Build Arm NN custom backend plugins

Version 21.08

Tutorial

Non-Confidential

Issue 01

102274_2108_01_en

Build Arm NN custom backend plugins

Tutorial

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	31 October 2019	Non-Confidential	First release
0100-02	25 August 2020	Non-Confidential	Updates the description of the Arm NN branch in the Before you begin section and new version of the ArmNNPluginFramework.zip file.
2108-01	6 September 2021	Non-Confidential	Removes Boost library dependency.

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web address

developer.arm.com

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1 Introduction

1.1 Conventions

The following subsections describe conventions used in Arm documents.

Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm® Glossary for more information: developer.arm.com/glossary.

Typographic conventions

Arm documentation uses typographical conventions to convey specific meaning.

Convention	Use
italic	Introduces special terminology, denotes cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.
`monospace underline`	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.
Caution	This represents a recommendation which, if not followed, might lead to system failure or damage.
Warning	This represents a requirement for the system that, if not followed, might result in system failure or damage.
Danger	This represents a requirement for the system that, if not followed, will result in system failure or damage.

Convention	Use
Note	This represents an important piece of information that needs your attention.
Tip	This represents a useful tip that might make it easier, better or faster to perform a task.
Remember	This is a reminder of something important that relates to the information you are reading.

1.2 Additional reading

This document contains information that is specific to this product. See the following documents for other relevant information:

Table 1-2: Arm publications

Document Name	Document ID	Licensee only
None	-	-

1.3 Feedback

Arm welcomes feedback on this product and its documentation.

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to errata@arm.com. Give:

- The title Build Arm NN custom backend plugins Tutorial.
- The number 102274_2108_01_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

1.4 Other information

See the Arm website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support
- Arm® Glossary.

