td>–</td>
<td>doca_ec_cap_get_max_block_size</td>
</tr>
<tr>
<td>Maximum buffer list length</td>
<td>–</td>
<td>doca_ec_cap_get_max_buf_list_len</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>coding matrix</td>
<td>A coding matrix as created by doca_ec_matrix_create() or doca_ec_matrix_create_from_raw()</td>
<td>-</td>
</tr>
</tbody>
</table>
| source buffer| Source original data buffer, holding a sequence containing all original blocks (e.g., block_1, block_2, etc.); the order matters | • The data length of src_buf should be a multiplication of the block size  
• The data length should also be aligned to 64B and with a minimum size of 64B |
| destination buffer| A destination buffer for the multiplication outcome blocks. The sequence containing all multiplication outcome blocks (dst_block_1, dst_block_2, etc.) is written to it upon successful completion of the task. | • The data is written to the tail segment extending the data segment  
• The minimal available memory in dst_buf should be the number of redundancy blocks * the block size, aligned to 64B and, in any case, at least 64B. |

**Note**

If a Galois multiplication task matrix is 10x4 (i.e., 10 original blocks, 4 multiplication outcome blocks), and the block size is 64KB:

- src_buf data length should be 10x64KB = 640KB
- The available memory for writing in dst_buf should be at least 4x64KB = 256KB
Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The destination buffer holds a sequence containing all multiplication outcome blocks (e.g., dst_block_1, dst_block_2, etc.)
- The destination buffer data segment is extended to include the outcome blocks

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 has been submitted, the source and destination buffer should not be read from/written to
- Source and destination buffers must not overlap
- Other limitations are described in DOCA Core Task
Create Task

This task creates redundancy blocks for the given original data blocks using a given coding matrix.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>doca_ec_task_create_set_conf</td>
<td>doca_ec_cap_task_create_is_supported</td>
</tr>
<tr>
<td>Maximum block size</td>
<td>–</td>
<td>doca_ec_cap_get_max_block_size</td>
</tr>
<tr>
<td>Maximum buffer list length</td>
<td>–</td>
<td>doca_ec_cap_get_max_buf_list_len</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>coding matrix</td>
<td>A coding matrix created by doca_ec_matrix_create() or doca_ec_matrix_create_from_raw()</td>
<td></td>
</tr>
</tbody>
</table>
| original data blocks| Source original data buffer, holding a sequence containing all original blocks (block_1, block_2, etc.); the order matters | • The data length of original_data_blocks should be a multiplication of the block size  
• The data length should also be aligned to 64B and with a minimum size of 64B  |
| redundancy          | A destination buffer for the redundancy blocks. The sequence containing all redundancy blocks (rdnc_block_1, | • The data will be written to the tail segment extending the data segment |
### Task Output

Common output as described in [DOCA Core Task](#).

### Task Completion Success

After the task completes successfully, the following happens:

- The destination buffer holds a sequence containing all redundancy blocks (rdnc_block_1, rdnc_block_2, etc.)
- The destination buffer data segment is extended to include the redundancy blocks.

---

**Note**

If a create task matrix is 10x4 (i.e., 10 original blocks, 4 redundancy blocks), and the block size is 64KB:

- **original_data_blocks** data length should be 10x64KB = 640KB
- The available memory for writing in **redundancy_blocks** should be at least 4x64KB = 256KB

---

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>block</td>
<td>runnc_block_2, etc.) is written to upon successful completion of the task.</td>
<td>- The minimal available memory in runnc_blocks should be the number of redundancy blocks * the block size, aligned to 64B and, in any case, at least 64B</td>
</tr>
</tbody>
</table>
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 buffers should not be read from/written to
- Source and destination buffers must not overlap
- Other limitations are described in DOCA Core Task

