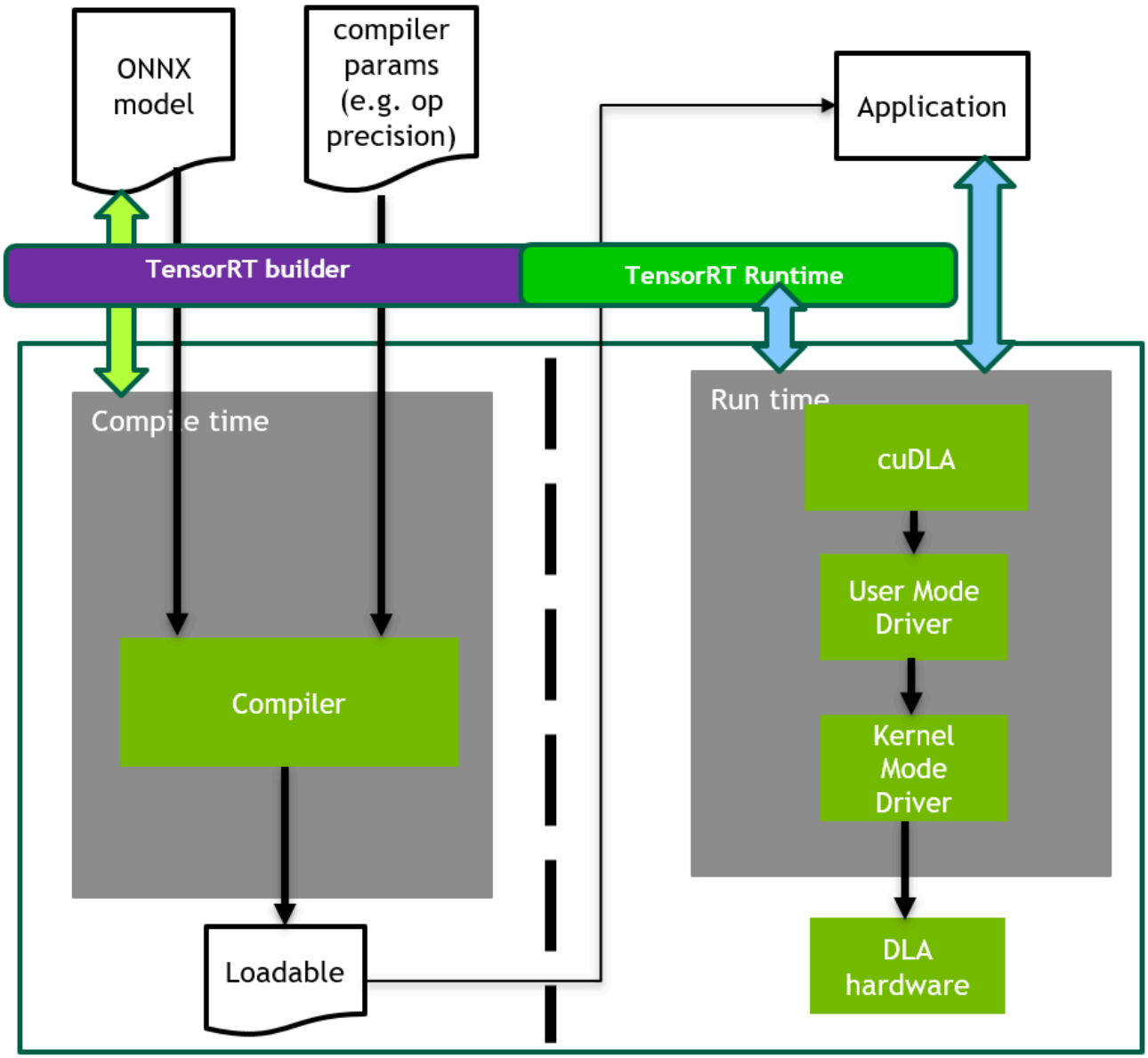
Working with DLA

NVIDIA DLA (Deep Learning Accelerator) is a fixed-function accelerator engine targeted for deep learning operations. It is designed to fully hardware accelerate convolutional neural networks. DLA supports various layers, such as convolution, deconvolution, fully connected, activation, pooling, batch normalization, etc. Refer to the DLA Supported Layers and Restrictions section for more information about DLA support in TensorRT layers.