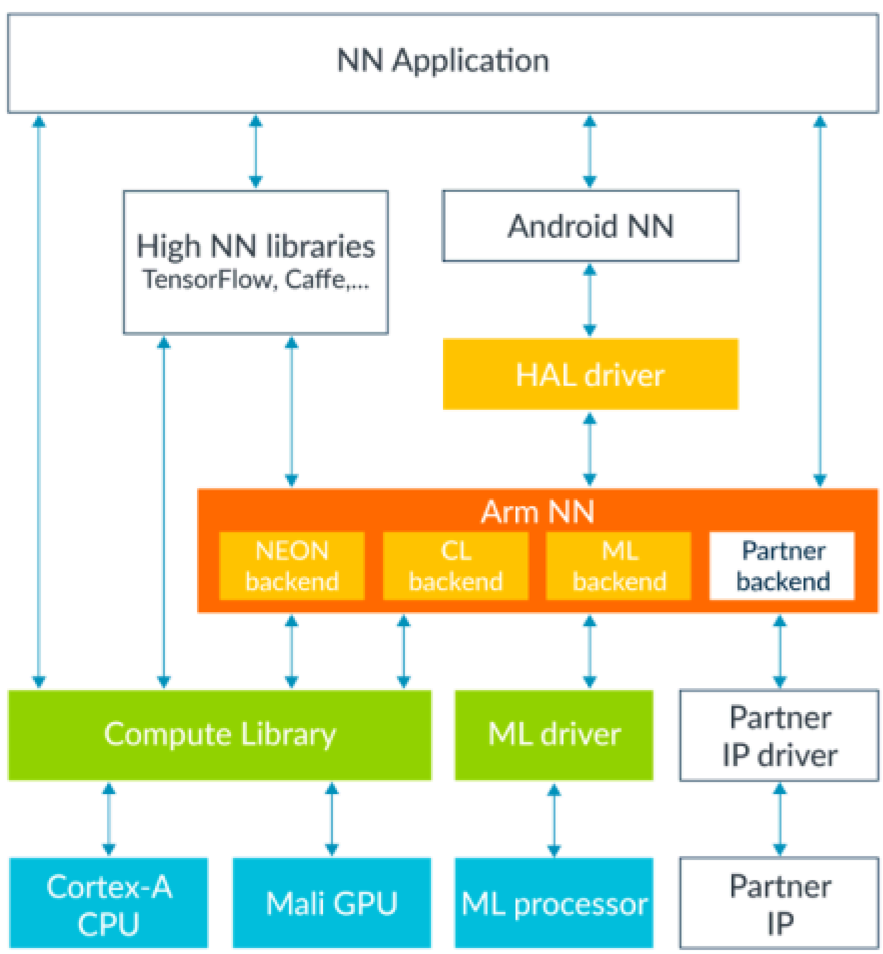
2 Overview

Arm NN is an inference middleware for CPUs, GPUs, and NPUs. Arm NN bridges the gap between existing NN frameworks and the underlying IP. Arm NN enables efficient translation of existing neural network frameworks, like TensorFlow Lite and ONNX. Arm NN allows these neural networks to run efficiently, without modification, across Arm Cortex-A CPUs, Arm Mali GPUs, and the Arm Machine Learning NPU processor.

Arm NN provides backends to allow workloads to run on Cortex-A CPUs, Mali-GPUs, and Arm ML processors.

Arm NN also lets you write your own custom backends to interface with third-party devices, as shown in the following diagram:

Figure 2-1: Flow of data to and from neural network application



This guide shows you how to write a custom backend for Arm NN, providing an example custom backend to illustrate the process. First, the guide takes you through the steps that are required to compile the custom plugin with Arm NN. Next, the guide explains how to run the tests to check that the plugin is working correctly. Finally, the guide explores the custom backend and shows how to write your own plugin.

2.1 Before you begin

You must have specific tools and software to follow the steps in this guide.

This guide assumes that you have a working Arm NN installation that is configured for TensorFlow. If not, refer to the instructions in Configure the Arm NN SDK build environment.

Arm recommends that you use the latest release branch of the Arm NN repository. Other branches cannot be guaranteed to work with the example backend.

This guide only requires the tools that are identified in Configure the Arm NN SDK build environment guide.

Note

The 21.08 release of Arm NN removes the Boost library dependency. You do not require the Boost library with Arm NN 21.08 and newer releases.

---

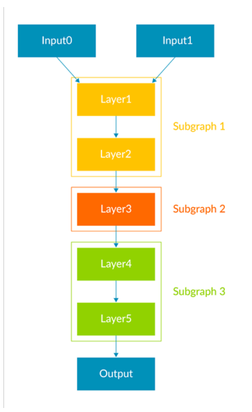
3 What is an Arm NN backend?

The Arm NN backend is an abstraction that maps the layers of a network graph to the hardware that is responsible for executing those layers. Arm NN provides ready-made backends to allow workloads to run on Cortex-A CPUs, Mali GPUs, and Arm ML processors. Arm NN also provides an interface so that you can write your own custom backends to interface with third-party devices.

Backends support one or more layers from the network graph, creating backend-specific workloads for the layers that they support, and then executing those workloads.

Each backend identifies the layers that it can process. The Arm NN then divides the original graph into several subgraphs to be assigned to the different backends. For example, in the following diagram, Arm NN divides the graph into three subgraphs. Arm NN does this by selecting the largest contiguous set of layers that can be processed by a single backend.

Figure 3-1: Arm NN subgraph and layers



When we look at this diagram, we can see that:

- Layers 1 and 2 can be executed by the same backend.
- Layers 4 and 5 can be executed by the same backend. This may be the same backend as Layers 1 and 2, or it may be a different backend.
- Layer 3 requires a different backend from all the other layers.

All backends must:

- implement the `IBackendInternal` interface
- identify themselves with a string that must be unique across all of the backends
- register themselves with `BackendRegistry`, so that Arm NN knows about them
- implement the `ILayerSupport` interface for the layers the backend intends to support
- implement the `IWorkloadFactory` interface, so that Arm NN can execute layers on the backend

You can learn more about backends in Write your own Arm NN backend plugin.

