Ethos N NPU/GPU Zero Copy Fallback Design Document

Abstract

Zero Copy fallback from NPU to GPU is potentially a way to provide the ability of a system with an EthosN NPU and a Mali GPU to execute an ML Network where some of the operators are not supported on the EthosN but can be run as shader code on the GPU in a performant manner by eliminating the overhead of copying the contents of memory from one component to the other by means of memory import support in the NPU and GPU driver software.

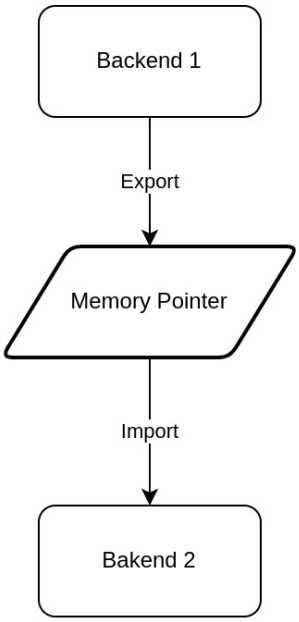
NOTE: that part of the investigation should be to establish if such a zero copy fallback will actually deliver the performance required, remember the GPU executes asynchronously and so the latency in the call might be excessive and prevent the required level of performance being attained especially if the call comes in the middle rather than the end of a network.

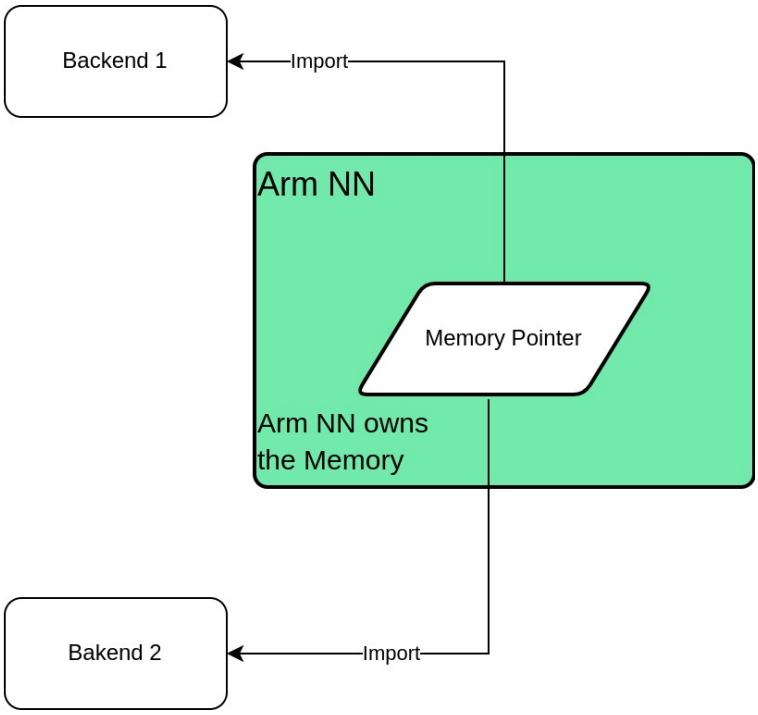
Overview

Currently, Arm NN only support GPU import of input and output. In case of fallback between backends, memory copy will be used. In order to support zero copy on NPU/GPU fallback, we need to add functionality to allow import between backends. Here is the table showing supports for each backend - Npu (EthosN), Gpu (Cl), and Cpu (Neon) for reference. Gpu Tensor cannot have map/unmap and import at the same time.