DLA is useful for offloading CNN processing from the GPU and is significantly more power-efficient for these workloads. In addition, it can provide an independent execution pipeline in cases where redundancy is important, such as mission-critical or safety applications.

For more information, refer to the DLA Developer page and the DLA tutorial Getting Started with the Deep Learning Accelerator on NVIDIA Jetson Orin.

When building a model for DLA, the TensorRT builder parses the network and calls the DLA compiler to compile the network into a DLA loadable. Refer to Using trtexec to see how to build and run networks on DLA.



Building and Launching the Loadable

There are several ways to build and launch a DLA loadable, either embedded in a TensorRT engine or a standalone form.

Refer to the DLA Standalone Mode section to generate a standalone DLA loadable outside TensorRT.

Using `trtexec`

To allow `trtexec` to use the DLA, you can use the `--useDLACore` flag. For example, to run the ResNet-50 network on DLA core 0 in FP16 mode, with GPU Fallback Mode for unsupported layers, run:

```
./trtexec --onnx=data/resnet50/ResNet50.onnx --useDLACore=0 --fp16 --allowGPUFal
```

The `trtexec` tool has additional arguments for running networks on DLA. For more information, refer to the Command-Line Programs section.

Using the TensorRT API

You can use the TensorRT API to build and run inference with DLA and to enable DLA at the layer level. The relevant APIs and samples are provided in the following sections.

Running on DLA during TensorRT Inference

The TensorRT builder can be configured to enable inference on DLA. DLA support is currently limited to networks running in FP16 and INT8 mode. The `DeviceType` enumeration is used to specify the device on which the network or layer executes. The following API functions in the `IBuilderConfig` class can be used to configure the network to use DLA:

- `setDeviceType(ILayer* layer, DeviceType deviceType)`: This function sets the `deviceType` on which the layer must execute.
- `getDeviceType(const ILayer* layer)`: This function can be used to return the `deviceType` that this layer executes on. If the layer is executing on the GPU, this returns `DeviceType::kGPU`.
- `canRunOnDLA(const ILayer* layer)`: This function checks whether a layer can run on DLA.
- `setDefaultDeviceType(DeviceType deviceType)`: This function sets the builder's default `deviceType`. It ensures that all the layers that can run on DLA run on DLA unless `setDeviceType` is used to override the `deviceType` for a layer.
- `getDefaultDeviceType()`: This function returns the default `deviceType` set by `setDefaultDeviceType`.
- `isDeviceTypeSet(const ILayer* layer)`: This function checks whether the `deviceType` has been explicitly set for this layer.
- `resetDeviceType(ILayer* layer)`: This function resets the `deviceType` for this layer. The value is reset to the `deviceType` specified by `setDefaultDeviceType` or `DeviceType::kGPU` if none is specified.

- `allowGPUFallback(bool setFallBackMode)`: This function notifies the builder to use GPU if a layer that was supposed to run on DLA cannot run on DLA. For more information, refer to the GPU Fallback Mode section.
- `reset()`: This function can reset the `IBuilderConfig` state, which sets the `deviceType` for all layers to `DeviceType::kGPU`. After reset, the builder can be reused to build another network with a different DLA config.

The following API functions in the `IBuilder` class can be used to help configure the network for using the DLA:

- `getMaxDLABatchSize()`: This function returns the maximum batch size DLA can support.

  > **Note**
  >
  > For any tensor, the total volume of index dimensions combined with the requested batch size must not exceed the value returned by this function.

- `getNbDLACores()`: This function returns the number of DLA cores available to the user.

