Documentation for the nGraph Library and Compiler Stack

[](https://github.com/NervanaSystems/ngraph/blob/master/LICENSE) [](https://www.github.com/NervanaSystems/ngraph)



Version 0.29

* [Introduction](index.odt/index.html" \l "document-introduction)

Framework Support

* [Basic concepts](index.odt/index.html" \l "document-frameworks/overview)
* [TensorFlow\*](index.odt/index.html" \l "document-frameworks/tensorflow_connect)
* [ONNX](index.odt/index.html" \l "document-frameworks/onnx_integ)
* [PaddlePaddle\*](index.odt/index.html" \l "document-frameworks/paddle_integ)

nGraph Core

* [Basic Concepts](index.odt/index.html" \l "document-core/overview)
* [Build and Test](index.odt/index.html" \l "document-buildlb)
* [Constructing Graphs](index.odt/index.html" \l "document-core/constructing-graphs/index)
* [Compiler Passes](index.odt/index.html" \l "document-core/passes/passes)
* [Pattern Matcher](index.odt/index.html" \l "document-core/fusion/index)
* [nGraph Core Ops](index.odt/index.html" \l "document-ops/index)
* [Provenance](index.odt/index.html" \l "document-provenance/index)
* [Quantization](index.odt/index.html" \l "document-core/quantization)
* [Dynamic Shapes](index.odt/index.html" \l "document-dynamic/index)

Backend Support

* [Basic Concepts](index.odt/index.html" \l "document-backends/index)
* [Adding New Backends](index.odt/index.html" \l "document-backends/cpp-api)

Training

* [Distributed Training](index.odt/index.html" \l "document-training/index)
* [Quantization-Aware Training](index.odt/index.html" \l "document-training/qat)

Validated Workloads

* [Validated Workloads](index.odt/index.html" \l "document-frameworks/validated/list)

Diagnostics

* [Diagnostics](index.odt/index.html" \l "document-inspection/debug_core)
* [Debug TensorFlow\*](index.odt/index.html" \l "document-inspection/debug_tf)
* [Debug ONNX](index.odt/index.html" \l "document-inspection/debug_onnx)
* [Debug PaddlePaddle\*](index.odt/index.html" \l "document-inspection/debug_paddle)
* [General Visualization Tools](index.odt/index.html" \l "document-inspection/viz_tools)
* [Performance testing with](index.odt/index.html" \l "document-inspection/profiling) nbench

Contributing

* [Contribution Guide](index.odt/index.html" \l "document-project/contribution-guide)
* [Glossary](index.odt/index.html" \l "document-glossary)

# nGraph Compiler Stack Documentation[¶](#ngraph-compiler-stack-documentation)

## Introduction[¶](#introduction)

Future developments in Artificial Intelligence will increasingly rely on better methods to accelerate the performance of deep learning workloads. As Deep Learning models become more complex, and as the volume of data those models are expected to handle increases rapidly, the deployment of scalable AI solutions becomes a greater challenge.

Today, two standard approaches to accelerate deep learning performance are:

1. **Design hardware solutions dedicated to deep learning computation** – Many companies, ranging from startups to established manufacturers such as Intel, are actively developing Application Specific Integrated Circuits to accelerate the performance of deep learning for both training and inference.
2. **Optimize software to accelerate performance** – nGraph Compiler, an open-source deep learning compiler, is Intel’s solution to deliver performance via software optimization. nGraph provides developers with a way to accelerate workloads via software and to provide a significant increase in performance for standard hardware targets such as CPUs and GPUs. For deploying scalable AI solutions, nGraph uses kernel libraries, a popular and effective method to improve deep learning performance. Where kernel libraries are available and perform well, we use them.

### Motivations[¶](#motivations)

The current State-of-the-Art software solution for deep learning computation is to integrate kernel libraries such as Intel® Math Kernel Library for Deep Neural Networks and Nvidia’s CuDNN into deep learning frameworks. These kernel libraries offer a performance boost during runtime on specific hardware targets through highly-optimized kernels and other operator-level optimizations.

However, kernel libraries have three main problems:

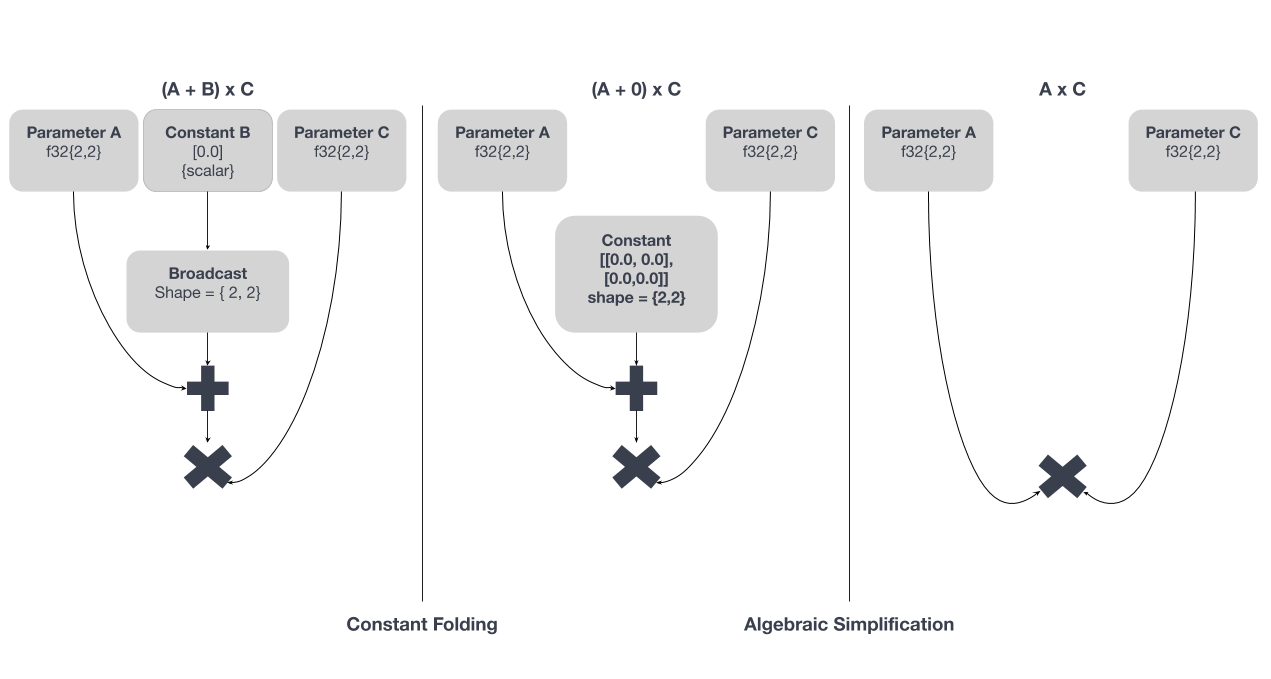
1. Kernel libraries do not support graph-level optimizations.
2. Framework integration of kernel libraries does not scale.
3. The number of required kernels keeps growing.

nGraph Compiler addresses the first two problems, and nGraph Compiler combined with PlaidML addresses the third problem. nGraph applies graph-level optimizations by taking the computational graph from a deep learning framework such as TensorFlow and reconstructing it with nGraph’s :abbr: IR (Intermediate Representation). nGraph IR centralizes computational graphs from various frameworks and provides a unified way to connect backends for targeted hardware. To address the third problem, nGraph is integrated with PlaidML, a tensor compiler, which generates code in LLVM, OpenCL, OpenGL, and Metal. Low-level optimizations are automatically applied to the generated code, resulting in a more efficient execution that does not require manual kernel integration for most hardware targets.

The following three sections explore the main problems of kernel libraries in more detail and describe how nGraph addresses them.

#### Problem 1: Kernel libraries do not support graph-level optimizations[¶](#problem-1-kernel-libraries-do-not-support-graph-level-optimizations)

The example diagrams below show how a deep learning framework, when integrated with a kernel library, can optimally run each operation in a computational graph, but the choice of operations in the graph may not be optimal.

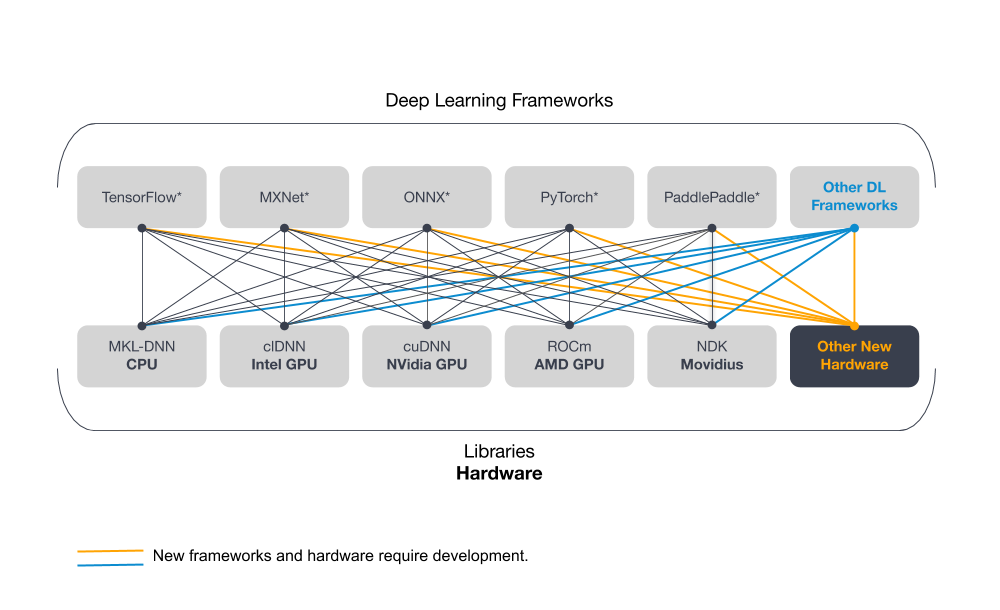
[](index.odt/_images/kernel-problem-1.png)

**Figure A**: The mathematical operations in a Deep Learning stack can be simplified significantly with a graph compiler

The computation is constructed to execute (A+B)\*C. With nGraph, we can further optimize the graph to be represented as A\*C. From the first graph shown on the left, the operation on the constant B can be computed at compile time (an optimization known as *constant folding*). The graph can be further simplified to the one on the right because the constant has a value of zero (known as *algebraic simplification*). Without such graph-level optimizations, a deep learning framework with a kernel library will compute all operations, resulting in suboptimal execution.

#### Problem 2: Framework integration of kernel libraries does not scale[¶](#problem-2-framework-integration-of-kernel-libraries-does-not-scale)

Due to the growing number of new deep learning accelerators, integrating kernel libraries with frameworks has become increasingly more difficult. For each new deep learning accelerator, a custom kernel library integration must be implemented by a team of experts. This labor-intensive work is further complicated by the number of frameworks, as illustrated in the following diagram.