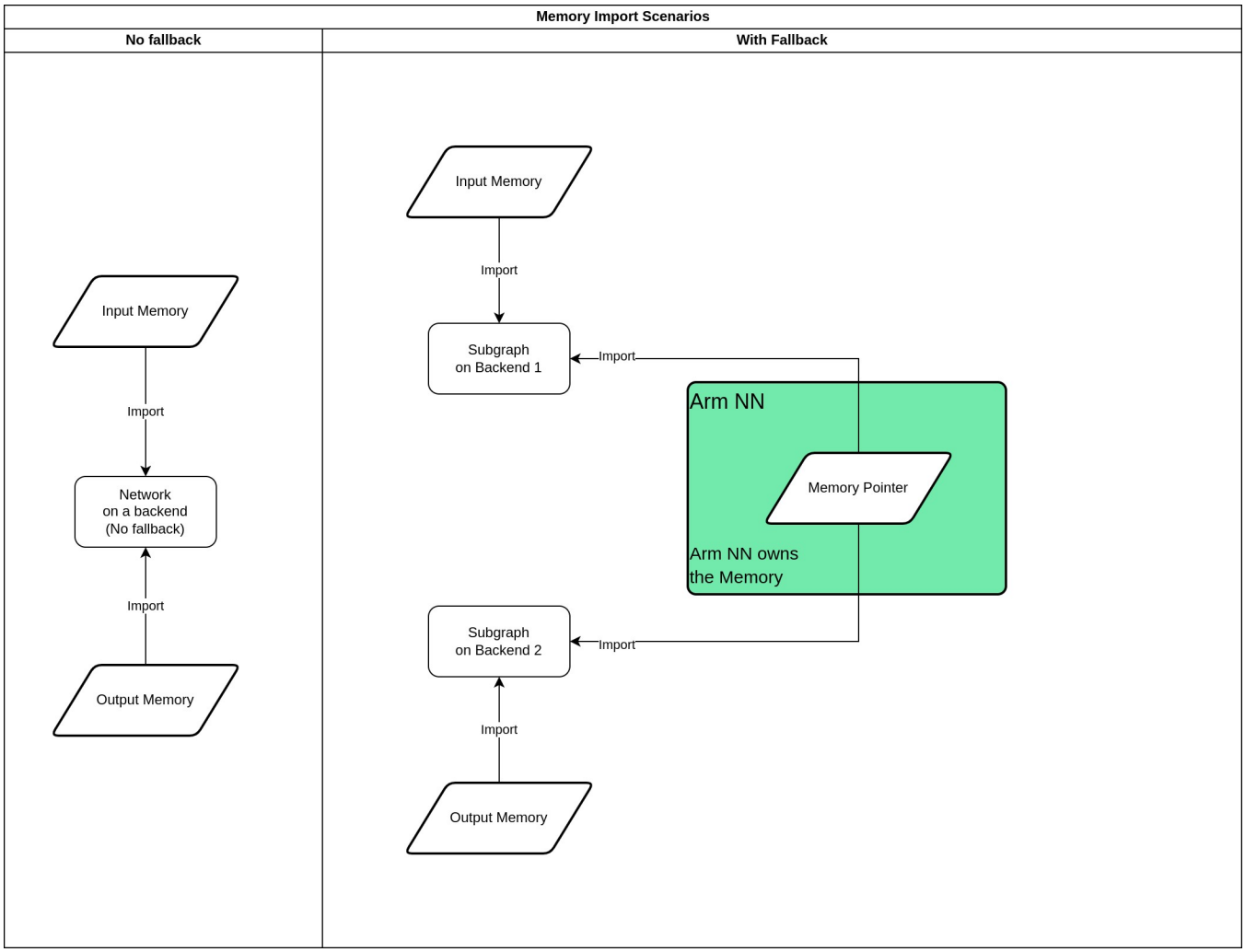
Support	EthosNImportTensorHandle	ClTensorHandle	ClImportTensorHandle	NeonTensorHandle (for reference)
Map/Unmap	✅	✅	❌	✅
Import	✅ (DmaBuf)	❌	✅ (Malloc, DmaBuf)	✅ (Malloc)
Export	❌	❌	❌	❌

The ideal solution to do zero copy in case of fallback between backends is that a source backend export a memory and import to a destination backend. However, no backend has support for memory export. Arm NN needs to handle the memory for each backend.





The process to execute of each subgraph will be similar to a network execution, where Arm NN enqueue output of the first backend and enqueue input of the second backend. We also need to ensure that all operators in backends 1 get execution before the the backends 2 start the execution.



Fallback memory manager could be similar to how we create memory for the import tests using dma_buf_test_exporter.ko

Currently our DmaBuf tests are tied to a version of Mali and the dma_buf_test_exporter.ko which only works on Debian 9 and only on a Hikey960. As these become more and more obsolete it may be necessary to update the tests to work on a newer board and with a newer OS. Doing so will require the following:

- Updated version of the Mali Driver which supports DMABuf (same version that we have in Odroids)
- Updated version of the dma_buf_test_exporter.ko which was compiled with that driver and is now loaded into the board being tested at runtime
- Boilerplate to make sure that version of the kernel is present before running the tests, and to load it if not
- Updates to the tests to make sure that any function calls still work, and update them to use the new kernel if not
- Updates to the existing CI jobs to check that the tests are being run on the new devices

Even though the scope of this design note is for zero copy (with DmaBuf) for NPU/GPU fallback, different scenarios need to be taken into account to prevent other failure cases. For example, when we enable memory import which will fallback on backends where import is not supported. The work we require for NPU/GPU fallback should be similar to CPU/GPU fallback (with malloc). During early development, working with CPU/GPU fallback might be easier and more familiar to us.

High Level Estimation

Total: 100 story points

Setup board for development + integrate to CI - 20

Fallback memory manager - 25

Memory Import workflow - 30

Performance Investigation - 25

Detailed Design

FallbackImportLayer

In order to import a memory to import it to the output of the subgraph of the source backend as well as import it to the input of the subgraph of the destination backend, we add a FallbackImportLayer (1 input, 1 output) to specify that it need to call Fallback Memory Manager to create a memory which aligns with both source and destination backends.

Note: BackendId of the FallbackImportLayer itself is destination backend (This is set when the FallbackImport layer is created). However, when the workload is created, it will created workload from source backend to do correct synchronisation from source backend so we have to also add the source backend to it.

Also add FallbackImportLayer in the list of LayerType.