4 The example plugin

The example backend implements a simple custom plugin to help show how you can write your own custom plugins. The example backend simulates optimizing addition layers by substituting them with a pre-compiled layer. This pre-compiled layer includes a pre-compiled object that represents an optimized alternative to the addition layer in Arm NN. This pre-compiled object is an instance of a `CustomPreCompiledObject`.

4.1 Build the example plugin

Follow these steps to integrate the example custom plugin with your existing Arm NN build.

Procedure

1. Download the ArmNNPluginFramework.zip file containing the example plugin to a temporary location, for example /tmp.
2. Extract the contents of the zip file:
```
cd /tmp
unzip ArmNNPluginFramework.zip
```
3. Copy the example plugin to the src/backends folder in your Arm NN installation:
```
cp -r /tmp/custom <armnn_install_dir>/armnn/src/backends/
```
4. Re-run CMake to produce the new makefiles that are needed to build the example plugin:
```
cd <armnn_install_dir>/armnn/build
cmake ..
```
5. Compile using the make command:
```
make -j32
```
6. Run all the Arm NN unit tests, including those supplied with the example plugin:
```
cd <armnn_install_dir>/armnn/build
./UnitTests
```

   The output should be:
```
Running 1204 test cases...
*** No errors detected
```

5 How the custom backend works

The unit tests that are included in the example custom plugin illustrate how the custom plugin works.

To see how it works, we will look at the `AdditionToPreCompiledTest()` example in the `CustomEndToEndTests.cpp` file.

The `AdditionToPreCompiledTest()` function:

- creates an initial graph with an addition layer, performing element-wise addition of vectors
- optimizes the graph. In the example, optimizing the graph substitutes the addition layer with a precompiled layer
- runs the inference on the optimized graph with some test values
- checks that the results are correct

Follow these steps to understand how the custom backend works.

1. Create an empty model object:

```
INetworkPtr net(INetwork::Create());
```

2. Add layers to the model:

```
IConnectableLayer* input1 = net->AddInputLayer(0);
IConnectableLayer* input2 = net->AddInputLayer(1);
IConnectableLayer* add    = net->AddAdditionLayer();
IConnectableLayer* output = net->AddOutputLayer(0);
```

3. Create the required connections between the layers:

```
input1->GetOutputSlot(0).Connect(add->GetInputSlot(0));
input2->GetOutputSlot(0).Connect(add->GetInputSlot(1));
```

```
add->GetOutputSlot(0).Connect(output->GetInputSlot(0));
```

The following diagram shows the connections between the layers.

Figure 5-1: Arm NN Plugin Framework diagram showing the connections between layers

4. Set the tensor information for each of the outputs:

```
TensorInfo tensorInfo(TensorShape({3, 4}), DataType::Float32);
input1->GetOutputSlot(0).SetTensorInfo(tensorInfo);
input2->GetOutputSlot(0).SetTensorInfo(tensorInfo);
add->GetOutputSlot(0).SetTensorInfo(tensorInfo);
```

5. Optimize the completed network using the `Optimize()` function:

```
IOptimizedNetworkPtr optimizedNet = Optimize(*net, defaultBackends,
        runtime->GetDeviceSpec());
```

The `Optimize()` function has the following specification:

```
IOptimizedNetworkPtr Optimize(INetwork, {BackendId,... }, IDeviceSpec,
                              OptimizerOptions, errMessages)
```

The `Optimize()` function:

- performs basic validation of the input network

- modifies the graph for correctness by:
  - inserting copy layers between backends
  - inserting FP32/FP16 conversion layers if necessary (specified in OptimizerOptions)
  - adding debug layers, if necessary (specified in OptimizerOptions)
- performs backend-independent optimizations by:
  - removing redundant operations
  - optimizing all permutes and reshapes where possible
- decides which backend to assign to each layer by:
  - using the Is<x>LayerSupported() function in the ILayerSupport interface to identify the preferred backend
- runs backend-specific optimizations by:
  - for each selected backend, extracting the subgraphs that can be executed on that backend
  - for each subgraph, calling OptimizeSubGraph() on the selected backend