[](index.odt/_images/kernel-problem-2.png)

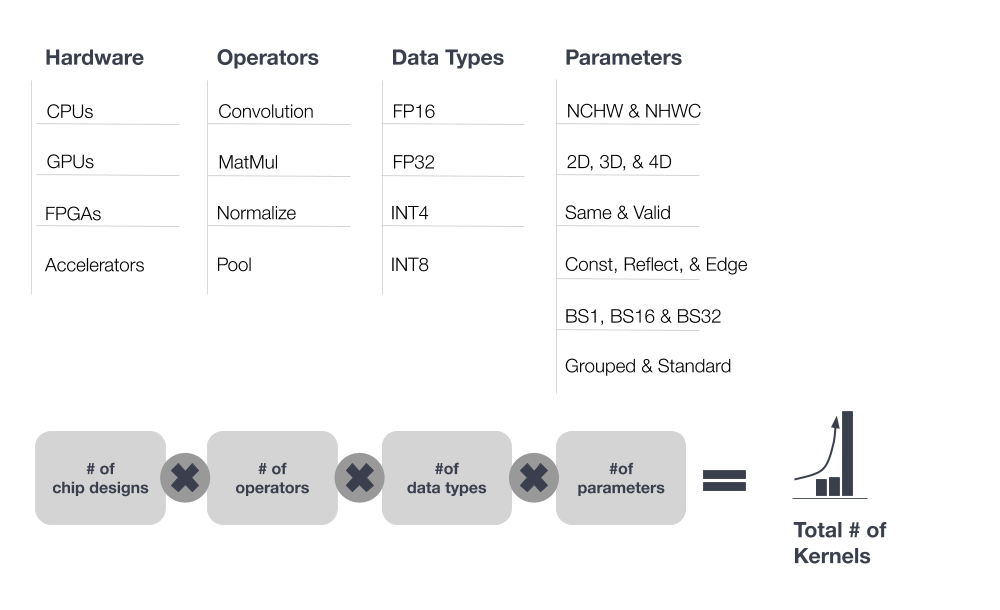
**Figure B**: A many-to-many problem

Each framework must be manually integrated with each hardware-specific kernel library. Additionally, each integration is unique to the framework and its set of deep learning operators, view on memory layout, feature set, etc. Each connection that needs to be made increases the amount of work, resulting in a fragile setup that is costly to maintain.

nGraph solves this problem with bridges. A bridge takes a computational graph or similar structure and reconstructs it in the nGraph IR along with a few primitive nGraph operations. With a unified computational graph, kernel libraries no longer need to be separately integrated into each deep learning framework. Instead, the libraries only need to support nGraph primitive operations, and this approach streamlines the integration process for the backend.

#### Problem 3: The number of required kernels keeps growing[¶](#problem-3-the-number-of-required-kernels-keeps-growing)

Integrating kernel libraries with multiple deep learning frameworks is a difficult task that becomes more complex with the growing number of kernels needed to achieve optimal performance. Past deep learning research has been built on a small set of standard computational primitives (convolution, GEMM, etc.). But as AI research advances and industrial deep learning applications continue to develop, the number of required kernels continues to increase exponentially. The number of required kernels is based on the number of chip designs, data types, operations, and the cardinality of each parameter per operation. Each connection in the following diagram represents significant work for what will ultimately be a fragile setup that is costly to maintain.

[](index.odt/_images/kernel-problem-3.png)

**Figure C**: Inevitable scaling problem

Integrating PlaidML with nGraph provides flexibility to support the latest deep learning models in the absence of hand-optimized kernels for new operations. PlaidML works together with nGraph to address the exponential growth of kernels.

PlaidML takes two inputs: the operation defined by the user and the machine description of the hardware target. It then automatically generates kernels that are iteratively optimized through an IR known as [Stripe](https://arxiv.org/abs/1903.06498). Integration of PlaidML with nGraph allows users to choose the hardware and framework that suits their needs, resulting in freedom from kernel libraries.

### Solution: nGraph and PlaidML[¶](#solution-ngraph-and-plaidml)

We developed nGraph and integrated it with PlaidML to allow developers to accelerate deep learning performance and address the problem of scalable kernel libraries. To address the problem of scaling backends, nGraph applies graph-level optimizations to deep learning computations and unifies computational graphs from deep learning frameworks with nGraph IR.

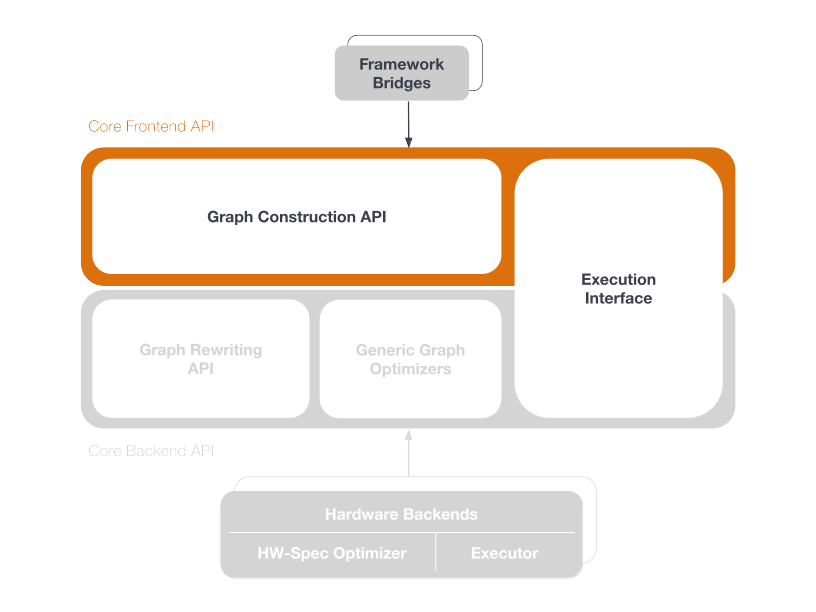
In conjunction with nGraph’s graph-level optimizations, PlaidML automatically applies low-level optimizations to improve deep learning performance. Additionally, PlaidML offers extensive support for various hardware targets due to its ability to generate code in LLVM, OpenCL, OpenGL, and Metal.

Given a backend with existing kernel libraries, nGraph can readily support the target hardware because the backend only needs to support a few primitive operations. If the hardware supports one of the coding languages supported by PlaidML, developers must specify the machine description to support the hardware. Together, nGraph and PlaidML provide the best of both worlds.

This documentation provides technical details of nGraph’s core functionality as well as framework and backend integrations. Creating a compiler stack like nGraph and PlaidML requires expert knowledge, and we’re confident that nGraph and PlaidML will make life easier for many kinds of developers:

1. Framework owners looking to support new hardware and custom chips.
2. Data scientists and ML developers wishing to accelerate deep learning performance.
3. New DL accelerator developers creating an end-to-end software stack from a deep learning framework to their silicon.

## Basic concepts[¶](#basic-concepts)

[](index.odt/_images/overview-framework-bridges.svg)

*A framework bridge connects to the nGraph graph construction API*

To understand how a data science [framework](index.odt/index.html" \l "term-framework) ([TensorFlow](index.odt/index.html" \l "document-frameworks/tensorflow_connect), PyTorch, [PaddlePaddle\*](index.odt/index.html" \l "document-frameworks/paddle_integ), and others) can unlock acceleration available in the nGraph Compiler, it helps to familiarize yourself with some basic concepts.

We use the term [bridge](index.odt/index.html" \l "term-bridge) to describe code that connects to any nGraph device backend(s) while maintaining the framework’s programmatic or user interface. We have a [bridge for the TensorFlow framework](https://github.com/tensorflow/ngraph-bridge/README.md). We also have a [PaddlePaddle\*](index.odt/index.html" \l "document-frameworks/paddle_integ) bridge. Intel previously [contributed work to an MXNet bridge](index.odt/index.html" \l "document-project/extras/testing_latency); however, support for the MXNet bridge is no longer active.

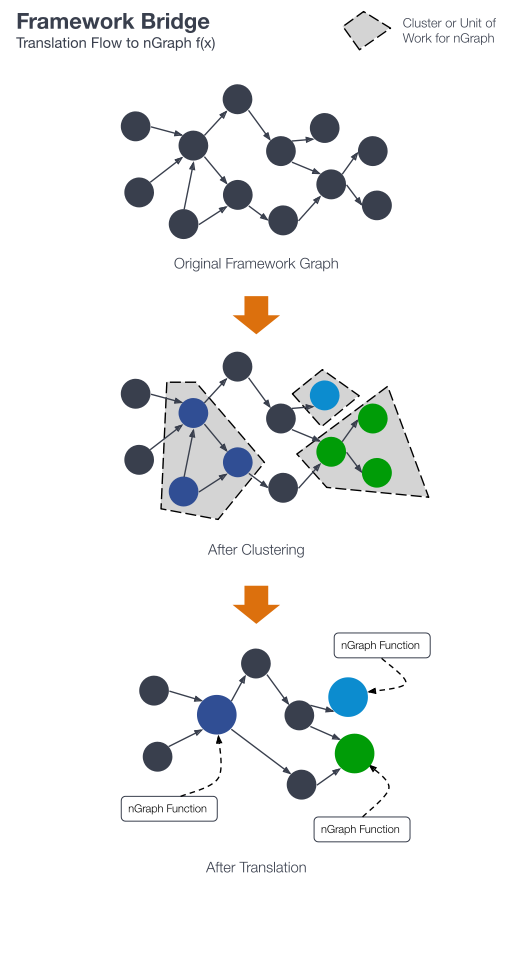
[ONNX](http://onnx.ai/) on its own is not a framework; it can be used with nGraph’s [Python API](index.odt/index.html" \l "document-python_api/index) to import and execute ONNX models.

Because it is framework agnostic (providing opportunities to optimize at the graph level), nGraph can do the heavy lifting required by many popular [workloads](index.odt/index.html" \l "document-frameworks/validated/list) without any additional effort of the framework user. Optimizations that were previously available only after careful integration of a kernel or hardware-specific library are exposed via the [Core graph construction API](index.odt/index.html" \l "document-core/constructing-graphs/index)

The illustration above shows how this works.

While a Deep Learning framework is ultimately meant for end-use by data scientists, or for deployment in cloud container environments, nGraph’s [Core ops](index.odt/index.html" \l "document-core/overview) are designed for framework builders themselves. We invite anyone working on new and novel frameworks or neural network designs to explore our highly-modularized stack of components.

Please read the other/index section for other framework-agnostic configurations available to users of the nGraph Compiler stack.

[](index.odt/_images/overview-translation-flow1.svg)

## TensorFlow\*[¶](#tensorflow)

See the [README](https://github.com/tensorflow/ngraph-bridge/blob/master/README.md) on the [ngraph\_bridge repo](https://github.com/tensorflow/ngraph-bridge) for the many ways to connect Tensorflow to nGraph, enabling a [DSO](http://csweb.cs.wfu.edu/~torgerse/Kokua/More_SGI/007-2360-010/sgi_html/ch03.html) backend that can speed up your TensorFlow training and inference workloads.

## ONNX[¶](#onnx)

nGraph is able to import and execute ONNX models. Models are converted to nGraph’s Intermediate Representation and converted to Function objects, which can be compiled and executed with nGraph backends.

You can use nGraph’s Python API to run an ONNX model and nGraph can be used as a backend to ONNX with the add-on package [nGraph ONNX](https://github.com/NervanaSystems/ngraph-onnx).

**Note:** In order to support ONNX, nGraph must be built with the NGRAPH\_ONNX\_IMPORT\_ENABLE flag. See [Building nGraph-ONNX](https://github.com/NervanaSystems/ngraph-onnx/blob/master/BUILDING.md) for more information. All nGraph packages published on PyPI are built with ONNX support.

### Importing an ONNX model[¶](#importing-an-onnx-model)

You can download models from the [ONNX Model Zoo](https://github.com/onnx/models). For example, ResNet-50:

$ wget https://s3.amazonaws.com/download.onnx/models/opset\_9/resnet50.tar.gz

$ tar -xzvf resnet50.tar.gz

Use the following Python commands to convert the downloaded model to an nGraph Function:

# Import ONNX and load an ONNX file from disk

>>> import onnx

>>> onnx\_protobuf = onnx.load('resnet50/model.onnx')

# Convert ONNX model to an ngraph model

>>> from ngraph.impl.onnx\_import import import\_onnx\_model

>>> ng\_function = import\_onnx\_model(onnx\_protobuf.SerializeToString())

# The importer returns a list of ngraph models for every ONNX graph output:

>>> print(ng\_function)

<Function: 'resnet50' ([1, 1000])>

This creates an nGraph Function object, which can be used to execute a computation on a chosen backend.

### Running a computation[¶](#running-a-computation)

You can now create an nGraph Runtime backend and use it to compile your Function to a backend-specific Computation object. Finally, you can execute your model by calling the created Computation object with input data:

# Using an nGraph runtime (CPU backend) create a callable computation object

>>> import ngraph as ng

>>> runtime = ng.runtime(backend\_name='CPU')

>>> resnet\_on\_cpu = runtime.computation(ng\_function)

>>> print(resnet\_on\_cpu)

<Computation: resnet50(Parameter\_269)>

# Load an image (or create a mock as in this example)

>>> import numpy as np

>>> picture = np.ones([1, 3, 224, 224], dtype=np.float32)

# Run computation on the picture:

>>> resnet\_on\_cpu(picture)

[array([[2.16105007e-04, 5.58412226e-04, 9.70510227e-05, 5.76671446e-05,

7.45318757e-05, 4.80892748e-04, 5.67404088e-04, 9.48728994e-05,

...