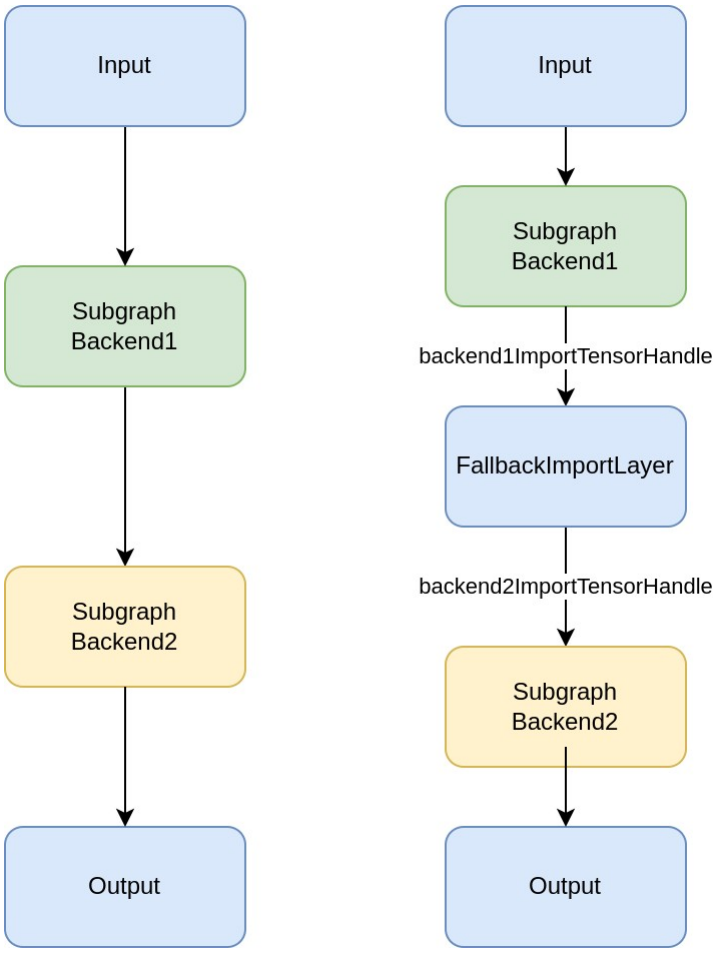
The actual memory creation will be done when we do EnqueueWorkload, by calling FallbackMemoryManager.

FallbackImportLayer

```
//
// Copyright © 2023 Arm Ltd and Contributors. All rights reserved.
// SPDX-License-Identifier: MIT
//
#pragma once

#include <Layer.hpp>

namespace armnn
{
  /// This layer represents a memory import in case of fallback
/// to import a memory to an output of the subgraph of the source backend
/// as well as import the same memory to an input of the subgraph of the destination backend.
class FallbackImportLayer : public Layer
{
public:
    /// Makes a workload for the FallbackImportLayer for synchronisation.
    /// @param [in] graph The graph where this layer can be found.
    /// @param [in] factory The workload factory which will create the workload.
    /// @return A pointer to the created workload, or nullptr if not created.
    virtual std::unique_ptr<IWorkload>CreateWorkload(const IWorkloadFactory& factory) const override;

    /// Create TensorHandles using destFactory.
    virtual void CreateTensorHandles(const TensorHandleFactoryRegistry& registry,
                                     const IWorkloadFactory& factory,
                                     const bool IsMemoryManaged = true);

    /// Creates a dynamically-allocated copy of this layer.
    /// @param [in] graph The graph into which this layer is being cloned.
    FallbackImportLayer* Clone(Graph& graph) const override;

    /// Check if the input tensor shape(s)
    /// will lead to a valid configuration of @ref FallbackImportLayer.
    void ValidateTensorShapesFromInputs() override;

    const BackendId& GetSrcBackendId() const { return m_SrcBackendId; }
    void SetSrcBackendId(const BackendId& id) override { m_SrcBackendId = id; }

        // Add getter settr for m_SrcTensorHandleFactory, m_DescTensorHandleFactory, m_MemSource

protected:
    /// Constructor to create a FallbackImportLayer.
    /// @param [in] name Optional name for the layer.
    FallbackImportLayer(const char* name);

    /// Default destructor
    ~FallbackImportLayer() = default;

private:
    BackendId m_SrcBackendId;

    ITensorHandleFactory m_SrcTensorHandleFactory;
    ITensorHandleFactory m_DescTensorHandleFactory;
    MemorySource m_MemSource;

  } // namespace
```