If the builder is not accessible, such as when a plan file is being loaded online in an inference application, then the DLA to be used can be specified differently using DLA extensions to the `IRuntime`. The following API functions in the IRuntime class can be used to configure the network to use DLA:

- `getNbDLACores()`: This function returns the number of DLA cores accessible to the user.
- `setDLACore(int dlaCore)`: The DLA core to execute on. Where `dlaCore` is a value between `0` and `getNbDLACores() - 1`. The default value is `0`.
- `getDLACore()`: The DLA core to which the runtime execution is assigned. The default value is `0`.

Example: Run Samples With DLA

This section details how to run a TensorRT sample with DLA enabled.

1. Create the builder.

```
auto builder = SampleUniquePtr<nvinfer1::IBuilder>(nvinfer1::createInferBuil
if (!builder) return false;
builder->setMaxBatchSize(batchSize);
config->setMaxWorkspaceSize(16_MB);
```

2. Enable `GPUFallback` mode.

```
config->setFlag(BuilderFlag::kGPU_FALLBACK);
config->setFlag(BuilderFlag::kFP16); or config->setFlag(BuilderFlag::kINT8);
```

3. Enable execution on DLA, where `dlaCore` specifies the DLA core to execute on.

```
config->setDefaultDeviceType(DeviceType::kDLA);
config->setDLACore(dlaCore);
```

4. With these additional changes, sampleMNIST is ready to execute on DLA. To run samples with DLA Core 1, append `--useDLACore=0` to the sample command.

Example: Enable DLA Mode for a Layer during Network Creation

In this example, let us create a simple network with Input, Convolution, and Output.

1. Create the builder, builder configuration, and the network.

```
IBuilder* builder = createInferBuilder(gLogger);
IBuilderConfig* config = builder.createBuilderConfig();
INetworkDefinition* network = builder->createNetworkV2(0U);
```

2. Add the Input layer to the network with the input dimensions.

```
auto data = network->addInput(INPUT_BLOB_NAME, dt, Dims3{1, INPUT_H, INPUT_W
```

3. Add the Convolution layer with hidden layer input nodes, strides, and weights for filter and bias.

```
auto conv1 = network->addConvolution(*data->getOutput(0), 20, DimsHW{5, 5},
conv1->setStride(DimsHW{1, 1});
```

4. Set the Convolution layer to run on DLA.

```
if(canRunOnDLA(conv1))
{
config->setFlag(BuilderFlag::kFP16); or config->setFlag(BuilderFlag::kINT8);
builder->setDeviceType(conv1, DeviceType::kDLA);

}
```

5. Mark the output.

```
network->markOutput(*conv1->getOutput(0));
```

6. Set the DLA core to execute on.

```
config->setDLACore(0)
```

Using the cuDLA API

cuDLA is an extension of the CUDA programming model that integrates DLA runtime software with CUDA. This integration makes it possible to launch DLA loadables using CUDA programming constructs such as streams and graphs.

CuDLA transparently manages shared buffers and synchronizes the tasks between GPU and DLA. Refer to the NVIDIA cuDLA documentation on how the cuDLA APIs can be used for these use cases while writing a cuDLA application.

Refer to the DLA Standalone Mode section for more information on using TensorRT to build a standalone DLA loadable usable with cuDLA.

DLA Supported Layers and Restrictions

This section lists the layers supported by DLA and the constraints associated with each layer.

General Restrictions

The following restrictions apply to all layers while running on DLA:

- The maximum supported batch size is 4096.

- The maximum supported size for non-batch dimensions is 8192.
- DLA does not support dynamic dimensions. Thus, the profile's `min`, `max`, and `opt` values must be equal for wildcard dimensions.
- The runtime dimensions must be the same as the dimensions used for building.
- TensorRT may split a network into multiple DLA loadable if any intermediate layers cannot run on DLA and `GPUFallback` is enabled. Otherwise, TensorRT can emit an error and fallback. For more information, refer to the GPU Fallback Mode section.
- Due to hardware and software memory limitations, only 16 DLA loadable can be loaded concurrently per core.
- Each layer must have the same batch size within a single DLA loadable. Layers with different batch sizes will be partitioned into separate DLA graphs.