Find more information about nGraph and ONNX in the [nGraph ONNX](https://github.com/NervanaSystems/ngraph-onnx) GitHub repository.

## PaddlePaddle\*[¶](#paddlepaddle)

PaddlePaddle is an open source deep learning framework developed by Baidu. It aims to enable performant large-scale distributed computation for deep learning. The nGraph Compiler stack’s integration to PaddlePaddle respects PaddlePaddle’s design philosophy to minimize switching cost for users. To access nGraph from PaddlePaddle, we’ve added three modules to PaddlePaddle:

* nGraph engine operator (op),
* nGraph engine, and
* nGraph bridge.

The nGraph engine op inherits the PaddlePaddle operator class to allow nGraph engine op to be called using methods consistent with other PaddlePaddle operators. When the nGraph engine is called by the aforementioned op, the nGraph bridge converts PaddlePaddle operators into nGraph operators. nGraph will then build a computational graph based on the converted ops according to the input topology.

### Integration design[¶](#integration-design)

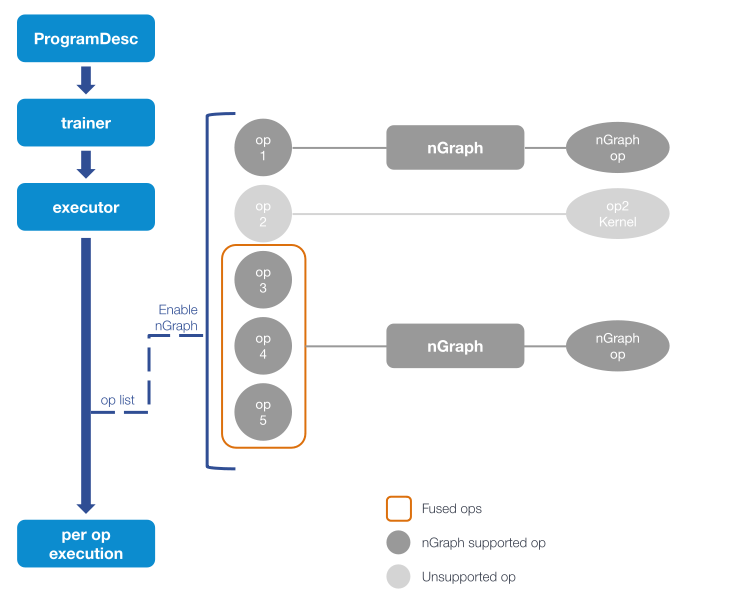
Key design criteria for nGraph-PaddlePaddle integration includes:

1. Minimal intermediate links between nGraph and PaddlePaddle, to reduce latency and improve performance.
2. Close to no switching cost for end users of PaddlePaddle framework.
3. Ease of maintenance.

To satisfy the first design criteria, nGraph designed its operator to match PaddlePaddle’s implementation. nGraph is triggered by the PaddlePaddle executor by one line of code.

After nGraph engine is called, it and the nGraph C++ backend manage all the heavy lifting for performance optimization. The Python frontend on PaddlePaddle remains the same, and end users need **no changes** in the code they write to be able to benefit from the increased performance. This design fulfills the second criteria.

Lastly, the code contributed by nGraph to PaddlePaddle repository mainly resides in the fluid/operator/ngraph directory, and having most of the nGraph code in one place allows for easy maintenance.

[](index.odt/_images/paddlepaddle_design.svg)

*Figure A* above depicts nGraph access from PaddlePaddle. The PaddlePaddle executor generates an executable operator according to the program description (ProgramDesc). nGraph scans the operator sequence before execution, and replaces the supported operators (or subgraphs) with nGraph operators. PaddlePaddle can then execute the nGraph operators and the unreplaced PaddlePaddle operators with a uniform interface. The unreplaced operators are executed by PaddlePaddle native implementation.

nGraph’s current integration reflected on PaddlePaddle’s github repository is organized in the following file structure:

[](index.odt/_images/PaddlePaddleDir.svg)

Compilation of nGraph is handled by the ngraph.cmake file in the cmake/external directory. Other newly-introduced files are located primarily in the paddle/fluid/operator/ngraph directory. The nGraph operators replacing PaddlePaddle operators as described in the previous section can be found in the ngraph/ops directory.

### Integration details[¶](#integration-details)

More details on implementation of nGraph engine op, nGraph engine, and nGraph bridges are provided below:

1. **nGraph engine op**: Triggers subgraphs to be executed by nGraph.
   * Input: Input variable set
   * Output: Output variable set
   * Attribute :
     + Graph: Serialized subgraph. The protobuffer described by PaddlePaddle is serialized and passed to nGraph **as a string**.
     + Interval: The interval of ops in operator list that will be executed by nGraph.
   * Related code :
     + Paddle/fluid/operators/ngraph/ngraph\_engine\_op.h [link to ngraph\_engine\_op header code](https://github.com/PaddlePaddle/Paddle/blob/develop/paddle/fluid/operators/ngraph/ngraph_engine_op.h)
     + Paddle/fluid/operators/ngraph/ngraph\_engine\_op.cc [link to ngraph\_engine\_op cpp code](https://github.com/PaddlePaddle/Paddle/blob/develop/paddle/fluid/operators/ngraph/ngraph_engine_op.cc)
2. **nGraph engine**: calls the nGraph Library to perform calculations.

The nGraph engine class includes the input and output required to build a nGraph function graph from the nGraph engine kernel, the execution function, and the data exchange between nGraph and PaddlePaddle. The primary methods are:

* + BuildNgIO: gets input and output variables.
  + GetNgFunction: obtains the nGraph function used in the calculation. It matches entire pattern of the input to the output and saves functions that need to be called repeatedly.
  + BuildNgFunction: builds nGraph functions.
  + Run: calls backend execution and exchange data with the paddle.
  + Related code :
    - Paddle/fluid/operators/ngraph/ngraph\_engine.h [link to ngraph\_engine header code](https://github.com/PaddlePaddle/Paddle/blob/develop/paddle/fluid/operators/ngraph/ngraph_engine.h)
    - Paddle/fluid/operators/ngraph/ngraph\_engine.cc [link to ngraph\_engine cpp code](https://github.com/PaddlePaddle/Paddle/blob/develop/paddle/fluid/operators/ngraph/ngraph_engine.cc)

1. **nGraph bridge**: converts PaddlePaddle operators to nGraph operators.

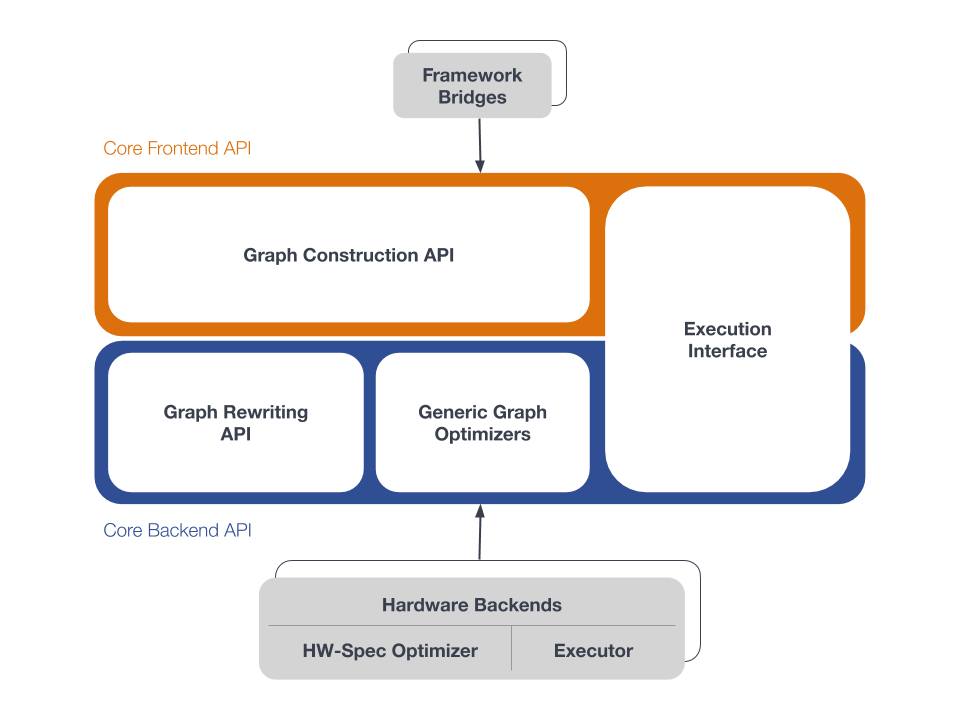
The nGraph bridge converts supported PaddlePaddle operators to nGraph operators, which results in a reconstruction of the subgraph with nGraph’s intermediate representation. The convertable operators are [located in the ngraph ops](https://github.com/PaddlePaddle/Paddle/tree/develop/paddle/fluid/operators/ngraph/ops) directory, and each operator has its own files for easy management. For the conversion of operators. There is a common unified interface to facilitate code development and operator transformation. The relevant interfaces are:

* + GetInputNode: obtains input node for the conversion operator. The nodes are managed through a map.
  + SetOutputNode: sets the constructed node to the map.
  + Related code :
    - Paddle/fluid/operators/ngraph/ngraph\_bridge.h [link to ngraph\_bridge header code](https://github.com/PaddlePaddle/Paddle/blob/develop/paddle/fluid/operators/ngraph/ngraph_bridge.h)
    - Paddle/fluid/operators/ngraph/ngraph\_bridge.cc [link to ngraph\_bridge cpp code](https://github.com/PaddlePaddle/Paddle/blob/develop/paddle/fluid/operators/ngraph/ngraph_bridge.cc)

### nGraph compilation control and trigger method[¶](#ngraph-compilation-control-and-trigger-method)

1. **Compile Control** – The compilation of nGraph is controlled with the WITH\_NGRAPH option. If compiled using WITH\_NGRAPH=ON, the nGraph Library will be downloaded and compiled. This option has a corresponding PADDLE\_WITH\_NGRAPH flag. If compiled WITH\_NGRAPH=OFF, the relevant code will not be compiled.
2. **Trigger Control** – FLAGS\_use\_ngraph triggers nGraph. If this option is set to true, nGraph will be triggered by the PaddlePaddle executor to convert and execute the supported subgraph. Demos are provided under paddle/benchmark/fluid/train/demo (link [train\_demo](https://github.com/PaddlePaddle/Paddle/tree/develop/paddle/fluid/train/demo)) and paddle/benchmark/fluid/train/imdb\_demo (link [imdb\_demo](https://github.com/PaddlePaddle/Paddle/tree/develop/paddle/fluid/train/imdb_demo))

## Basic Concepts[¶](#basic-concepts)

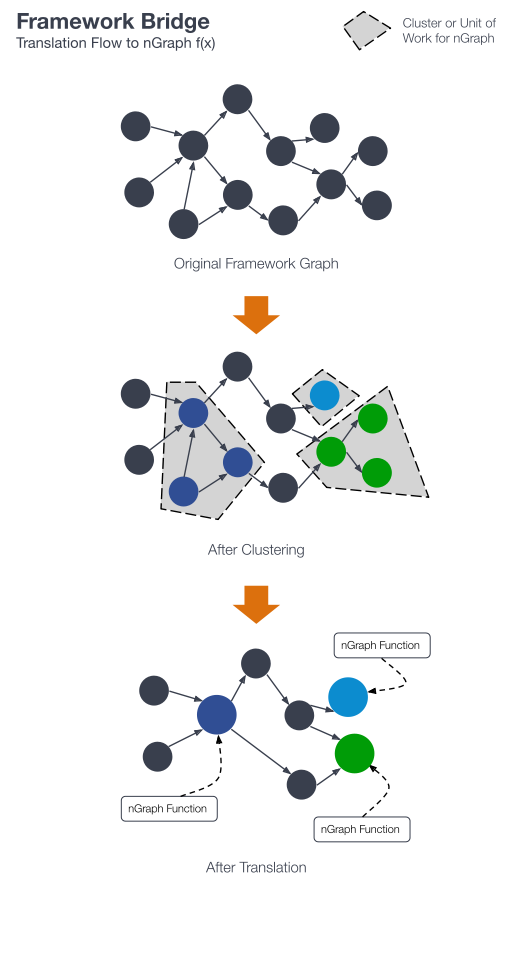


The whole nGraph Compiler stack

The nGraph Compiler stack consists of bridges, core, and backends. We’ll examine each of these briefly to get started.