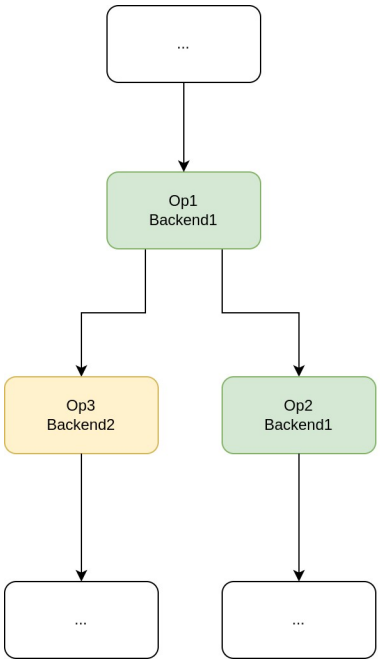
Simple Scenario

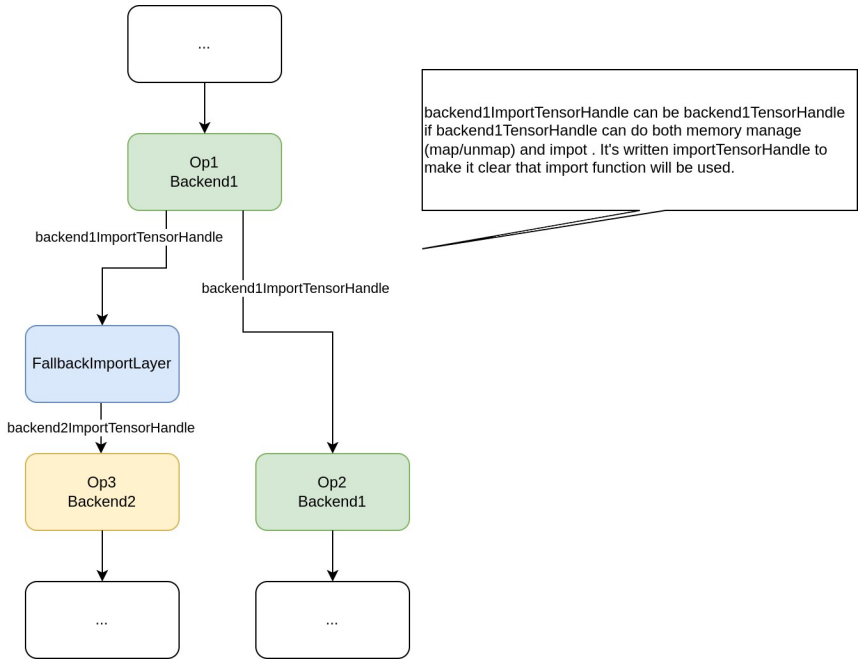
Note: ImportTensorHandle is used to specify that import function will be used and TensorHandle is used to specify that the memory managed (map /unmap) will be used. In case of backends that the TensorHandle can do both memory managed and import, ImportTensorHandle and TensorHandle can be the same, such as NeonTensorHandle, RefTensorHandle. But in case of Cl, there are ClImportTensorHandle and ClTensorHandle.



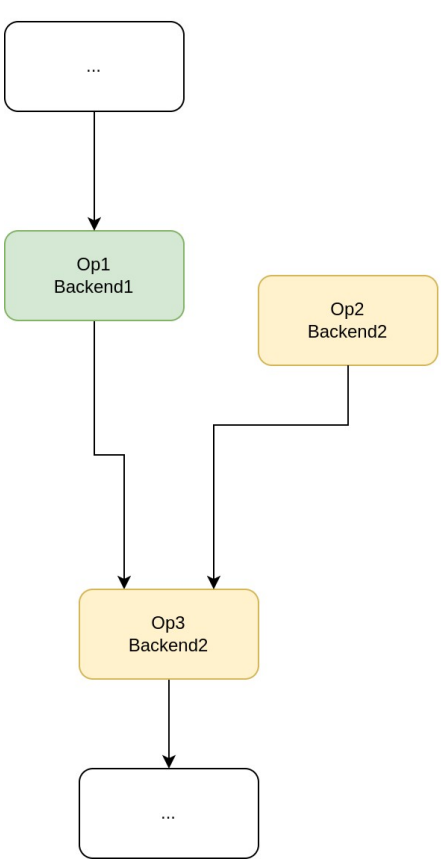
More complex scenarios

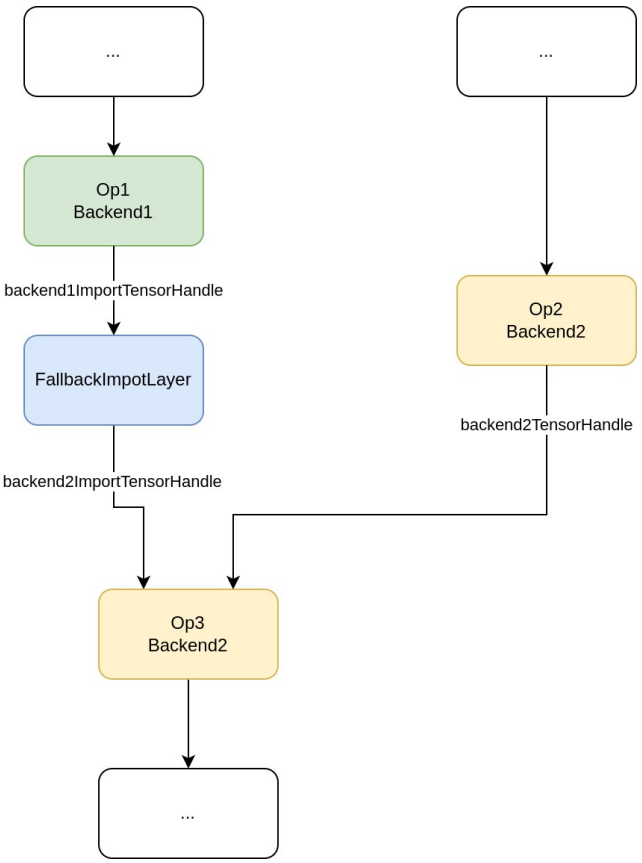
In more complex scenario where the last layer of a subgraph from source backend connects to multiple layers and one fallback to different backend.