> **Note**
>
> Batch size for DLA is the product of all index dimensions except the `CHW` dimensions. For example, if input dimensions are `NPQRS`, the effective batch size is `N*P`.

Layer Support and Restrictions

The following list provides layer support and restrictions to the specified layers while running on DLA:

Activation layer

- Only two spatial dimension operations are supported.
- Both FP16 and INT8 are supported.
- Functions supported: `ReLU`, `Sigmoid`, `TanH`, `Clipped ReLU`, and `Leaky ReLU`.
  - A negative slope is not supported for *ReLU*.
  - `Clipped ReLU` only supports values in the range `[1, 127]`.
  - `TanH`, `Sigmoid` INT8 support is supported by auto-upgrading to FP16.

Comparison operations (Equal, Greater, Less)

- It only supports INT8 layer precision and INT8 inputs except when using constants, which should be of the FP32 type and filled with the same value.
- DLA requires that the comparison operation output be FP16 or INT8 type. Thus, the comparison layer must be immediately followed by a Cast operation (`IIdentityLayer` or `ICastLayer`) to FP16 or INT8 and should have no direct consumers other than this Cast operation.
- The `ElementWise` comparison layer and the subsequent `IIdentityLayer` or `ICastLayer` mentioned above explicitly set your device types to DLA and their precisions to INT8. Otherwise, these layers will run on the GPU.
- Even with GPU fallback allowed, you should expect failures in engine construction in some cases, such as when DLA loadable compilation fails. If this is the case, unset the device types and/or precisions of both the `ElementWise` comparison layer and `IIdentityLayer` or `ICastLayer` to have both offloaded to GPU.

Concatenation layer

- DLA supports concatenation only along the channel axis.
- Concat must have at least two inputs.
- All the inputs must have the same spatial dimensions.
- Both FP16 and INT8 are supported.
- With INT8 mode, the inputs' dynamic range must be the same.
- With INT8 mode, the dynamic range of output must be equal to each of the inputs.

Convolution and Fully Connected layers

- Only two spatial dimension operations are supported.
- Both FP16 and INT8 are supported.
- Each dimension of the kernel size must be in the range `[1, 32]`.
- Padding must be in the range `[0, 31]`.
- Dimensions of padding must be less than the corresponding kernel dimension.
- Dimensions of stride must be in the range `[1, 8]`.
- The number of output maps must be in the range `[1, 8192]`.

- Number of input channels `[1, 8192]`.
- For operations using the `TensorFormat::kDLA_LINEAR`, `TensorFormat::kCHW16`, and `TensorFormat::kCHW32` formats, the number of groups must be in the range `[1, 8192]`.
- For operations using the `TensorFormat::kDLA_HWC4` format, the number of groups must be in the range `[1, 4]`.
- Dilated convolution must be in the range `[1, 32]`.
- Operations are not supported if the CBUF size requirement `wtBanksForOneKernel + minDataBanks` exceeds the `numConvBufBankAllotted` limitation `16`, where CBUF is the internal convolution cache that stores input weights and activation before operating on them, `wtBanksForOneKernel` is the minimum banks for one kernel to store the minimum weight/kernel elements needed for convolution, and `minDataBanks` is the minimum banks to store the minimum activation data needed for convolution. Detailed details are displayed in the logging output when a convolution layer fails validation due to CBUF constraints.

Deconvolution layer

- Only two spatial dimensions are supported.
- Both FP16 and INT8 are supported.
- The kernel dimensions and strides must be in the range `[1, 32]` or must be `1x[64, 96, 128]` and `[64, 96, 128]x1`.
- TensorRT has disabled deconvolution square kernels and strides in the range `[23 - 32]` on DLA as they significantly slow down compilation.
- The padding must be `0`.
- Grouped deconvolution must be `1`.
- Dilated deconvolutions must be `1`.
- The number of input channels must be in the range `[1, 8192]`.
- The number of output channels must be in the range `[1, 8192]`.