6. Create and configure the runtime object:

```
IRuntime::CreationOptions options;
IRuntimePtr runtime(IRuntime::Create(options));
```

7. Load the optimized network:

```
NetworkId networkId;
runtime->LoadNetwork(networkId, std::move(optimizedNet));
```

The LoadNetwork() function:

- creates a LoadedNetwork object and adds it to the runtime
- creates a list of workloads, one per layer, using the backend's IWorkloadFactory object
- returns a network identifier, networkId, to use later for running the optimized network

8. Create sample input and output data structures:

```
std::vector <float> input1Data
{
    1.f, 2.f, 3.f, 4.f, 5.f, 6.f, 7.f, 8.f, 9.f, 10.f, 11.f, 12.f
};
std::vector<float> input2Data
{
    100.f, 200.f, 300.f, 400.f, 500.f, 600.f, 700.f, 800.f, 900.f, 1000.f,
 1100.f, 1200.f
};
std::vector<float> outputData(12);

InputTensors inputTensors
{
    { 0, ConstTensor(runtime->GetInputTensorInfo(networkId, 0), input1Data.da\
ta()) },
    { 1, ConstTensor(runtime->GetInputTensorInfo(networkId, 0), input2Data.da\
ta()) }
};
```

```
OutputTensors outputTensors
{
    { 0, Tensor(runtime->GetOutputTensorInfo(networkId, 0), outputData.data()) }
};
```

9. Run the inference:

```
runtime->EnqueueWorkload(networkId, inputTensors, outputTensors);
```

The `networkId` is the one that is returned by the earlier call to `LoadNetwork()`. The `EnqueueWorkload()` function executes all workloads sequentially on the assigned backends and places the result in the output tensor buffers.

6 Write your own Arm NN backend plugin

The example custom plugin provides a useful template for writing your own backend. We will look at the different things that you need to do when writing your own backend. We will use the code from the example plugin to illustrate the process.

6.1 Build system integration

Before you can build your custom plugin, you will need to integrate the plugin with the Arm NN build system. Arm NN uses the CMake build management system. Follow the steps in this section to write your own Arm NN backend plugin.

Procedure

1. Create a directory for your custom plugin in armnn/src/backends, for example custom:

```
mkdir <armnn_install_dir>/armnn/src/backends/custom
```

2. Create a `backend.cmake` file to specify what needs to be built. The `backend.cmake` file in the example plugin contains:

```
add_subdirectory(${PROJECT_SOURCE_DIR}/src/backends/custom)
list(APPEND armnnLibraries armnnCustomBackend)
list(APPEND armnnLibraries armnnCustomBackendWorkloads)
list(APPEND armnnUnitTestLibraries armnnCustomBackendUnitTests)
```

3. Create `CMakeLists.txt` files in each directory to specify the rules to build the new build targets. For example, here is the `CMakeLists.txt` file in the top-level `custom` directory:

```
list(APPEND armnnCustomBackend_sources
    CustomBackend.cpp
    CustomBackend.hpp
    CustomBackendUtils.cpp
    CustomBackendUtils.hpp
    CustomLayerSupport.cpp
    CustomLayerSupport.hpp
    CustomPreCompiledObject.cpp
    CustomPreCompiledObject.hpp
    CustomWorkloadFactory.cpp
    CustomWorkloadFactory.hpp
)

add_library(armnnCustomBackend OBJECT ${armnnCustomBackend_sources})
target_include_directories(armnnCustomBackend PRIVATE ${PROJECT_SOURCE_DIR}/src/
armnn)
target_include_directories(armnnCustomBackend PRIVATE ${PROJECT_SOURCE_DIR}/src/
armnnUtils)
target_include_directories(armnnCustomBackend PRIVATE ${PROJECT_SOURCE_DIR}/src/
backends)

add_subdirectory(workloads)

if(BUILD_UNIT_TESTS)
    add_subdirectory(test)
endif()
```