Destination backend has multiple inputs.





FallbackImportWorkload

FallbackImportWorkload will be created to be sync after execution of the subgraph in case the source backend has asynchronous execution. This is currently especially for GPU case - ClFallbackImportWorkload.

In additional, before execution, FallbackImports have to be enqueued for import after EnqueueInput before EnqueueOutput and EnqueueWorkload. This is the actual work to create an aligned memory and import to the output of the subgraph of the source backend as well as import it to the input of the subgraph of the destination backend. Base FallbackImportWorkload do nothing as the memory import will be done during EnqueueFallbackImport.

ClFallbackImportWorkload

```
void ClFallbackImportWorkload::Execute() const
{
    ARMNN_SCOPED_PROFILING_EVENT_CL_NAME_GUID("ClFallbackImportWorkload_Execute");

    // Waits for all queued CL workloads to finish before executing the next backend, so that the destination
backend get the correct imported input values.
    try
    {
        arm_compute::CLScheduler::get().sync();
    }
    catch (const cl::Error&)
    {
        // Throw Exception
    }
}
```

When creating workload, FallbackImportWorkload is a special case to create the workload from source backend. LoadedNetwork::GetWorkloadFactory has to be modified to get source backend so that it create correct workload.

GetWorkloadFactory

```
const IWorkloadFactory& LoadedNetwork::GetWorkloadFactory(const Layer& layer) const
{
    const IWorkloadFactory* workloadFactory = nullptr;

        BackendId backendId = layer.GetBackendId();

    if (layer.GetType() == LayerType::FallbackImport)
    {
              // need to cast layer to FallbackImportLayer to be able to call GetSrcBackendId()
        backendId = layer.GetSrcBackendId();
    }

    auto it = m_WorkloadFactories.find(layer.GetBackendId());
    if (it ==  m_WorkloadFactories.end())
    {
        throw RuntimeException(fmt::format("No workload factory for {0} to be used for layer: {1}",
                                           layer.GetBackendId().Get(),
                                           layer.GetNameStr()),
                                           CHECK_LOCATION());
    }

    workloadFactory = it->second.get();

    ARMNN_ASSERT_MSG(workloadFactory, "No workload factory");

    return *workloadFactory;
}
```

Memory Import Workflow

Remove the capability FallbackImportDisabled to disable fallback import in ClImportTensorHandleFactory and add FallbackImportMemoryRequired instead, so that memory import is allowed in case of GPU fallback when EdgeStrategy is calculated.

capability

```
std::vector<Capability> ClImportTensorHandleFactory::GetCapabilities(const IConnectableLayer* layer,
                                                                     const IConnectableLayer* connectedLayer,
                                                                     CapabilityClass capabilityClass)
{
    IgnoreUnused(layer);
    IgnoreUnused(connectedLayer);
    td::vector<Capability> capabilities;
    if (capabilityClass == CapabilityClass::FallbackImportMemoryRequired)
    {
        Capability capability(CapabilityClass::FallbackImportMemoryRequired, true);
        capabilities.push_back(capability);
    }
    return capabilities;
}
```

In *Graph::AddCompatibilityLayers* where the EdgeStrategy is checked, add the FallbackImport layer to import subgraph output as well as subgraph input if the srcFactory requires fallback import memory (by checking the FallbackImportMemoryRequired Capability of srcFactory. Otherwise, ad MemImport layer as previously.

Also set srcBackends, srcTensorHandleFactories, memsource to the newly created FallbackImportlayer.