ElementWise layer

- Only two spatial dimension operations are supported.

- Both FP16 and INT8 are supported.
- Operations supported: `Sum`, `Sub`, `Product`, `Max`, `Min`, `Div`, `Pow`, `Equal`, `Greater`, and `Less` (described separately).
- Broadcasting is supported when one of the operands has one of the following shape configurations:
  - NCHW (that is, shapes equal)
  - NC11 (that is, N and C equal, H and W are 1)
  - N111 (that is, N equal, C, H, and W are 1)
- `Div` operation
  - The first input (dividend) can be INT8, FP16, or an FP32 constant. The second input (divisor) must be INT8 or an FP32 constant.
  - If one of the inputs is constant, all values of its weights must be the same. Additionally, the other input must be non-constant in INT8.
- `Pow` operation
  - One input must be an FP32 constant filled with the same value; the other must be an INT8 non-constant.

LRN (Local Response Normalization) layer

- Allowed window sizes are `3`, `5`, `7`, or `9`.
- The normalization region supported is `ACROSS_CHANNELS`.
- LRN INT8 is supported by auto-upgrading to FP16.

Parametric ReLU layer

- Slope input must be a build time constant with the same rank as the input tensor.

Pooling layer

- Only two spatial dimension operations are supported.
- Both FP16 and INT8 are supported.
- Operations supported: `kMAX`, `kAVERAGE`.

- Dimensions of the window must be in the range `[1, 8]`.
- Dimensions of padding must be in the range `[0, 7]`.
- Dimensions of stride must be in the range `[1, 16]`.
- With INT8 mode, input and output tensor scales must be the same.

Reduce layer

- Only supports 4D input tensors.
- All input non-batch dimensions must be in the range `[1, 8192]`.
- Both FP16 and INT8 are supported.
- Only supports MAX operation type where any combination of the CHW axes is reduced.

Resize layer

- The number of scales must be exactly `4`.
- The first two scale elements must be exactly `1` (for unchanged batch and channel dimensions).
- The last two elements in scales, representing the scale values along height and width dimensions, respectively, must be integer values in the range of `[1, 32]` in nearest-neighbor mode and `[1, 4]` in bilinear mode.
- Note that for bilinear resize INT8 mode, when the input dynamic range is larger than the output dynamic range, the layer will be upgraded to FP16 to preserve accuracy. This can negatively affect the latency.

Scale layer

- Only two spatial dimension operations are supported.
- Both FP16 and INT8 are supported.
- Mode supported: `Uniform`, `Per-Channel`, and `ElementWise`.
- Only `scale` and `shift` operations are supported.

Shuffle layer

- Only supports 4D input tensors.

- All input non-batch dimensions must be in the range `[1, 8192]`.
- Note that DLA decomposes the layer into standalone transpose and reshape operations. This means that the above restrictions apply individually to each decomposed operation.
- Batch dimensions cannot be involved in either reshapes or transposes.

Slice layer

- Both FP16 and INT8 are supported.
- It supports batch sizes up to the general DLA maximum.
- All input non-batch dimensions must be in the range `[1, 8192]`.
- Only supports 4D inputs and slicing at CHW dimensions.
- Only supports static slicing, so slice parameters must be provided statically using TensorRT `ISliceLayer` setter APIs or as constant input tensors.

Softmax layer

- Only supported on NVIDIA Orin, not Xavier.
- All input non-batch dimensions must be in the range `[1, 8192]`.
- The axis must be one of the non-batch dimensions.
- Supports FP16 and INT8 precision.
- Internally, there are two modes, and the mode is selected based on the given input tensor shape.
  - The accurate mode is triggered when all non-batch, non-axis dimensions are `1`.
  - The optimized mode allows the non-batch, non-axis dimensions to be greater than `1` but restricts the axis dimension to 1024 and involves an approximation that may cause a small error in the output. The magnitude of the error increases as the size of the axis dimension approaches 1024.