4. Create a `backend.mk` file to specify the source files. This file is used for Android builds:

```
BACKEND_SOURCES := \
        CustomBackend.cpp \
```

```
        CustomBackendUtils.cpp \
        CustomLayerSupport.cpp \
        CustomPreCompiledObject.cpp \
        CustomWorkloadFactory.cpp \
        workloads/CustomAdditionWorkload.cpp \
        workloads/CustomPreCompiledWorkload.cpp

BACKEND_TEST_SOURCES := \
        test/CustomCreateWorkloadTests.cpp \
        test/CustomEndToEndTests.cpp
```

6.2 Identify and register your plugin

All backends must identify themselves with a unique `BackendId`.

Here is the code in `CustomBackend.cpp` that provides the unique ID.

```
const BackendId& CustomBackend::GetIdStatic()
{
    static const BackendId s_Id{"Custom"};
    return s_Id;
}
```

Plugins must also register with the `BackendRegistry`. A helper structure, `BackendRegistry::StaticRegistryInitializer`, is provided to register the backend:

```
static BackendRegistry::StaticRegistryInitializer g_RegisterHelper
{
    BackendRegistryInstance(),
    CustomBackend::GetIdStatic(),
    []()
    {
        return IBackendInternalUniquePtr(new CustomBackend());
    }
};
```

6.3 Implement the IBackendInternal interface

All backends need to implement the IBackendInternal interface.

Here are the interface functions to implement:

- `IMemoryManagerUniquePtr CreateMemoryManager()`
- `IWorkloadFactoryPtr CreateWorkloadFactory(IMemoryManagerSharedPtr)`
    - The returned `IWorkloadFactory` object is used to create the workload layer computation units.
- `IBackendContextPtr CreateBackendContext(IRuntime::CreationOptions)`
- `ILayerSupportSharedPtr GetLayerSupport()`
    - During optimization, Arm NN needs to decide which layers are supported by the backend.
    - `IsLayer<x>Supported()` functions indicate whether the backend supports the specified layer.

- `OptimizationViews OptimizeSubGraph(SubGraph)`
    - The subgraph to optimize is passed as the input to this function.
    - The function returns an object containing a list of subgraph substitutions, a list of failed subgraph optimizations, and a list of untouched subgraphs.

The following sections look at each of these functions in more detail, as seen in `CustomBackend.cpp`.

6.4 Memory management using the CreateMemoryManager function

The purpose of memory management is to minimize memory usage by allocating memory just before it is needed, and releasing it when the memory is no longer required.

All backends must support the `IBackendInternal` interface `CreateMemoryManager()` method, which returns a unique pointer to an `IMemoryManager` object:

```
IBackendInternal::IMemoryManagerUniquePtr MyBackend::CreateMemoryManager() const
{
    return std::make_unique<MyMemoryManager>(...);
}
```

In this example, `MyMemoryManager` is a class that is derived from `IBackendInternal::IMemoryManager`.

A backend that does not support a memory manager, such as the example plugin, should return an empty pointer, as you can see here:

```
IBackendInternal::IMemoryManagerUniquePtr MyBackend::CreateMemoryManager() const
{
    return IBackendInternal::IMemoryManagerUniquePtr{};
}
```

The `IMemoryManager` interface defines two pure virtual methods that are implemented by the derived class for the backend:

- `virtual void Acquire() = 0;`
    - `Acquire()` is called by the `LoadedNetwork` before the model is executed.
    - The backend memory manager should allocate any memory that it needs for running the inference.

- `virtual void Release() = 0;`
  - `Release()` is called by the `LoadedNetwork`, in its destructor, after the model is executed.
  - The backend memory manager should free any memory that it previously allocated.

The backend memory manager uses internal memory management to further optimize memory usage.

6.5 Workload factories using the CreateWorkloadFactory function

Each layer is executed using a workload. A workload is used to enqueue a layer for computation.