Update Task

This task executes updates the redundancy blocks for the given original data blocks, using an update coding matrix.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td><code>doca_ec_task_update_set_conf</code></td>
<td><code>doca_ec_cap_task_update_is_supported</code></td>
</tr>
<tr>
<td>Maximum block size</td>
<td>-</td>
<td><code>doca_ec_cap_get_max_block_size</code></td>
</tr>
<tr>
<td>Maximum buffer list length</td>
<td>-</td>
<td><code>doca_ec_cap_get_max_buf_list_len</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>update matrix</td>
<td>An update coding matrix created by <code>doca_ec_matrix_create_update()</code> or <code>doca_ec_matrix_create_from_raw()</code></td>
<td>-</td>
</tr>
</tbody>
</table>
| original updated and RDNC blocks | A source buffer with data, holding a sequence containing the original data block and its updated data block, for each block that was updated, followed by the old redundancy blocks (old_data_block_i, updated_data_block_i, old_data_block_j, updated_data_block_j, ..., rdnc_block_1, rdnc_block_2, etc.) | • The data length of original_updated_and_rdnc_blocks should be a multiplication of the block size  
  • The data length should also be aligned to 64B and with a minimum size of 64B |
| updated RDNC blocks | A destination buffer for the updated redundancy blocks. The sequence containing the updated redundancy blocks (rdnc_block_1, rdnc_block_2, etc.) is written to it upon successful completion of the task | • The data is written to the tail segment extending the data segment  
  • The minimal available memory in updated_rdnc_blocks should be the number of redundancy blocks * the block size, aligned to 64B and, in any case, at least 64B |

**Note**

using an update task matrix, in which 3 data block were updated and there are 4 redundancy blocks, and the block size is 64KB:
Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task completes successfully, the following happens:

- The destination buffer holds a sequence containing the updated redundancy blocks (rdnc_block_1, rdnc_block_2, etc.)
- The destination buffer data segment is extended to include the updated redundancy blocks

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

- original_updated_and_rdnc_blocks data length should be $(3+3+4=10)\times64\text{KB} = 640\text{KB}$
- The available memory for writing in updated_rdnc_blocks should be at least $4\times64\text{KB} = 256\text{KB}$
The operation is not atomic

Once the task has been submitted, the source and destination buffers should not be read from/written to

Source and destination buffers must not overlap

Other limitations described in DOCA Core Task

Recover Task

This task executes recovers data blocks for, using given available original data blocks and redundancy blocks and a given coding matrix.

Task Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>API to Set the Configuration</th>
<th>API to Query Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable the task</td>
<td>doca_ec_task_recover_set_conf</td>
<td>doca_ec_cap_task_recover_is_supported</td>
</tr>
<tr>
<td>Maximum block size</td>
<td>–</td>
<td>doca_ec_cap_get_max_block_size</td>
</tr>
<tr>
<td>Maximum buffer list length</td>
<td>–</td>
<td>doca_ec_cap_get_max_buf_list_len</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>recover matrix</td>
<td>A coding matrix create by doca_ec_matrix_create() or doca_ec_matrix_create_from_raw()</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| available blocks | A source buffer with data, holding a sequence containing available data blocks and redundancy blocks (data_block_a, data_block_b, data_block_c, ..., rdnc_block_x, rdnc_block_y, etc.) | - The total number of blocks given should be equal to the number of original data blocks  
- The data length of available_blocks should be a multiplication of the block size  
- The data length should also be aligned to 64B and with a minimum size of 64B |
| recovered data blocks | A destination buffer for the recovered data blocks. The sequence containing the recovered data blocks (data_block_i, data_block_j, etc.) is written to it upon successful completion of the task | - The data is written to the tail segment extending the data segment  
- The minimal available memory in recovered_data_blocks should be the number of missing data blocks * the block size, aligned to 64B and, in any case, at least 64B. |

**Note**

Using a recover task matrix, based on an original 10x4 coding matrix (i.e., 10 original blocks, 4 redundancy blocks), and a block size of 64KB:

- 10 available blocks should be given in total (e.g., 7 data blocks and 3 redundancy blocks)
- available_blocks data length should be 10x64KB = 640KB
- The available memory for writing in recovered_data_blocks should be at least 3x64KB = 192KB
Task Output

Common output as described in DOCA Core Task.

Task Completion Success

After the task is completed successfully the data is transformed to destination.

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 buffers should not be read from/written to
- Source and destination must not overlap
- The amount of blocks that can be recovered are limited to the number of redundancy blocks created
- Other limitations are described in DOCA Core Task
DOCA Erasure Coding Samples

This section provides DOCA Erasure Coding sample implementation on top of the BlueField-3 DPU (and higher).

Info

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

Sample Prerequisites

N/A

Running the Sample

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:

   cd /opt/mellanox/doca/samples/doca_erasure_coding/<sample_name>
   meson /tmp/build
   ninja -C /tmp/build
The binary `doca_<sample_name>` is created under `/tmp/build/.