Unary layer

- DLA supports `ABS`, `SIN`, `COS`, and `ATAN` operation types.
- For `SIN`, `COS`, and `ATAN`, input precision must be INT8.

- All input non-batch dimensions must be in the range `[1, 8192]`.

Inference on NVIDIA Orin

Due to the difference in hardware specifications between NVIDIA Orin and Xavier DLA, FP16 convolution operations on NVIDIA Orin may experience an increase in latency of up to 2x.

On NVIDIA Orin, DLA stores weights for non-convolution operations (FP16 and INT8) inside loadable as FP19 values (which use 4-byte containers). The channel dimensions are padded to multiples of either 16 (FP16) or 32 (INT8) for those FP19 values. Especially in the case of large per-element `Scale`, `Add`, or `Sub` operations, this can inflate the size of the DLA loadable, inflating the engine containing such a loadable. Graph optimization may unintentionally trigger this behavior by changing the type of a layer, for example, when an `ElementWise` multiplication layer with a constant layer as weights is fused into a scale layer.

GPU Fallback Mode

The `GPUFallbackMode` sets the builder to use GPU if a layer marked to run on DLA could not run on DLA. A layer cannot run on DLA due to the following reasons:

- The `layer` operation is not supported on DLA.
- The parameters specified are out of the supported range for DLA.
- The given batch size exceeds the maximum permissible DLA batch size. For more information, refer to the DLA Supported Layers and Restrictions section.
- A combination of layers in the network causes the internal state to exceed what the DLA can support.
- There are no DLA engines available on the platform.

When GPU fallback is disabled, an error is emitted if a layer cannot be run on DLA.

I/O Formats on DLA

DLA supports formats that are unique to the device and have constraints on their layout due to vector width byte requirements.

For DLA input tensors, `kDLA_LINEAR(FP16, INT8)`, `kDLA_HWC4(FP16, INT8)`, `kCHW16(FP16)`, and `kCHW32(INT8)` are supported.

For DLA output tensors, only `kDLA_LINEAR(FP16, INT8)`, `kCHW16(FP16)`, and `kCHW32(INT8)` are supported.

For `kCHW16` and `kCHW32` formats, if `C` is not an integer multiple, it must be padded to the next 32-byte boundary.

For `kDLA_LINEAR` format, the stride along the `W` dimension must be padded up to 64 bytes. The memory format is equivalent to a `C` array with dimensions `[N][C][H][roundUp(W, 64/elementSize)]` where `elementSize` is `2` for FP16 and `1` for Int8, with the tensor coordinates `(n, c, h, w)` mapping to array subscript `[n][c][h][w]`.

For `kDLA_HWC4` format, the stride along the `W` dimension must be a multiple of 32 bytes on Xavier and 64 bytes on NVIDIA Orin.

- When `C == 1`, TensorRT maps the format to the native grayscale image format.
- When `C == 3` or `C == 4`, it maps to the native color image format. If `C == 3`, the stride for stepping along the W axis must be padded to `4` in elements.
  - In this case, the padded channel is located at the 4th index. Ideally, the padding value does not matter because the DLA compiler paddings the 4th channel in the weights to zero; however, it is safe for the application to allocate a zero-filled buffer of four channels and populate three valid channels.
- When `C` is `{1, 3, 4}`, then padded `C'` is `{1, 4, 4}` respectively, the memory layout is equivalent to a `C` array with dimensions `[N][H][roundUp(W, 32/C'/elementSize)][C']` where `elementSize` is `2` for FP16 and `1` for Int8. The tensor coordinates `(n, c, h, w)` mapping to array subscript `[n][h][w][c]`, `roundUp` calculates the smallest multiple of `64/elementSize` greater than or equal to `W`.

When using `kDLA_HWC4` as the DLA input format, it has the following requirements:

- `C` must be `1`, `3`, or `4`
- The first layer must be convolution.
- The convolution parameters must meet DLA requirements. For more information, refer to the DLA Supported Layers and Restrictions section.