A framework bridge interfaces with the “frontend” Core API. A framework bridge is a component that sits between a framework like TensorFlow or PaddlePaddle, and the nGraph Core frontend API. A framework bridge does two things: first, it translates a framework’s operations into graphs in nGraph’s in-memory Intermediary Representation. Second, it executes the nGraph IR graphs via the backend execution interface.

The details of bridge implementation vary from framework to framework, but there are some common patterns: a fairly typical example for a graph-based framework is illustrated here, and consists of basically two phases: a **clustering** phase and a **translation** phase.

[](index.odt/_images/overview-translation-flow.svg)

Translation flow to an nGraph function

The clustering phase operates on the original framework’s graph. During this stage, we look for maximal subgraphs containing nodes that can be translated to data flow functions in nGraph. The ability to capture subgraphs of the original graph means that we maintain interoperability with the native framework runtime. Any node that is not placed in a cluster can still by handled by the native framework. On the other hand, identifying maximal subgraphs means that we can avoid unnecessary handoffs between the native framework runtime and nGraph; minimizing this is good for performance.

In the second phase, called translation, we cut out each cluster subgraph, translate it into an nGraph Function, and replace the cluster subgraph with a stand-in node called an “encapsulation node” that holds a pointer to the nGraph Function. Later, at runtime, those functions will be invoked when the framework asks us to execute the encapsulation node.

It’s worth noting that backends have total freedom to rewrite the nGraph Functions: they can do it for the sake of structural or algorithmic optimization of the graph, for easy integration with kernel libraries, or for any or no reason at all.

### Namespaces in nGraph[¶](#namespaces-in-ngraph)

What follows here is a table of all documented namespaces with brief descriptions:

| Namespace | Description | Location in Repo | Docs |
| --- | --- | --- | --- |
| ngraph | The Intel nGraph C++ API | ngraph | Implicit namespace omitted from most API documentation |
| builder | Convenience functions that create additional graph nodes to implement commonly-used recipes; for example, auto-broadcast | [builder](https://github.com/NervanaSystems/ngraph/tree/master/src/ngraph/builder) | Coming Soon |
| descriptor | Descriptors are compile-time representations of objects that will appear at run-time | [descriptor](https://github.com/NervanaSystems/ngraph/tree/master/src/ngraph/descriptor) | Coming Soon |
| op | Ops used in graph construction | [op](https://github.com/NervanaSystems/ngraph/tree/master/src/ngraph/op) | [List of Core ops](index.odt/index.html" \l "document-ops/index) |
| runtime | The objects and methods used for executing the graph | [runtime](https://github.com/NervanaSystems/ngraph/tree/master/src/ngraph/runtime) | [Backend APIs](index.odt/index.html" \l "document-backends/cpp-api) |

## Build and Test[¶](#build-and-test)

* [Building nGraph from source](#default-ngflags)
* [Building nGraph-PlaidML from source](#ngraph-plaidml-backend)

There are a few common paths to take when manually building the nGraph Compiler stack from source code. Today nGraph supports various developers working on all parts of the Deep Learning stack, and the way you decide to build or install components ought to depend on the capabilities of your hardware, and how you intend to use it.

A “from scratch” source-code build of the nGraph Library enables the CPU, Interpreter, and unit tests by default. See [Building nGraph from source](#default-ngflags) for more detail.

A “from scratch” source-code build that defaults to the PlaidML backend contains rich algorithm libraries akin to those that were previously available only to developers willing to spend extensive time writing, testing, and customizing kernels. An NGRAPH\_PLAIDML dist can function like a framework that lets developers compose, train, and even deploy DL models in their preferred language on neural networks of any size. This is a good option if, for example, you are working on a laptop with a high-end GPU that you want to use for compute. See [Building nGraph-PlaidML from source](#ngraph-plaidml-backend) for instructions on how to build.

In either case, there are some prerequisites that your system will need to build from sources.

### Prerequisites[¶](#prerequisites)

| Operating System | Compiler | Build System | Status | Additional Packages |
| --- | --- | --- | --- | --- |
| CentOS 7.4 64-bit | GCC 4.8 | CMake 3.9.0 | supported | wget zlib-devel ncurses-libs ncurses-devel patch diffutils gcc-c++ make git perl-Data-Dumper |
| Ubuntu 16.04 or 18.04 (LTS) 64-bit | Clang 6 | CMake 3.5.1 + GNU Make | supported | build-essential cmake clang-format-6.0 clang-tidy-6.0 clang-6.0 git curl zlib1g zlib1g-dev libtinfo-dev unzip autoconf automake libtool |
| Clear Linux\* OS for Intel® Architecture version 28880+ | Clang 8.0 | CMake 3.14.2 | experimental | bundles machine-learning-basic c-basic python-basic python-basic-dev dev-utils |

#### Building nGraph from source[¶](#building-ngraph-from-source)

**Important**

The default **cmake** procedure (no build flags) will install ngraph\_dist to an OS-level location like /usr/bin/ngraph\_dist or /usr/lib/ngraph\_dist. Here we specify how to build locally to the location of ~/ngraph\_dist with the cmake target -DCMAKE\_INSTALL\_PREFIX=~/ngraph\_dist.

All of the nGraph Library documentation presumes that ngraph\_dist gets installed locally. The system location can be used just as easily by customizing paths on that system. See the ngraph/CMakeLists.txt file to change or customize the default CMake procedure.

* [Ubuntu LTS build steps](#ubuntu)
* [CentOS 7.4 build steps](#centos)

### Ubuntu LTS build steps[¶](#ubuntu-lts-build-steps)

The process documented here will work on Ubuntu\* 16.04 (LTS) or on Ubuntu 18.04 (LTS).

1. Ensure you have installed the [Prerequisites](#prerequisites) for Ubuntu\*.
2. Clone the NervanaSystems ngraph repo:

$ git clone https://github.com/NervanaSystems/ngraph.git

$ cd ngraph

1. Create a build directory outside of the ngraph/src directory tree; somewhere like ngraph/build, for example:

$ mkdir build && cd build

1. Generate the GNU Makefiles in the customary manner (from within the build directory). This command enables ONNX support in the library and sets the target build location at ~/ngraph\_dist, where it can be found easily.

$ cmake .. -DNGRAPH\_ONNX\_IMPORT\_ENABLE=ON -DCMAKE\_INSTALL\_PREFIX=~/ngraph\_dist

**Other optional build flags** – If running gcc-5.4.0 or clang-3.9, remember that you can also append cmake with the prebuilt LLVM option to speed-up the build. Another option if your deployment system has Intel® Advanced Vector Extensions (Intel® AVX) is to target the accelerations available directly by compiling the build as follows during the cmake step: -DNGRAPH\_TARGET\_ARCH=skylake-avx512.

$ cmake .. [-DNGRAPH\_TARGET\_ARCH=skylake-avx512]

1. Run $ make and make install to install libngraph.so and the header files to ~/ngraph\_dist:

$ make # note: make -j <N> may work, but sometimes results in out-of-memory errors if too many compilation processes are used

$ make install

1. (Optional, requires [doxygen](http://www.doxygen.nl/index.html), [Sphinx](http://www.sphinx-doc.org/en/stable/), and [breathe](https://breathe.readthedocs.io/en/latest/)). Run make html inside the doc/sphinx directory of the cloned source to build a copy of the [website docs](https://ngraph.nervanasys.com/docs/latest/) locally. The low-level API docs with inheritance and collaboration diagrams can be found inside the /docs/doxygen/ directory. See the [Contributing to documentation](index.odt/index.html" \l "document-project/doc-contributor-README) for more details about how to build documentation for nGraph.

### CentOS 7.4 build steps[¶](#centos-7-4-build-steps)

The process documented here will work on CentOS 7.4.

1. Ensure you have installed the [Prerequisites](#prerequisites) for CentOS\*, and update the system with **yum**.

$ sudo yum update

1. Install Cmake 3.4:

$ wget https://cmake.org/files/v3.4/cmake-3.5.0.tar.gz

$ tar -xzvf cmake-3.5.0.tar.gz

$ cd cmake-3.5.0

$ ./bootstrap --system-curl --prefix=~/cmake

$ make && make install

1. Clone the NervanaSystems ngraph repo via HTTPS and use Cmake 3.5.0 to build nGraph Libraries to ~/ngraph\_dist. This command enables ONNX support in the library (optional).

$ cd /opt/libraries

$ git clone https://github.com/NervanaSystems/ngraph.git

$ cd ngraph && mkdir build && cd build

$ ~/cmake/bin/cmake .. -DCMAKE\_INSTALL\_PREFIX=~/ngraph\_dist -DNGRAPH\_ONNX\_IMPORT\_ENABLE=ON

$ make && sudo make install

#### Building nGraph-PlaidML from source[¶](#building-ngraph-plaidml-from-source)

The following instructions will create the ~/ngraph\_plaidml\_dist locally:

1. Ensure you have installed the [Prerequisites](#prerequisites) for your OS.
2. Install the prerequisites for the backend. Our hybrid NGRAPH\_PLAIDML backend works best with Python3 versions. We recommend that you use a virtual environment, due to some of the difficulties that users have seen when trying to install outside of a venv.

$ sudo apt install python3-pip

$ pip install plaidml

$ plaidml-setup

1. Clone the source code, create and enter your build directory:

$ git clone https://github.com/NervanaSystems/ngraph.git

$ cd ngraph && mkdir build && cd build

1. Prepare the CMake files as follows:

$ cmake .. -DCMAKE\_INSTALL\_PREFIX=~/ngraph\_plaidml\_dist -DNGRAPH\_CPU\_ENABLE=OFF -DNGRAPH\_PLAIDML\_ENABLE=ON

1. Run **make** and make install. Note that if you are building outside a local or user path, you may need to run make install as the root user.

$ make

$ make install

This should create the shared library libplaidml\_backend.so and nbench. Note that if you built in a virtual environment and run make check from it, the Google Test may report failures. Full tests can be run when PlaidML devices are available at the machine level.

For more about working with the PlaidML backend from nGraph, see our API documentation [PlaidML from nGraph](index.odt/index.html" \l "document-backends/plaidml-ng-api/index).

### macOS\* development[¶](#macos-development)

**Note:** Although we do not currently offer full support for the macOS platform, some configurations and features may work.

The repository includes two scripts (maint/check-code-format.sh and maint/apply-code-format.sh) that are used respectively to check adherence to libngraph code formatting conventions, and to automatically reformat code according to those conventions. These scripts require the command clang-format-3.9 to be in your PATH. Run the following commands (you will need to adjust them if you are not using bash):

$ brew install llvm@3.9 automake

$ mkdir -p $HOME/bin

$ ln -s /usr/local/opt/llvm@3.9/bin/clang-format $HOME/bin/clang-format-3.9

$ echo 'export PATH=$HOME/bin:$PATH' >> $HOME/.bash\_profile

#### Testing the build[¶](#testing-the-build)

We use the [googletest framework](https://github.com/google/googletest.git) from Google for unit tests. The cmake command automatically downloaded a copy of the needed gtest files when it configured the build directory.

To perform unit tests on the install:

1. Create and configure the build directory as described in our [Build and Test](index.odt/index.html" \l "document-buildlb) guide.
2. Enter the build directory and run make check:

$ cd build/

$ make check

## Constructing Graphs[¶](#constructing-graphs)

### Execute a computation[¶](#execute-a-computation)

This section explains how to manually perform the steps that would normally be performed by a framework [bridge](index.odt/index.html" \l "term-bridge) to execute a computation. nGraph graphs are targeted toward automatic construction; it is far easier for a processor (a CPU, GPU, or [purpose-built silicon](https://www.intel.ai/nervana-nnp)) to execute a computation than it is for a human to map out how that computation happens. Unfortunately, things that make by-hand graph construction simpler tend to make automatic construction more difficult, and vice versa.