EdgeStrategy

```
    ForEachLayer([this, &backends, &registry, MayNeedCompatibilityLayer, IsCompatibilityStrategy](Layer*
srcLayer)
    {
        ARMNN_ASSERT(srcLayer);

        if (!MayNeedCompatibilityLayer(*srcLayer))
        {
            // The current layer does not need copy layers, move to the next one
            return;
        }

        const std::vector<OutputSlot>& srcOutputSlots = srcLayer->GetOutputSlots();
        for (unsigned int srcOutputIndex = 0; srcOutputIndex < srcOutputSlots.size(); srcOutputIndex++)
        {
            OutputSlot& srcOutputSlot = srcLayer->GetOutputSlot(srcOutputIndex);
            const std::vector<InputSlot*> srcConnections = srcOutputSlot.GetConnections();
            const std::vector<EdgeStrategy> srcEdgeStrategies = srcOutputSlot.GetEdgeStrategies();
            for (unsigned int srcConnectionIndex = 0; srcConnectionIndex < srcConnections.size();
srcConnectionIndex++)
            {
                InputSlot* dstInputSlot = srcConnections[srcConnectionIndex];
                ARMNN_ASSERT(dstInputSlot);

                EdgeStrategy strategy = srcEdgeStrategies[srcConnectionIndex];
                ARMNN_ASSERT_MSG(strategy != EdgeStrategy::Undefined,
                                 "Undefined memory strategy found while adding copy layers for
compatibility");

                const Layer& dstLayer = dstInputSlot->GetOwningLayer();
                if (MayNeedCompatibilityLayer(dstLayer) &&
                    IsCompatibilityStrategy(strategy))
                {
                    // A copy layer is needed in between the source and destination layers.
```

```
                    // Record the operation rather than attempting to modify the graph as we go.
                    // (invalidating iterators)
                    const std::string compLayerName = fmt::format("[ {} ({}) -> {} ({}) ]",
                                                                  srcLayer->GetName(),
                                                                  srcOutputIndex,
                                                                  dstLayer.GetName(),
                                                                  dstInputSlot->GetSlotIndex());
                    Layer* compLayer = nullptr;
                    if (strategy == EdgeStrategy::CopyToTarget)
                    {
                        compLayer = InsertNewLayer<MemCopyLayer>(*dstInputSlot, compLayerName.c_str());
                    }
                    //
                    else if (strategy == EdgeStrategy::ExportToTarget)
                    {
                        auto srcPref = srcOutputSlot.GetTensorHandleFactoryId();
                        auto srcFactory = registry.GetFactory(srcPref);
                        // If source TensorHandleFactory supports requires FallbackImportMemoryRequired, add
FallbackImportLayer
                        // Otherwise, add MemImportLayer
                        if (srcFactory->GetCapabilities(srcLayer, destLayer, FallbackImportMemoryRequired))
                        {
                            // Insert Fallback import layer to import subgraph output as well as subgraph
input
                            compLayer = InsertNewLayer<FallbackImportLayer>(*dstInputSlot, compLayerName.
c_str());
                                                        // TODO Set Srcbackends, srcTensorHandleFactories,
memsource to the newly created FallbackImportlayer.                         }
                        else
                        {   compLayer = InsertNewLayer<MemImportLayer>(*dstInputSlot, compLayerName.c_str());

                        }
                    }
                    else
                    {
                        throw Exception("Invalid edge strategy found.");
                    }

                    compLayer->SetBackendId(dstLayer.GetBackendId());

                    OutputSlot& compOutputSlot = compLayer->GetOutputSlot(0);
                    auto backendIt = backends.find(dstLayer.GetBackendId());
                    if (backendIt != backends.end() &&
                        backendIt->second &&
                        backendIt->second->SupportsTensorAllocatorAPI())
                    {
                        auto backend = backendIt->second.get();
                        auto tensorHandleFactoryIds = backend->GetHandleFactoryPreferences();
                        bool found = false;

                        for (auto preference : tensorHandleFactoryIds)
                        {
                            auto factory = registry.GetFactory(preference);
                            if (factory)
                            {
                                auto srcPref = srcOutputSlot.GetTensorHandleFactoryId();
                                auto srcFactory = registry.GetFactory(srcPref);

                                if (srcFactory)
                                {
                                    bool canExportImport =
                                        (factory->GetImportFlags() & srcFactory->GetExportFlags()) != 0;

                                    if ((strategy == EdgeStrategy::CopyToTarget && factory-
>SupportsMapUnmap()) ||
                                        (strategy == EdgeStrategy::ExportToTarget && canExportImport))
                                    {
```

```
                                    compOutputSlot.SetTensorHandleFactory(preference);
                                    found = true;
                                    break;
                                }
                            }
                        }
                    }

                    if (!found)
                    {
                        compOutputSlot.SetTensorHandleFactory(ITensorHandleFactory::LegacyFactoryId);
                    }
                }
                else
                {
                    compOutputSlot.SetTensorHandleFactory(ITensorHandleFactory::LegacyFactoryId);
                }

                // The output strategy of a compatibility layer is always DirectCompatibility.
                compOutputSlot.SetEdgeStrategy(0, EdgeStrategy::DirectCompatibility);

                // Recalculate the connection index on the previous layer as we have just inserted into
it.
                const std::vector<InputSlot*>& newSourceConnections = srcOutputSlot.GetConnections();
                auto newSrcConnectionIndex = std::distance(newSourceConnections.begin(),
                                                           std::find(newSourceConnections.begin(),
                                                                     newSourceConnections.end(),
                                                                     &compLayer->GetInputSlot(0)));

                // The input strategy of a compatibility layer is always DirectCompatibilty.
                srcOutputSlot.SetEdgeStrategy(armnn::numeric_cast<unsigned int>(newSrcConnectionIndex),
                                              EdgeStrategy::DirectCompatibility);
            }
        }
    }
});
```

When a network is loaded (LoadedNetwork::LoadedNetwork), TensorHandle will be created for each layer. For MemImport layer and FallbacyImportLayer, a TensorHandle will be created with import enabled. If the layer is before the output layer (and import is enable) or FallbackImport layer, create a TensorHandle with import enabled (IsMemoryManaged = false)