When GPU fallback is enabled, TensorRT may insert reformatting layers to meet the DLA requirements. Otherwise, the input and output formats must be compatible with DLA. In all

cases, the strides that TensorRT expects data to be formatted with can be obtained by querying `IExecutionContext::getStrides`.

DLA Standalone Mode

If you need to run inference outside of TensorRT, you can use `EngineCapability::kDLA_STANDALONE` to generate a DLA loadable instead of a TensorRT engine. This loadable can then be used with the cuDLA API.

Building A DLA Loadable Using C++

1. Set the default device type and engine capability to DLA standalone mode.

```
builderConfig->setDefaultDeviceType(DeviceType::kDLA);
builderConfig->setEngineCapability(EngineCapability::kDLA_STANDALONE);
```

2. Specify FP16, INT8, or both. For example:

```
builderConfig->setFlag(BuilderFlag::kFP16);
```

3. DLA standalone mode disallows reformatting; therefore, `BuilderFlag::kDIRECT_IO` needs to be set.

```
builderConfig->setFlag(BuilderFlag::kDIRECT_IO);
```

4. Set the allowed formats for I/O tensors to one or more of those that are DLA-supported.
5. Build as normal.

Using trtexec To Generate A DLA Loadable

The `trtexec` tool can generate a DLA loadable instead of a TensorRT engine. Specifying both `--useDLACore` and `--safe` parameters sets the builder capability to `EngineCapability::kDLA_STANDALONE`. Additionally, specifying `--inputIOFormats` and `--outputIOFormats` restricts I/O data type and memory layout. The DLA loadable is saved into a file by specifying `--saveEngine` parameter.

For example, to generate an FP16 DLA loadable for an ONNX model using `trtexec`, run:

```
./trtexec --onnx=model.onnx --saveEngine=model_loadable.bin --useDLACore=0 --fp1
```

Customizing DLA Memory Pools

You can customize the size of the memory pools allocated to each DLA subnetwork in a network using the `IBuilderConfig::setMemoryPoolLimit` (C++) or the `IBuilderConfig.set_memory_pool_limit` (Python). There are three types of DLA memory pools (refer to the `MemoryPoolType` enum for details):

Managed SRAM

- Behaves like a cache, and larger values may improve performance.
- If no managed SRAM is available, DLA can still run by falling back to local DRAM.
- On Orin, each DLA core has 1 MiB of dedicated SRAM. On Xavier, 4 MiB of SRAM is shared across multiple cores, including the 2 DLA cores.

Local DRAM

- Used to store intermediate tensors in the DLA subnetwork. Larger values may allow larger subnetworks to be offloaded to DLA.

Global DRAM

- Used to store weights in the DLA subnetwork. Larger values may allow larger subnetworks to be offloaded to DLA.

The memory required for each subnetwork may be less than the pool size, in which case a smaller amount will be allocated. The pool size serves only as an upper bound.

Note that all DLA memory pools require sizes that are powers of 2, with a minimum of 4 KiB. Violating this requirement results in a DLA loadable compilation failure.

In multi-subnetwork situations, it is important to remember that the pool sizes apply per DLA subnetwork, not for the whole network, so it is necessary to know the total amount of resources consumed. In particular, your network can consume at most twice the managed SRAM as the pool size in aggregate.

The default managed SRAM pool size for NVIDIA Orin is set to 0.5 MiB, whereas Xavier has 1 MiB as the default. Orin has a strict per-core limit, whereas Xavier has some flexibility. This Orin default guarantees that in all situations, the aggregate-managed SRAM consumption of your engine will stay below the hardware limit. Still, if your engine has only a single DLA subnetwork, this would mean your engine only consumes half the hardware limit, so you may see a perf boost by increasing the pool size to 1 MiB.

Determining DLA Memory Pool Usage

Upon successfully compiling loadable from the given network, the builder reports the number of subnetwork candidates successfully compiled into loadable and the total