Nevertheless, it can be helpful to break down what is happening during graph construction. The documetation that follows explains two approaches frameworks can use to compile with nGraph operations:

* [Using complete shapes](#scenario-one)
* [Using partial shapes](#scenario-two)

The nGraph Intermediate Representation uses a strong, dynamic type system, including static shapes. This means that at compilation, every tensor (or, equivalently, every node output) in the graph is assigned **complete shape information**; that is, one and only one shape. The static process by which this assignment takes place is called [shape propagation](index.odt/index.html" \l "term-shape-propagation).

In the [first scenario](#scenario-one), the [model description](index.odt/index.html" \l "term-model-description) walk-through is based on the abc.cpp code in the /doc/examples/abc directory, and it deconstructs the steps that must happen (either programmatically or manually) in order to successfully execute a computation given complete shape information.

#### Scenario One: Using Complete Shapes[¶](#scenario-one-using-complete-shapes)

A step-by-step example of how a framework might execute with complete shape information is provided here. For a step-by-step example using dynamic shapes, see [Scenario Two: Known Partial Shape](#scenario-two).

* [Define the computation](#define-cmp)
* [Specify the backend upon which to run the computation](#specify-backend)
* [Compile the computation](#compile-cmp)
* [Allocate backend storage for the inputs and outputs](#allocate-backend-storage)
* [Initialize the inputs](#initialize-inputs)
* [Invoke the computation](#invoke-cmp)
* [Access the outputs](#access-outputs)

##### Define the computation[¶](#define-the-computation)

To a [framework](index.odt/index.html" \l "term-framework), a computation is simply a transformation of inputs to outputs. While a [bridge](index.odt/index.html" \l "term-bridge) can programmatically construct the graph from a framework’s representation of the computation, graph construction can be somewhat more tedious when done manually. For anyone interested in specific nodes (vertices) or edges of a computation that reveal “what is happening where”, it can be helpful to think of a computation as a zoomed-out and *stateless* [data-flow graph](index.odt/index.html" \l "term-data-flow-graph) where all of the nodes are well-defined tensor operations and all of the edges denote use of an output from one operation as an input for another operation.

Most of the public portion of the nGraph API is in the ngraph namespace, so we will omit the namespace. Use of namespaces other than std will be namespaces in ngraph. For example, the op::Add is assumed to refer to ngraph::op::Add. A computation’s graph is constructed from ops; each is a member of a subclass of op::Op, which, in turn, is a subclass of Node. Not all graphs are computation, but all graphs are composed entirely of instances of Node. Computation graphs contain only op::Op nodes.

We mostly use [shared pointers](index.odt/index.html" \l "term-shared-pointer) for nodes, i.e. std::shared\_ptr<Node>, so that they will be automatically deallocated when they are no longer needed. More detail on shared pointers is given in the glossary.

Every node has zero or more *inputs*, zero or more *outputs*, and zero or more *attributes*.

The specifics for each type permitted on a core Op-specific basis can be discovered in our [List of Core ops](index.odt/index.html" \l "document-ops/index) docs. For our purpose to [define a computation](#define-cmp), nodes should be thought of as essentially immutable; that is, when constructing a node, we need to supply all of its inputs. We get this process started with ops that have no inputs, since any op with no inputs is going to first need some inputs.

op::Parameter specifes the tensors that will be passed to the computation. They receive their values from outside of the graph, so they have no inputs. They have attributes for the element type and the shape of the tensor that will be passed to them.

// Build the graph

Shape s{2, 3};

auto a = std::make\_shared<op::Parameter>(element::f32, s);

auto b = std::make\_shared<op::Parameter>(element::f32, s);

auto c = std::make\_shared<op::Parameter>(element::f32, s);

The above code makes three parameter nodes where each is a 32-bit float of shape (2, 3) and a row-major element layout.

To create a graph for (a + b) \* c, first make an op::Add node with inputs from a and b, and an op::Multiply node from the add node and c:

auto t0 = std::make\_shared<op::Add>(a, b);

auto t1 = std::make\_shared<op::Multiply>(t0, c);

When the op::Add op is constructed, it will check that the element types and shapes of its inputs match; to support multiple frameworks, ngraph does not do automatic type conversion or broadcasting. In this case, they match, and the shape of the unique output of t0 will be a 32-bit float with shape (2, 3). Similarly, op::Multiply checks that its inputs match and sets the element type and shape of its unique output.

Once the graph is built, we need to package it in a Function:

auto f = std::make\_shared<Function>(OutputVector{t1},

ParameterVector{a, b, c});

The first argument to the constuctor specifies the nodes that the function will return; in this case, the product. An OutputVector is a vector of references to outputs of op::Node. The second argument specifies the parameters of the function, in the order they are to be passed to the compiled function. A ParameterVector is a vector of shared pointers to op::Parameter.

Important

The parameter vector must include **every** parameter used in the computation of the results.

##### Specify the backend upon which to run the computation[¶](#specify-the-backend-upon-which-to-run-the-computation)

For a framework bridge, a *backend* is the environment that can perform the computations; it can be done with a CPU, GPU, or [purpose-built silicon](https://www.intel.ai/nervana-nnp). A *transformer* can compile computations for a backend, allocate and deallocate tensors, and invoke computations.

Factory-like managers for classes of backend managers can compile a Function and allocate backends. A backend is somewhat analogous to a multi-threaded process.

There are two backends for the CPU: the optimized "CPU" backend, which uses the [DNNL](https://intel.github.io/mkl-dnn/), and the "INTERPRETER" backend, which runs reference versions of kernels that favor implementation clarity over speed. The "INTERPRETER" backend can be slow, and is primarily intended for testing. See the documentation on [runtime options for various backends](index.odt/index.html" \l "document-backends/index) for additional details.

To continue with our original example and select the "CPU\_Backend":

// Create the backend

auto backend = runtime::Backend::create("CPU");

##### Compile the computation[¶](#compile-the-computation)

Compilation triggers something that can be used as a factory for producing a CallFrame which is a *function* and its associated *state* that can run in a single thread at a time. A CallFrame may be reused, but any particular CallFrame must only be running in one thread at any time. If more than one thread needs to execute the function at the same time, create multiple CallFrame objects from the ExternalFunction.

##### Allocate backend storage for the inputs and outputs[¶](#allocate-backend-storage-for-the-inputs-and-outputs)

At the graph level, functions are stateless. They do have internal state related to execution, but there is no user-visible state. Variables must be passed as arguments. If the function updates variables, it must return the updated variables.

To invoke a function, tensors must be provided for every input and every output. At this time, a tensor used as an input cannot also be used as an output. If variables are being updated, you should use a double-buffering approach where you switch between odd/even generations of variables on each update.

Backends are responsible for managing storage. If the storage is off-CPU, caches are used to minimize copying between device and CPU. We can allocate storage for the three parameters and the return value.

// Allocate tensors for arguments a, b, c

auto t\_a = backend->create\_tensor(element::f32, s);

auto t\_b = backend->create\_tensor(element::f32, s);

auto t\_c = backend->create\_tensor(element::f32, s);

// Allocate tensor for the result

auto t\_result = backend->create\_tensor(element::f32, s);

Each tensor is a shared pointer to a [Tensorview](index.odt/index.html" \l "term-tensorview), which is the interface backends implement for tensor use. When there are no more references to the tensor view, it will be freed when convenient for the backend. See the [Backend APIs](index.odt/index.html" \l "document-backends/cpp-api) documentation for details on how to work with Tensor.

##### Initialize the inputs[¶](#initialize-the-inputs)

Next we need to copy some data into the tensors.

// Initialize tensors

float v\_a[2][3] = {{1, 2, 3}, {4, 5, 6}};

float v\_b[2][3] = {{7, 8, 9}, {10, 11, 12}};

float v\_c[2][3] = {{1, 0, -1}, {-1, 1, 2}};

t\_a->write(&v\_a, sizeof(v\_a));

t\_b->write(&v\_b, sizeof(v\_b));

t\_c->write(&v\_c, sizeof(v\_c));

The runtime::Tensor interface has write and read methods for copying data to/from the tensor.

##### Invoke the computation[¶](#invoke-the-computation)

To invoke the function, we simply pass argument and resultant tensors to the call frame:

// Invoke the function

auto exec = backend->compile(f);

##### Access the outputs[¶](#access-the-outputs)

We can use the read method to access the result:

// Get the result

float r[2][3];

t\_result->read(&r, sizeof(r));

std::cout << "[" << std::endl;

for (size\_t i = 0; i < s[0]; ++i)

{

std::cout << " [";

for (size\_t j = 0; j < s[1]; ++j)

{

std::cout << r[i][j] << ' ';

}

std::cout << ']' << std::endl;

}

std::cout << ']' << std::endl;

return 0;

##### Compiling with Complete Shape Information[¶](#compiling-with-complete-shape-information)

“The (a + b) \* c example for executing a computation on nGraph”[¶](#id1)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78 | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Copyright 2017-2020 Intel Corporation  //  // Licensed under the Apache License, Version 2.0 (the "License");  // you may not use this file except in compliance with the License.  // You may obtain a copy of the License at  //  // http://www.apache.org/licenses/LICENSE-2.0  //  // Unless required by applicable law or agreed to in writing, software  // distributed under the License is distributed on an "AS IS" BASIS,  // WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  // See the License for the specific language governing permissions and  // limitations under the License.  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  #include <iostream>  #include <ngraph/ngraph.hpp>  using namespace ngraph;  int main()  {  // Build the graph  Shape s{2, 3};  auto a = std::make\_shared<op::Parameter>(element::f32, s);  auto b = std::make\_shared<op::Parameter>(element::f32, s);  auto c = std::make\_shared<op::Parameter>(element::f32, s);  auto t0 = std::make\_shared<op::Add>(a, b);  auto t1 = std::make\_shared<op::Multiply>(t0, c);  // Make the function  auto f = std::make\_shared<Function>(OutputVector{t1},  ParameterVector{a, b, c});  // Create the backend  auto backend = runtime::Backend::create("CPU");  // Allocate tensors for arguments a, b, c  auto t\_a = backend->create\_tensor(element::f32, s);  auto t\_b = backend->create\_tensor(element::f32, s);  auto t\_c = backend->create\_tensor(element::f32, s);  // Allocate tensor for the result  auto t\_result = backend->create\_tensor(element::f32, s);  // Initialize tensors  float v\_a[2][3] = {{1, 2, 3}, {4, 5, 6}};  float v\_b[2][3] = {{7, 8, 9}, {10, 11, 12}};  float v\_c[2][3] = {{1, 0, -1}, {-1, 1, 2}};  t\_a->write(&v\_a, sizeof(v\_a));  t\_b->write(&v\_b, sizeof(v\_b));  t\_c->write(&v\_c, sizeof(v\_c));  // Invoke the function  auto exec = backend->compile(f);  exec->call({t\_result}, {t\_a, t\_b, t\_c});  // Get the result  float r[2][3];  t\_result->read(&r, sizeof(r));  std::cout << "[" << std::endl;  for (size\_t i = 0; i < s[0]; ++i)  {  std::cout << " [";  for (size\_t j = 0; j < s[1]; ++j)  {  std::cout << r[i][j] << ' ';  }  std::cout << ']' << std::endl;  }  std::cout << ']' << std::endl;  return 0;  } |

#### Scenario Two: Known Partial Shape[¶](#scenario-two-known-partial-shape)

The [second scenario](#scenario-two) involves the use of dynamic tensors. A [dynamic tensor](index.odt/index.html" \l "term-dynamic-tensor) is a tensor whose shape can change from one “iteration” to the next. When a dynamic tensor is created, a framework [bridge](index.odt/index.html" \l "term-bridge) might supply only *partial* shape information: it might be **all** the tensor dimensions, **some** of the tensor dimensions, or **none** of the tensor dimensions; furthermore, the rank of the tensor may be left unspecified. The “actual” shape of the tensor is not specified until some function writes some value to it. The actual shape can change when the value of the tensor is overwritten. It is the backend’s responsibility to set the actual shape. The [model description](index.odt/index.html" \l "term-model-description) for the second scenario based on the partial\_shape.cpp code in the /doc/examples/dynamic\_tensor directory, and it deconstructs the steps that must happen (either programmatically or manually) in order to successfully retreive shape data.