CreateTensorHandles

```
                        switch (layer->GetType())
            {
                case LayerType::Input:
                case LayerType::MemImport:
                                case LayerType::FallbackIMport:
                {
                    // If IsImportEnabled is true then we need to set IsMemoryManaged
                    // to false when creating TensorHandles
                    layer->CreateTensorHandles(m_TensorHandleFactoryRegistry,
                                               workloadFactory,
                                               !supportsExternalManager &&
                                               (m_NetworkProperties.m_InputSource == MemorySource::
Undefined));
                    break;
                }
                case LayerType::Constant:
                {
                    layer->CreateTensorHandles(m_TensorHandleFactoryRegistry, workloadFactory, true);
                    break;
                }
                                default:
                {
                    // Look for a layer with 1 OutputSlot which has 1 connection and that connection is an
Output Layer
                    // If Import is enabled disable memory management so we can do memory import, otherwise
we do a copy
                    if ((layer->GetNumOutputSlots() == 1) &&
                       (layer->GetOutputSlots()[0].GetNumConnections() == 1) &&
                       (layer->GetOutputSlots()[0].GetConnection(0)->GetOwningLayer().GetType() == LayerType::
Output))
                    {
                        layer->CreateTensorHandles(m_TensorHandleFactoryRegistry,
                                                   workloadFactory,
                                                   !supportsExternalManager &&
                                                   (m_NetworkProperties.m_OutputSource == MemorySource::
Undefined));
                    }
                    // The layer before FallbackImportLayer
                    else if ((layer->GetNumOutputSlots() == 1) &&
                       (layer->GetOutputSlots()[0].GetNumConnections() == 1) &&
                       (layer->GetOutputSlots()[0].GetConnection(0)->GetOwningLayer().GetType() == LayerType::
FallbackImport))
                    {
                        layer->CreateTensorHandles(m_TensorHandleFactoryRegistry,
                                                   workloadFactory,
                                                   !supportsExternalManager &&
                                                   (m_NetworkProperties.m_OutputSource == MemorySource::
Undefined));
                    }
                    else
                    {
                        layer->CreateTensorHandles(m_TensorHandleFactoryRegistry,
                                                   workloadFactory,
                                                   !supportsExternalManager);
                    }
                }
                        }
```

Also add FallbackMemoryManager to LoadedNetwork, which will be created when load network (LoadedNetwork::LoadedNetwork)

LoadedNetwork-FallbackManager

```
std::unique_ptr<FallbackMemoryManager> m_FallbackMemoryManager;
```

Add GetMemoryAlignment to ITensorHandleFactory which should be implemented in each TensorHandleFactory for each backend.

EnqueueFallbackImport

```
size_t GetMemoryAlignment();
```

When network is loaded by LoadedNetwork::LoadedNetwork, FallbackImportWorkload will be created to be sync after execution of the subgraph in case the source backend has asynchronous execution. This is currently especially for GPU case - ClFallbackImportWorkload. In order to do the actual import so that the source backend and destination backend get the same aligned memory, EnqueueFallbackImport() is added to LoadedNetwork to create an aligned memory and import to the output of the subgraph of the source backend as well as import it to the input of the subgraph of the destination backend. This will be called after EnqueueInput in LoadedNetwork::EnqueueWorkload.

EnqueueFallbackImport

```
void EnqueueFallbackImport(const BindableLayer& layer)
{
    if (layer.GetType() != LayerType::FallbackImport)
    {
        throw InvalidArgumentException("EnqueueFallbackImport: given layer not a FallbackImport layer");
    }


    // cast to FallbackImport layer

        // Get memory alignment for source
    ITensorHandleFactory srcTensorHandleFactory = fallbackImportLayer.GetSrcTensorHandleFactory();
    size_t sourceAlignment = srcTensorHandleFactory.GetMemoryAlignment(fallbackImportLayer.GetMemorySource);

    // Do the same for destination
    // Get memory alignment for destination

    // Call FallbackMemoryManager to create aligned memory and import to the output of the subgraph of the
source backend as well as import it to the input of the subgraph of the destination backend.
    void* memory = m_FallbackMemoryManager->CreateFallbackMemory(fallbackImportLayer.GetMemorySource,
sourceAlignment, destAlignment);

    // Get source layer
    // Import the created memory to the output handler of the source layer

    // Import the created memory to its (FallbackImportLayer) output handler to be consume as an import of
the destination layer.
}
```

EnqueueWorkload

```
// call EnqueueFallbackImport after calling EnqueueInput()
for (auto&& layer : order)
{
        if (layer->GetType() == LayerType::FallBackImport)
    {
        EnqueueFallbackImport(layer);
    }
}
```

On network unload, we have to deallocate fallback memory by calling FallbackMemoryManager and pass the network Id. This should be added in *LoadedNetwork::FreeWorkingMemory()*.