3. Sample (e.g., `doca_erasure_coding_recover`) usage:

Usage: `doca_erasure_coding_recover [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:**
- `-p, --pci-addr` DOCA device PCI device address - default: 03:00.0
- `-i, --input` Input file/folder to ec - default: self
- `-o, --output` Output file/folder to ec - default: /tmp
- `-b, --both` Do both (encode & decode) - default: false
- `-x, --matrix` Matrix - {cauchy, vandermonde} - default: cauchy
- `-t, --data` Data block count - default: 2
- `-r, --rdnc` Redundancy block count - default: 2
- `-d, --delete-index` Indices of data blocks to delete; comma-separated (i.e., 0,3,4) - default: 0

**Note**

Current BlueField-3 limitations:

- Data block count range – 1-128
- Redundancy block count – 1-32
- Block size – 64B-128MB
4. For additional information per sample, use the `-h` option:

```
/tmp/build/doca_<sample_name> -h
```

**Samples**

**Erasure Coding Recover**

This sample illustrates how to use DOCA Erasure Coding (EC) library to encode and decode a file block (and entire file).

The sample logic includes 3 steps:

1. Encoding – create redundancy.
2. Deleting – simulating disaster.
3. Decoding – recovering data.

The encode logic includes:

1. Locating a DOCA device.
2. Initializing the required DOCA Core structures, such as the progress engine (PE), memory maps, and buffer inventory.
3. Reading source original data file and splitting it to a specified number of blocks, `<data block count>`, specified for the sample to the output directory.
4. Populating two DOCA memory maps with a memory range, one for the source data and one for the result.
5. Allocating buffers from DOCA buffer inventory for each memory range.
6. Creating an EC object.
7. Connecting the EC context to the PE.
8. Setting a state change callback function for the PE, with the following logic:

   - Printing a log with every state change
   - Indicating that the user may stop progress the PE once it is back in idle state

9. Setting the configuration to the EC create task, including setting callback functions as follows:

   - Successful completion callback:
     1. Writing the resulting redundancy blocks to the output directory (count is specified by `<redundancy block count>`).
     2. Freeing the task.
     3. Saving the result of the task and the callback. If there was an error in step a., the relevant error value is saved.
     4. Stopping the context.

   - Failed completion callback:
     1. Saving the result of the task and the callback.
     2. Freeing the task.
     3. Stopping the context.

10. Creating EC encoding matrix by the matrix type specified to the sample.

11. Allocating and submitting an EC create task.

12. Progressing the PE until the context returns to idle state, either as a result of a successful run in which all tasks have been successfully completed, or as a result of a fatal error.

13. Destroying all EC and DOCA Core structures.

The delete logic includes:

   1. Deleting the block files specified with `<indices of data blocks to delete>`.
The decode logic includes:

1. Locating a DOCA device.

2. Initializing the required DOCA Core structures, such as the PE, memory maps, and buffer inventory.

3. Reading the output directory (source remaining data) and determining the block size and which blocks are missing (needing recovery).

4. Populating two DOCA memory maps with a memory range, one for the source data and one for the result.

5. Allocating buffers from DOCA buffer inventory for each memory range.

6. Creating an EC object.

7. Connecting the EC context to the PE.

8. Setting a state change callback function for the PE, with the following logic:
   - Printing a log with every state change
   - Indicating that the user may stop progress the PE once it is back in idle state

9. Setting the configuration to the EC recover task, including setting callback functions as following:
   - Successful completion callback:
     1. Writing the resulting recovered blocks to the output directory.
     2. Writing the recovered file to the output path.
     3. Freeing the task.
     4. Saving the result of the task and the callback. If there was an error in step a., the relevant error value is saved.
     5. Stopping the context.
   - Failed completion callback:
1. Saving the result of the task and the callback.

2. Freeing the task.

3. Stopping the context.

10. Creating EC encoding matrix by the matrix type specified to the sample.

11. Creating EC decoding matrix, with `doca_ec_matrix_create_recover()`, using the encoding matrix.

12. Allocating and submitting an EC recover task.

13. Progressing the PE until the context returns to idle state, either as a result of a successful run in which all tasks have been successfully completed, or as a result of a fatal error.

14. Destroying all DOCA EC and DOCA Core structures.

References:

- `/opt/mellanox/doca/samples/doca_erasure_coding/doca_erasure_coding_recover/erasure_coding_recover`

- `/opt/mellanox/doca/samples/doca_erasure_coding/doca_erasure_coding_recover/erasure_coding_recover`