* [Create a dynamic tensor](#create-dyn-tensor)
* [Initialize input of shape](#call-graph-vw)
* [Get the result](#dyn-ten-result)
* [Compiling with Known Partial Shape](#kpsh)

Create and compile a graph where the provided info of shape x is (2,?):

auto x\_shape\_info = PartialShape{2, Dimension::dynamic()};

auto x = make\_shared<op::Parameter>(element::i32, x\_shape\_info);

auto a = x + x;

auto f = make\_shared<Function>(OutputVector{a}, ParameterVector{x});

auto be = runtime::Backend::create("CPU", true);

auto ex = be->compile(f);

##### Create a dynamic tensor[¶](#create-a-dynamic-tensor)

Create a dynamic tensor of shape (2,?)

auto t\_out = be->create\_dynamic\_tensor(element::i32, x\_shape\_info);

execute(be, ex, t\_out, 3);

execute(be, ex, t\_out, 11);

execute(be, ex, t\_out, 20);

At this point, t\_out->get\_shape() would throw an exception, while t\_out->get\_partial\_shape() would return "(2,?)".

##### Initialize input of shape[¶](#initialize-input-of-shape)

auto t\_in = be->create\_tensor(element::i32, Shape{2, n});

{

vector<int32\_t> t\_val(2 \* n);

iota(t\_val.begin(), t\_val.end(), 0);

t\_in->write(&t\_val[0], t\_val.size() \* sizeof(t\_val[0]));

}

At this point, t\_out->get\_shape() would return Shape{2,3}, while t\_out->get\_partial\_shape() would return "(2,?)".

##### Get the result[¶](#get-the-result)

ex->call({t\_out}, {t\_in});

auto s = t\_out->get\_shape();

vector<int32\_t> r(s[0] \* s[1]);

t\_out->read(&r[0], r.size() \* sizeof(r[0]));

cout << "[" << endl;

for (size\_t i = 0; i < s[0]; ++i)

{

cout << " [";

for (size\_t j = 0; j < s[1]; ++j)

{

cout << r[i \* s[1] + j] << ' ';

}

cout << ']' << endl;

}

cout << ']' << endl;

}

At this point, t\_out->get\_shape() would return Shape{2,20}, while t\_out->get\_partial\_shape() would return "(2,?)".

##### Compiling with Known Partial Shape[¶](#compiling-with-known-partial-shape)

“Full code for compiling with dynamic tensors and partial shape”[¶](#id2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80 | //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Copyright 2017-2020 Intel Corporation  //  // Licensed under the Apache License, Version 2.0 (the "License");  // you may not use this file except in compliance with the License.  // You may obtain a copy of the License at  //  // http://www.apache.org/licenses/LICENSE-2.0  //  // Unless required by applicable law or agreed to in writing, software  // distributed under the License is distributed on an "AS IS" BASIS,  // WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  // See the License for the specific language governing permissions and  // limitations under the License.  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  #include <iostream>  #include <numeric>  #include <vector>  #include <ngraph/ngraph.hpp>  using namespace std;  using namespace ngraph;  void execute(shared\_ptr<runtime::Backend> be,  shared\_ptr<runtime::Executable> ex,  shared\_ptr<runtime::Tensor> t\_out,  uint32\_t n);  int main()  {  // Create and compile a graph where the provided info of shape of x is  // (2,?)  auto x\_shape\_info = PartialShape{2, Dimension::dynamic()};  auto x = make\_shared<op::Parameter>(element::i32, x\_shape\_info);  auto a = x + x;  auto f = make\_shared<Function>(OutputVector{a}, ParameterVector{x});  auto be = runtime::Backend::create("CPU", true);  auto ex = be->compile(f);  // Create a dynamic tensor of shape (2,?)  auto t\_out = be->create\_dynamic\_tensor(element::i32, x\_shape\_info);  execute(be, ex, t\_out, 3);  execute(be, ex, t\_out, 11);  execute(be, ex, t\_out, 20);  return 0;  }  void execute(shared\_ptr<runtime::Backend> be,  shared\_ptr<runtime::Executable> ex,  shared\_ptr<runtime::Tensor> t\_out,  uint32\_t n)  {  // Initialize input of shape (2, n)  auto t\_in = be->create\_tensor(element::i32, Shape{2, n});  {  vector<int32\_t> t\_val(2 \* n);  iota(t\_val.begin(), t\_val.end(), 0);  t\_in->write(&t\_val[0], t\_val.size() \* sizeof(t\_val[0]));  }  // Get the result  ex->call({t\_out}, {t\_in});  auto s = t\_out->get\_shape();  vector<int32\_t> r(s[0] \* s[1]);  t\_out->read(&r[0], r.size() \* sizeof(r[0]));  cout << "[" << endl;  for (size\_t i = 0; i < s[0]; ++i)  {  cout << " [";  for (size\_t j = 0; j < s[1]; ++j)  {  cout << r[i \* s[1] + j] << ' ';  }  cout << ']' << endl;  }  cout << ']' << endl;  } |

### Build a graph with operators[¶](#build-a-graph-with-operators)

This section illustrates the use of C++ operators to simplify the building of graphs.

Several C++ operators are overloaded to simplify graph construction. For example, the following:

auto t1 = std::make\_shared<op::Multiply>(t0, c);

can be simplified to:

auto t1 = (a + b) \* c;

The expression a + b is equivalent to std::make\_shared<op::Add>(a, b) and the \* operator similarly returns std::make\_shared<op::Multiply> to its arguments.

### Make a stateful computation[¶](#make-a-stateful-computation)

In this section, we show how to make a stateful computation from nGraph’s stateless operations. The basic idea is that any computation with side-effects can be factored into a stateless function that transforms the old state into the new state.

#### An example from C++[¶](#an-example-from-c)

Let’s start with a simple C++ example, a function count that returns how many times it has already been called:

update.cpp[¶](#id1)

int count()

{

static int counter = 0;

return counter++;

}

The static variable counter provides state for this function. The state is initialized to 0. Every time count is called, the current value of counter is returned and counter is incremented. To convert this to use a stateless function, define a function that takes the current value of counter as an argument and returns the updated value.

std::tuple<int, int> stateless\_count(int counter)

{

return std::tuple<int, int>(counter, counter + 1);

}

To use this version of counting,

int counter = 0;

{

auto r(stateless\_count(counter));

counter = std::get<1>(r);

std::cout << std::get<0>(r);

}

std::cout << ", ";

{

auto r(stateless\_count(counter));

counter = std::get<1>(r);

std::cout << std::get<0>(r);

}

std::cout << std::endl;

#### Update in nGraph[¶](#update-in-ngraph)

In working with nGraph-based construction of graphs, updating takes the same approach. During training, we include all the weights as arguments to the training function and return the updated weights along with any other results. For more complex forms of training, such as those using momentum, we would add the momentum tensors as additional arguments and include their updated values as additional results. A simple case is illustrated in the documentation for how to [Derive a trainable model](index.odt/index.html" \l "document-core/constructing-graphs/derive-for-training).

### Derive a trainable model[¶](#derive-a-trainable-model)

Documentation in this section describes one of the possible ways to turn a DL model for inference into one that can be used for training.

Additionally, and to provide a more complete walk-through that *also* trains the model, our example includes the use of a simple data loader for uncompressed MNIST data.

* [Model overview](#model-overview)
* [Code structure](#code-structure)
  + [Inference](#inference)
  + [Loss](#loss)
  + [Backprop](#backprop)
  + [Update](#update)

#### Automating graph construction[¶](#automating-graph-construction)

In a Machine Learning ecosystem, it makes sense to use automation and abstraction where possible. nGraph was designed to automatically use the “ops” of tensors provided by a framework when constructing graphs. However, nGraph’s graph-construction API operates at a fundamentally lower level than a typical framework’s API, and writing a model directly in nGraph would be somewhat akin to programming in assembly language: not impossible, but not the easiest thing for humans to do.

To make the task easier for developers who need to customize the “automatic”, construction of graphs, we’ve provided some demonstration code for how this could be done. We know, for example, that a trainable model can be derived from any graph that has been constructed with weight-based updates.

The following example named mnist\_mlp.cpp represents a hand-designed inference model being converted to a model that can be trained with nGraph.

#### Model overview[¶](#model-overview)

Due to the lower-level nature of the graph-construction API, the example we’ve selected to document here is a relatively simple model: a fully-connected topology with one hidden layer followed by Softmax.

Remember that in nGraph, the graph is stateless; values for the weights must be provided as parameters along with the normal inputs. Starting with the graph for inference, we will use it to create a graph for training. The training function will return tensors for the updated weights.

Note

This example illustrates how to convert an inference model into one that can be trained. Depending on the framework, bridge code may do something similar, or the framework might do this operation itself. Here we do the conversion with nGraph because the computation for training a model is significantly larger than for inference, and doing the conversion manually is tedious and error-prone.

#### Code structure[¶](#code-structure)

##### Inference[¶](#inference)

We begin by building the graph, starting with the input parameter X. We also define a fully-connected layer, including parameters for weights and bias:

auto X = std::make\_shared<op::Parameter>(

element::f32, Shape{batch\_size, input\_size});

// Layer 0

auto W0 = std::make\_shared<op::Parameter>(element::f32,

Shape{input\_size, l0\_size});

auto b0 =

std::make\_shared<op::Parameter>(element::f32, Shape{l0\_size});

auto l0\_dot = std::make\_shared<op::Dot>(X, W0, 1);

auto b0\_broadcast = std::make\_shared<op::Broadcast>(

b0, Shape{batch\_size, l0\_size}, AxisSet{0});

auto l0 = std::make\_shared<op::Relu>(l0\_dot + b0\_broadcast);

Repeat the process for the next layer,

auto W1 = std::make\_shared<op::Parameter>(element::f32,

Shape{l0\_size, l1\_size});

auto b1 =

std::make\_shared<op::Parameter>(element::f32, Shape{l1\_size});

auto l1\_dot = std::make\_shared<op::Dot>(l0, W1, 1);

auto b1\_broadcast = std::make\_shared<op::Broadcast>(

b1, Shape{batch\_size, l1\_size}, AxisSet{0});

auto l1 = l1\_dot + b1\_broadcast;

and normalize everything with a softmax.

// Softmax

auto softmax = std::make\_shared<op::Softmax>(l1, AxisSet{1});

##### Loss[¶](#loss)

We use cross-entropy to compute the loss. nGraph does not currenty have a core op for cross-entropy, so we implement it directly, adding clipping to prevent underflow.

auto Y =

std::make\_shared<op::Parameter>(element::f32, Shape{batch\_size});

auto labels =

std::make\_shared<op::OneHot>(Y, Shape{batch\_size, output\_size}, 1);

auto softmax\_clip\_value = std::make\_shared<op::Constant>(

element::f32, Shape{}, std::vector<float>{log\_min});

auto softmax\_clip\_broadcast = std::make\_shared<op::Broadcast>(

softmax\_clip\_value, Shape{batch\_size, output\_size}, AxisSet{0, 1});

auto softmax\_clip =

std::make\_shared<op::Maximum>(softmax, softmax\_clip\_broadcast);

auto softmax\_log = std::make\_shared<op::Log>(softmax\_clip);

auto prod = std::make\_shared<op::Multiply>(softmax\_log, labels);

auto N = std::make\_shared<op::Parameter>(element::f32, Shape{});

auto loss = std::make\_shared<op::Divide>(

std::make\_shared<op::Sum>(prod, AxisSet{0, 1}), N);

##### Backprop[¶](#backprop)

We want to reduce the loss by adjusting the weights. We compute the adjustments using the reverse-mode autodiff algorithm, commonly referred to as “backprop” because of the way it is implemented in interpreted frameworks. In nGraph, we augment the loss computation with computations for the weight adjustments. This allows the calculations for the adjustments to be further optimized.