Each workload that is created by a `WorkloadFactory`, creates workloads that are specific to each layer. This means that each backend needs its own `WorkloadFactory`.

All workloads must:

- implement the `IWorkload` interface
- implement the `Create<x>` methods to execute the operator on the backend hardware by:
  - reading the input tensors
  - writing the result to the output tensors

You can see the example code in `CustomWorkloadFactory.cpp`.

6.6 Backend context using the CreateBackendContext function

The `IBackendContext` interface defines virtual methods that are implemented by the derived class for the backend.

The following code shows the `IBackendContext` interface defining virtual methods:

```
IBackendInternal::IBackendContextPtr CustomBackend::CreateBackendContext(const IRun\
time::CreationOptions&) const

{

    return IBackendContextPtr{};

}
```

Here you can see how these virtual methods are defined in armnn/src/backends/backendsCommon/IBackendContext.hpp:

```
class IBackendContext
```

```
{
protected:
    IBackendContext(const IRuntime::CreationOptions&) {}
public:
    // Before and after Load network events
    virtual bool BeforeLoadNetwork(NetworkId networkId) = 0;
    virtual bool AfterLoadNetwork(NetworkId networkId) = 0;

    // Before and after Unload network events
    virtual bool BeforeUnloadNetwork(NetworkId networkId) = 0;
    virtual bool AfterUnloadNetwork(NetworkId networkId) = 0;

    virtual ~IBackendContext() {}
};
```

The `IBackendContext` interface includes some methods that provide callback-like functionality. These methods are called by Arm NN before and after loading or unloading a network respectively. These methods allow the user to run any code, for example to clear a cache or synch threads, triggered by a specific load or unload network event.

6.7 Deciding which backends to assign to each layer using the GetLayerSupport function

During optimization, Arm NN must decide which layers are supported by the backend.

The `IsLayer<x>Supported()` functions indicate whether the backend supports the specified layer. For example:

```
bool CustomLayerSupport::IsAdditionSupported(const TensorInfo& input0,
                                             const TensorInfo& input1,
                                             const TensorInfo& output,
                                             Optional<std::string&> reasonIfUnsup\
ported) const
{
    ignore_unused(input1);
    ignore_unused(output);
    return IsDataTypeSupported(input0.GetDataType(), reasonIfUnsupported);
}
```

6.8 Optimization using the OptimizeSubGraph function

The optimizer calls `OptimizeSubGraph()` on the selected backend, for each subgraph.

From the `IBackendInternal` interface:

```
OptimizationViews OptimizeSubGraph(const SubGraph& subGraph) const = 0;

class OptimizationViews
{
  ...
  Substitutions SuccesfulOptimizations; // Proposed substitutions from successful
 optimizations
  Subgraphs FailedOptimizations; // Subgraphs from the original subgraph which can\
not be supported
  Subgraphs UntouchedSubgraphs;  // Subgraphs from the original subgraph which re\
main unmodified
};

struct SubstitutionPair
{
  // Subgraph of Layers from the original graph which should be replaced
  SubgraphView SubstitutableSubgraph;

  // A subgraph of new layers which will replace layers in m_SubstitutableSubgraph
  SubgraphView ReplacementSubgraph;
};
```

Example optimizations might include:

- merging layers, for more efficient execution
- adding permute layers to modify the data layout for execution on the backend

The `OptimizeSubGraph()` function does the following:

- If no optimization was attempted for part of the input subgraph, the optimization function adds it to the list of untouched subgraphs.
- If part of the input subgraph cannot be supported by the backend, the optimization function adds it to the list of failed optimizations. Arm NN tries to re-assign each failed subgraph to other backends, if they are available.
- If part of the input subgraph can be optimized, the optimization function creates a substitution pair. The substitutable subgraph in the original graph is replaced with the corresponding replacement subgraph.

7 Next steps

You can learn more about developing custom backends in the GitHub README.md file.

You can learn more about Arm NN.