DeallocateFallbackMemory

```
m_FallbackMemoryManager->DeallocateMemory();
```

Fallback Memory Manager

Fallback Memory Manager is used to allocate a memory that align for both source and destination backends to be able to import it to the output of the subgraph of the source backend as well as to the input of the subgraph of the destination backend. The memory will be registered for each Network Id and be kept for the whole network life and removed when the network is unloaded. Benefit of this is so that we could reuse the network without reallocating the memory.

FallbackMemManager

```
class FallbackMemoryManager
{
public:
        void* CreateFallbackMemory(MemorySource memSource, const size_t sourceAlignment, const size_t
destAlignment)
        {
                /// Check if source alignment = destination alignment
                /// If not, find the least common alignment
                /// Create a BufferStorage as specified by MemorySource that aligns to both source and
destination
        }
        void DeallocateMemory()
        {
                /// Remove all fallback memories registered with the Network Id
  }
private:
        /// Storage to register the memories belongs to each Network Id, the memories will be kept for the
whole time where the network is alive, and be removed when the network is unloaded.
         std::vector<BufferStorage>  m_BufferStoragesMap;

}
```

FallbackMemoryAllocator

MallocFallbackMemoryAllocator and DmabufFallbackMemoryAllocator implement armnn::ICustomAllocator to allocate fallback memory that aligned between source and destination backends. Example of Dmabuf memory allocator can be found in ClDmaBuf/DmaBufClBackendCustomAllocator.hpp (armnn-internal-tests). MallocFallbackMemoryAllocator is optional for easily test on CPU, the example of malloc allocation can be found in src/backends /cl/test/ClFallbackTests.cpp.

FallbackMemAllocator

```
class MallocFallbackMemoryAllocator : public armnn::ICustomAllocator
{
public:
    MallocFallbackMemoryAllocator();

    void* allocate(size_t size, size_t alignment);

    void free(void* ptr);

    armnn::MemorySource GetMemorySourceType()
        {
            return armnn::MemorySource::Malloc;
        }

    void PopulateData(void* ptr, const uint8_t* inData, size_t len);
    void RetrieveData(void* ptr, uint8_t* outData, size_t len);
};



class DmabufFallbackMemoryAllocator : public armnn::ICustomAllocator
{
public:
    DmabufFallbackMemoryAllocator();

    void* allocate(size_t size, size_t alignment);

    void free(void* ptr);

    armnn::MemorySource GetMemorySourceType()
        {
            return armnn::MemorySource::DmaBuf;
        }

    void PopulateData(void* ptr, const uint8_t* inData, size_t len);
    void RetrieveData(void* ptr, uint8_t* outData, size_t len);

};
```

Performance Improvement

Run tests and compare the scenarios between memory import and memory copy in case of fallback (GPU NPU). Also investigate if without alignment when we create the memory will reduce memory consumption while getting the same result and performance.

The work we require for NPU/GPU fallback (with DmaBuf) should be similar to CPU/GPU fallback (with malloc). We could also test with CPU/GPU fallback which might be easier and more familiar to us during development. However, the actual tests for GPU/NPU fallback also need to be added to ensure that it works correctly. As CPU and GPU can be directly compatible, which means in case of fallback from CPU to GPU, it can use MemImportLayer and import memory without using FallbackMemoryManager to import the memory to both backends. In order to test CPU GPU with FallbackMemoryManager, we have to add CapabilityClass::FallbackImportMemoryRequired to NeonTensorHandleFactory. This could reduce the performance so it should be added for testing purpose only, not in the actual implementation on production.

API/ABI/Version number implications

API changes

Implementation Plan

1. FallbackImportLayer front-end
2. FallbackMemoryManager implementation
3. MallocFallbackMemoryAllocator (optional for easily test on CPU)
4. DmabufFallbackMemoryAllocator
5. FallbackImportWorkload implementation
6. EnqueueFallbackImport function implementation
7. Fallback import workflow
8. End-to-end tests
    a. GPU CPU (optional for easily test on CPU)
    b. CPU GPU (optional for easily test on CPU)
    c. GPU NPU
    d. NPU GPU
    e. GPU NPU GPU
    f. NPU GPU NPU
9. Performance investigation

Test Strategy

Unit tests for each functionalities and end-to-end tests covering different fallback scenarios. The works we require for NPU/GPU fallback (with DmaBuf) should be similar to CPU/GPU fallback (with malloc). We could also test with CPU/GPU fallback which might be easier and more familiar to us during development. However, the actual tests for GPU/NPU fallback also need to be added to ensure that it works correctly.

As CPU and GPU can be direct compatible, which means in case of fallback from CPU to GPU, it can use MemImportLayer and import memory without using FallbackMemoryManager to import the memory to both backends. In order to test CPU GPU with FallbackMemoryManager, we have to add CapabilityClass::FallbackImportMemoryRequired to NeonTensorHandleFactory.

- GPU CPU
- CPU GPU
- GPU NPU
- NPU GPU
- GPU NPU GPU
- NPU GPU NPU

Different complexity also need to be tested for the fallback scenarios.

- Simple: single input - output between subgraphs
- Multiple inputs - outputs between subgraphs