// Each of W0, b0, W1, and b1

auto learning\_rate =

std::make\_shared<op::Parameter>(element::f32, Shape{});

auto delta = -learning\_rate \* loss;

For any node N, if the update for loss is delta, the update computation for N will be given by the node

auto update = loss->backprop\_node(N, delta);

auto W1\_next = W1 + adjoints.backprop\_output(W1);

auto b1\_next = b1 + adjoints.backprop\_output(b1);

The different update nodes will share intermediate computations. So to get the updated values for the weights as computed with the specified [backend](index.odt/index.html" \l "document-backends/index):

auto t\_W0 = make\_output\_tensor(backend, W0, 0);

auto t\_b0 = make\_output\_tensor(backend, b0, 0);

auto t\_W1 = make\_output\_tensor(backend, W1, 0);

auto t\_b1 = make\_output\_tensor(backend, b1, 0);

std::function<float()> rand(

std::bind(std::uniform\_real\_distribution<float>(-1.0f, 1.0f),

std::default\_random\_engine(0)));

randomize(rand, t\_W0);

randomize(rand, t\_b0);

randomize(rand, t\_W1);

randomize(rand, t\_b1);

// Allocate inputs

auto t\_X = make\_output\_tensor(backend, X, 0);

auto t\_Y = make\_output\_tensor(backend, Y, 0);

auto t\_learning\_rate = make\_output\_tensor(backend, learning\_rate, 0);

auto t\_N = make\_output\_tensor(backend, N, 0);

set\_scalar(t\_N, static\_cast<float>(batch\_size), 0);

// Allocate updated variables

auto t\_W0\_next = make\_output\_tensor(backend, W0\_next, 0);

auto t\_b0\_next = make\_output\_tensor(backend, b0\_next, 0);

auto t\_W1\_next = make\_output\_tensor(backend, W1\_next, 0);

auto t\_b1\_next = make\_output\_tensor(backend, b1\_next, 0);

auto t\_loss = make\_output\_tensor(backend, loss, 0);

auto t\_softmax = make\_output\_tensor(backend, softmax, 0);

##### Update[¶](#update)

Since nGraph is stateless, we train by making a function that has the original weights among its inputs and the updated weights among the results. For training, we’ll also need the labeled training data as inputs, and we’ll return the loss as an additional result. We’ll also want to track how well we are doing; this is a function that returns the loss and has the labeled testing data as input. Although we can use the same nodes in different functions, nGraph currently does not allow the same nodes to be compiled in different functions, so we compile clones of the nodes.

NodeMap train\_node\_map;

auto train\_function = clone\_function(

Function(

OutputVector{

loss, softmax, W0\_next, b0\_next, W1\_next, b1\_next},

ParameterVector{X, Y, N, learning\_rate, W0, b0, W1, b1}),

train\_node\_map);

auto train\_exec = backend->compile(train\_function);

### Distribute training across multiple nGraph backends[¶](#distribute-training-across-multiple-ngraph-backends)

Important

Distributed training is not officially supported in version 0.29; however, the following configuration options have worked for nGraph devices with mixed or limited success in testing.

In the [previous section](index.odt/index.html" \l "document-core/constructing-graphs/derive-for-training), we described the steps needed to create a “trainable” nGraph model. Here we demonstrate how to train a data parallel model by distributing the graph to more than one device.

Frameworks can implement distributed training with nGraph versions prior to 0.13:

* Use -DNGRAPH\_DISTRIBUTED\_ENABLE=OMPI to enable distributed training with OpenMPI. Use of this flag requires that OpenMPI be a pre-existing library in the system. If it’s not present on the system, install [OpenMPI](https://www.open-mpi.org/software/ompi/v2.1/) version 2.1.1 or later before running the compile.
* Use -DNGRAPH\_DISTRIBUTED\_ENABLE=MLSL to enable the option for Intel® Machine Learning Scaling Library for Linux\* OS:

Note

The Intel® MLSL option applies to Intel® Architecture CPUs (CPU) and Interpreter backends only. For all other backends, OpenMPI is presently the only supported option. We recommend the use of Intel MLSL for CPU backends to avoid an extra download step.

Finally, to run the training using two nGraph devices, invoke mpirun.

To deploy data-parallel training, the AllReduce op should be added after the steps needed to complete the [backpropagation](index.odt/index.html" \l "document-core/constructing-graphs/derive-for-training); the new code is highlighted below:

ngraph::autodiff::Adjoints adjoints(OutputVector{loss},

OutputVector{delta});

auto grad\_W0 = adjoints.backprop\_output(W0);

auto grad\_b0 = adjoints.backprop\_output(b0);

auto grad\_W1 = adjoints.backprop\_output(W1);

auto grad\_b1 = adjoints.backprop\_output(b1);

auto avg\_grad\_W0 = std::make\_shared<op::AllReduce>(grad\_W0);

auto avg\_grad\_b0 = std::make\_shared<op::AllReduce>(grad\_b0);

auto avg\_grad\_W1 = std::make\_shared<op::AllReduce>(grad\_W1);

auto avg\_grad\_b1 = std::make\_shared<op::AllReduce>(grad\_b1);

auto W0\_next = W0 + avg\_grad\_W0;

auto b0\_next = b0 + avg\_grad\_b0;

auto W1\_next = W1 + avg\_grad\_W1;

auto b1\_next = b1 + avg\_grad\_b1;

See the [full code](https://github.com/NervanaSystems/ngraph/blob/master/doc/examples/mnist_mlp/dist_mnist_mlp.cpp) in the examples folder /doc/examples/mnist\_mlp/dist\_mnist\_mlp.cpp.

mpirun -np 2 dist\_mnist\_mlp

### Import a model[¶](#import-a-model)

[Importing a model from ONNX](#from-onnx)

nGraph APIs can be used to run inference on a model that has been *exported* from a Deep Learning framework. An export produces a file with a serialized model that can be loaded and passed to one of the nGraph backends.

#### Importing a model from ONNX[¶](#importing-a-model-from-onnx)

The most-widely supported [export](index.odt/index.html" \l "term-export) format available today is [ONNX](http://onnx.ai/). Models that have been serialized to ONNX are easy to identify; they are usually named <some\_model>.onnx or <some\_model>.onnx.pb. These [tutorials from ONNX](https://github.com/onnx/tutorials) describe how to turn trained models into an .onnx export.

Important

If you landed on this page and you already have an .onnx or an .onnx.pb formatted file, you should be able to run the inference without needing to dig into anything from the “Frameworks” sections. You will, however, need to have completed the steps outlined in our [Build and Test](index.odt/index.html" \l "document-buildlb) guide.

To demonstrate functionality, we’ll use an already-serialized CIFAR10 model trained via ResNet20. Remember that this model has already been trained and exported from a framework such as Caffe2, PyTorch or CNTK; we are simply going to build an nGraph representation of the model, execute it, and produce some outputs.

#### Installing ngraph\_onnx with nGraph from scratch[¶](#installing-ngraph-onnx-with-ngraph-from-scratch)

See the documentation on: [building nGraph and nGraph-ONNX](https://github.com/NervanaSystems/ngraph-onnx/blob/master/BUILDING.md) for the latest instructions.

#### Importing a serialized model[¶](#importing-a-serialized-model)

After building and installing ngraph\_onnx, we can import a model that has been serialized by ONNX, interact locally with the model by running Python code, create and load objects, and run inference.

This section assumes that you have your own ONNX model. With this example model from Microsoft\*’s Deep Learning framework, [CNTK](https://www.microsoft.com/en-us/cognitive-toolkit/features/model-gallery/), we can outline the procedure to show how to run ResNet on model that has been trained on the CIFAR10 data set and serialized with ONNX.

##### (Optional) Localize your export to the virtual environment[¶](#optional-localize-your-export-to-the-virtual-environment)

For this example, let’s say that our serialized file was output under our $HOME directory, say at ~/onnx\_conversions/trained\_model.onnx. To make loading this file easier, you can run the example below from your Venv in that directory. If you invoke your python interpreter in a different directory, you will need to specify the relative path to the location of the .onnx file.

Important

If you invoke your Python interpreter in directory other than where you outputted your trained model, you will need to specify the **relative** path to the location of the .onnx file.

(onnx) $ cd ~/onnx\_conversions

(onnx) $ python3

##### Enable ONNX and load an ONNX file from disk[¶](#enable-onnx-and-load-an-onnx-file-from-disk)

import onnx

onnx\_protobuf = onnx.load('/path/to/model/cntk\_ResNet20\_CIFAR10/model.onnx')

##### Convert an ONNX model to an ngraph model[¶](#convert-an-onnx-model-to-an-ngraph-model)

from ngraph\_onnx.onnx\_importer.importer import import\_onnx\_model

ng\_model = import\_onnx\_model(onnx\_protobuf)[0]

The importer returns a list of ngraph models for every ONNX graph output:

print(ng\_models)

[{

'name': 'Plus5475\_Output\_0',

'output': <Add: 'Add\_1972' ([1, 10])>,

'inputs': [<Parameter: 'Parameter\_1104' ([1, 3, 32, 32], float)>]

}]

The output field contains the ngraph node corrsponding to the output node in the imported ONNX computational graph. The inputs list contains all input parameters for the computation which generates the output.

##### Using ngraph\_api, create a callable computation object[¶](#using-ngraph-api-create-a-callable-computation-object)

import ngraph as ng

runtime = ng.runtime(backend\_name='CPU')

resnet = runtime.computation(ng\_model['output'], \*ng\_model['inputs'])

##### Load or create an image[¶](#load-or-create-an-image)

import numpy as np

picture = np.ones([1, 3, 32, 32])

##### Run ResNet inference on picture[¶](#run-resnet-inference-on-picture)

resnet(picture)

#### Put it all together[¶](#put-it-all-together)

“Demo sample code to run inference with nGraph”[¶](#id1)

import onnx

onnx\_protobuf = onnx.load('/path/to/model/cntk\_ResNet20\_CIFAR10/model.onnx')

# Convert a serialized ONNX model to an ngraph model

from ngraph\_onnx.onnx\_importer.importer import import\_onnx\_model

ng\_model = import\_onnx\_model(onnx\_protobuf)[0]

# Using an ngraph runtime (CPU backend), create a callable computation

import ngraph as ng

runtime = ng.runtime(backend\_name='CPU')

resnet = runtime.computation(ng\_model['output'], \*ng\_model['inputs'])

# Load or create an image

import numpy as np

picture = np.ones([1, 3, 32, 32])

# Run ResNet inference on picture

resnet(picture)

Outputs will vary greatly, depending on your model; for demonstration purposes, the code will look something like:

array([[ 1.312082 , -1.6729496, 4.2079577, 1.4012241, -3.5463796,

2.3433776, 1.7799224, -1.6155214, 0.0777044, -4.2944093]],

dtype=float32)

### Python API[¶](#python-api)

This section contains the Python API component of the nGraph Compiler stack. The Python API exposes nGraph™ C++ operations to Python users. For quick-start you can find an example of the API usage below.

Note that the output at print(model) may vary; it varies according to the number of nodes or variety of [step](index.odt/index.html" \l "term-step) used to compute the printed solution. Various NNs configured in different ways should produce the same result for simple calculations or accountings. More complex computations may have minor variations with respect to how precise they ought to be. For example, a more efficient graph <Multiply: 'Multiply\_12' ([2, 2])> can also be achieved with some configurations.

“Basic example”[¶](#id1)

import numpy as np

import ngraph as ng

A = ng.parameter(shape=[2, 2], name='A', dtype=np.float32)

B = ng.parameter(shape=[2, 2], name='B')

C = ng.parameter(shape=[2, 2], name='C')

# >>> print(A)

# <Parameter: 'A' ([2, 2], float)>

model = (A + B) \* C

# >>> print(model)

# <Multiply: 'Multiply\_14' ([2, 2])>

runtime = ng.runtime(backend\_name='CPU')

# >>> print(runtime)

# <Runtime: Backend='CPU'>

computation = runtime.computation(model, A, B, C)

# >>> print(computation)

# <Computation: Multiply\_14(A, B, C)>

value\_a = np.array([[1, 2], [3, 4]], dtype=np.float32)

value\_b = np.array([[5, 6], [7, 8]], dtype=np.float32)

value\_c = np.array([[9, 10], [11, 12]], dtype=np.float32)

result = computation(value\_a, value\_b, value\_c)

# >>> print(result)

# [[ 54. 80.]

# [110. 144.]]

#### nGraph Python APIs[¶](#ngraph-python-apis)

|  |  |
| --- | --- |
| [ngraph](index.odt/index.html" \l "module-ngraph) | ngraph module namespace, exposing factory functions for all ops and other classes. |
| [ngraph.exceptions](index.odt/index.html" \l "module-ngraph.exceptions) | ngraph exceptions hierarchy. |
| [ngraph.ops](index.odt/index.html" \l "module-ngraph.ops) | Factory functions for all ngraph ops. |
| [ngraph.runtime](index.odt/index.html" \l "module-ngraph.runtime) | Create a Runtime object (helper factory). |

The “How to” articles in this section explain how to build or construct graphs with nGraph components. The recipes are all framework agnostic; in other words, if an entity (framework or user) wishes to make use of target-based computational resources, it can either:

* Do the tasks programatically through a framework, or
* Provide a serialized model that can be imported to run on one of the nGraph backends.

Note

This section is aimed at intermediate-level developers. It assumes an understanding of the concepts in the previous sections. It does not assume knowledge of any particular frontend framework.

## Compiler Passes[¶](#compiler-passes)

### List of passes[¶](#list-of-passes)

The kinds of compiler passes available can be broken down into different buckets:

#### Graph Optimization Passes[¶](#graph-optimization-passes)

| Graph Optimization Passes | More Detail |
| --- | --- |
| AlgebraicSimplification | [Algebraic Simplification](#algebraic-simpl) |
| CommonSubexpressionElimination | [Common Subexpression Elimination](#common-subex-elim) |
| ConstantFolding | [Constant Folding](#constant-fold) |
| CoreFusion | [Core Fusion](#core-fusion) |
| ReshapeElimination | [Reshape Elimination](#reshape-transpose-elim) |
| ReshapeSinking | [Reshape Sinking](#reshape-transpose-sink) |

#### Node Optimization Passes[¶](#node-optimization-passes)

| Node Optimization Passes | More Detail |
| --- | --- |
| NopElimination |  |
| ZeroDimTensorElimination |  |

#### Memory Assignment Passes[¶](#memory-assignment-passes)

| Memory Assignment Passes | More Detail |
| --- | --- |
| AssignLayout |  |
| Liveness |  |
| MemoryLayout |  |
| PropagateCacheability |  |

#### Codegen Passes[¶](#codegen-passes)

Important

Codegen is currently experimental only.

| Codegen Passes | More Detail |
| --- | --- |
| CommonFunctionCollection | Experimental Only |

#### Debug Passes[¶](#debug-passes)

| Debug Passes | More Detail |
| --- | --- |
| DumpSorted |  |
| MemoryVisualize |  |
| Serialization |  |
| VisualizeTree |  |

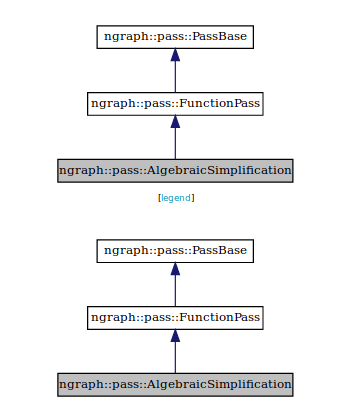
#### Maintenance Passes[¶](#maintenance-passes)

| Maintenance Passes | More Detail |
| --- | --- |
| GetOutputElementElimination |  |
| LikeReplacement |  |
| ValidateGraph |  |

Important

All of the above passes are currently implementable; more detailed documentation for each pass may be a Work In Progress (WIP).

##### Algebraic Simplification[¶](#algebraic-simplification)

[](index.odt/_images/algebraic-simpl.png)

Algebraic simplification

The **Algebraic Simplification** pass implements what amounts to a “grab bag” of algebraic simplification rules. It does some basic things like rewrite “zero times x” to simply “zero”, or “zero plus x” to plain “x”.

It can also do a number of tricks more specific to deep learning. For example, if we discover that a tensor is being sliced up by adjacent segments, only to have those slices concatenated back together again, we can skip the slicing and concatting altogether. Or, if a tensor is being padded, but the actual width of the padding is zero all around, we can skip the padding step entirely.

Several other transformations like this are implemented in the algebraic simplification pass. And while none of these transformations might seem particularly impressive on their own, when everything comes together the results of this pass often yield improvement even on the initial graph straight out of the bridge. This pass is also quite important as a “glue” pass that can be used to clean up and/or re-simplify after other passes have done their own tricks. See the example on [Compiler Passes](index.odt/index.html" \l "document-core/passes/passes) for an example of how effective this can be.

##### Common Subexpression Elimination[¶](#common-subexpression-elimination)

##### Constant Folding[¶](#constant-folding)

##### Core Fusion[¶](#core-fusion)

##### Reshape Elimination[¶](#reshape-elimination)

The pass also called **Reshape/Transpose Elimination** will find and optimize where we can “push” two Transpose ops through a matrix multiplication. For example, if you have two matrices (say, *foo* and *bar*), both of these matrices will be transposed (to produce *foo.t* and *bar.t*, respectively), after which *foo.t* and *bar.t* get multiplied together.

Often a more efficient way to implement this is to switch the order of the arguments *foo* and *bar*, multiply them together, and then transpose the output of the matmul. Effectively, this cuts two Transpose operations down to just one, where the **Reshape/Transpose** elimination will do that rewrite for you.

Another common pattern that can be optimized via nGraph is the case where two transpositions cancel each other out. One example of this is taking the “Transpose” of the transpose of a matrix, though actually a more common case is when the graph is translating among different batch formats. We can often move these operations around through a process called **Reshape sinking/swimming**, and in cases where two transposes wind up canceling each other out, we can cut them both out of the graph.

##### Reshape Sinking[¶](#reshape-sinking)

##### Zero-Element Tensor Elimination[¶](#zero-element-tensor-elimination)

### Passes that use Matcher[¶](#passes-that-use-matcher)

* CPUFusion (GraphRewrite)
* CoreFusion (GraphRewrite)
* ReshapeElimination (GraphRewrite)
* AlgebraicSimplification
* CPUPostLayoutOptimizations (GraphRewrite)
* CPURnnMatFusion
* and many more…

#### Register simplify\_neg handler[¶](#register-simplify-neg-handler)

static std::unordered\_map<std::type\_index, std::function<bool(std::shared\_ptr<Node>)>>

initialize\_const\_values\_to\_ops()

{

return std::unordered\_map<std::type\_index, std::function<bool(std::shared\_ptr<Node>)>>({

{TI(op::Add), simplify\_add},

{TI(op::Multiply), simplify\_multiply},

{TI(op::Sum), simplify\_sum},

{TI(op::Negative), simplify\_neg}

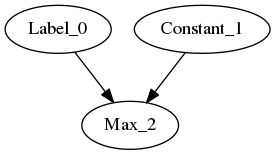
});

}

##### Add a fusion[¶](#add-a-fusion)

max(0, A) = Relu(A)

##### Pattern for capturing[¶](#pattern-for-capturing)



max(0, A) = Relu(A)

namespace ngraph

{

namespace pass

{

class CoreFusion;

}

}

class ngraph::pass::CoreFusion : public ngraph::pass::GraphRewrite

{

public:

CoreFusion()

: GraphRewrite()

{

construct\_relu\_pattern();

}

//this should go in a cpp file.

void construct\_relu\_pattern()

{

auto iconst0 = ngraph::make\_zero(element::i32, Shape{});

auto val = make\_shared(iconst0);

auto zero = make\_shared(iconst0, nullptr, NodeVector{iconst0});

auto broadcast\_pred = [](std::shared\_ptr n) {

return static\_cast(std::dynamic\_pointer\_cast(n));

};

auto skip\_broadcast = std::make\_shared(zero, broadcast\_pred);

auto max = make\_shared(skip\_broadcast, val);

pattern::graph\_rewrite\_callback callback = [val, zero](pattern::Matcher& m) {

NGRAPH\_DEBUG << "In a callback for construct\_relu\_pattern against "

<< m.get\_match\_root()->get\_name();

auto pattern\_map = m.get\_pattern\_map();

auto mzero = m.get\_pattern\_map()[zero];

if (!ngraph::is\_zero(mzero))

{

NGRAPH\_DEBUG << "zero constant = " << mzero->get\_name() << " not equal to 0n";

return false;

}

auto mpattern = m.get\_match\_root();

auto cg = shared\_ptr(new op::Relu(pattern\_map[val]));

ngraph::replace\_node(m.get\_match\_root(), cg);

return true;

};

auto m = make\_shared(max, callback);

this->add\_matcher(m);

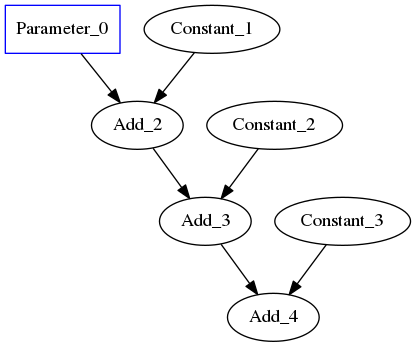
}

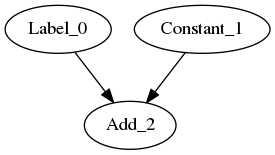
};

#### Recurrent patterns[¶](#recurrent-patterns)

Equivalent to "A(BC)+A" in regexes

(((A + 0) + 0) + 0) = A





Shape shape{};

auto a = make\_shared<op::Parameter>(element::i32, shape);

auto b = make\_shared<op::Parameter>(element::i32, shape);

auto rpattern = std::make\_shared<pattern::op::Label>(b);

auto iconst0 = ngraph::make\_zero(element::i32, shape);

auto abs = make\_shared<op::Abs>(a);

auto add1 = iconst0 + b;

auto add2 = iconst0 + add1;

auto add3 = iconst0 + add2;

auto padd = iconst0 + rpattern;

std::set<std::shared\_ptr<pattern::op::Label>> empty\_correlated\_matches;

RecurrentMatcher rm(padd, rpattern, empty\_correlated\_matches, nullptr);

ASSERT\_TRUE(rm.match(add3));

### Basic concepts[¶](#basic-concepts)

*Generic graph optimization passes*

This section discusses how to use nGraph to create a Pass Manager for your backend, and provides both a simple and a complex example to follow.

The pass manager infrastructure in nGraph makes it easy to reuse and mix the generic optimization passes. It also permits you to roll your own device-specific optimizations; that is, the same unified interface and APIs may be used to cover both things.

Invoking these passes is fairly straightforward, illustrated by the following steps and the code below.

1. Create a “pass manager” object (line 1)
2. Populate it with the desired pass or passes (lines 2-4)
3. Invoke the pass manager with a pointer to your unoptimized graph, and it will return a pointer to an optimized graph (lines 5-8)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | pass::Manager pass\_manager;  pass\_manager.register\_pass<pass::Liveness>();  pass\_manager.register\_pass<pass::MemoryLayout>();  Shape shape{1};  auto c = op::Constant::create(element::i32, shape, {5});  auto f = make\_shared<Function>(make\_shared<op::Negative>(c), ParameterVector{});  pass\_manager.run\_passes(f); |

nGraph Core includes a large library of hardware-agnostic passes useful for almost any kind of hardware backend. Some of these passes are likely familiar to people who are comfortable with classical compiler designs. Others, like the reshape/transpose elimination and sinking passes, are quite specific to deep learning.

### A simple example[¶](#a-simple-example)

Here’s a fairly straightforward function graph: it has 4 ops: [Convolution](index.odt/index.html" \l "document-ops/convolution), [Broadcast](index.odt/index.html" \l "document-ops/broadcast), [Add](index.odt/index.html" \l "document-ops/add), and [Relu](index.odt/index.html" \l "document-ops/relu). With nGraph, backends have the ability to rewrite the graph in ways that are specific to the underlying device/hardware’s capabilities.

When, for example, the device is an Intel® Architecture IA CPU, it can support a fused ConvolutionBiasReLU kernel. The backend is able to rewrite the graph into its own custom ops that more closely match the hardware-specific primitives; here they get matched via Intel® MKL-DNN.

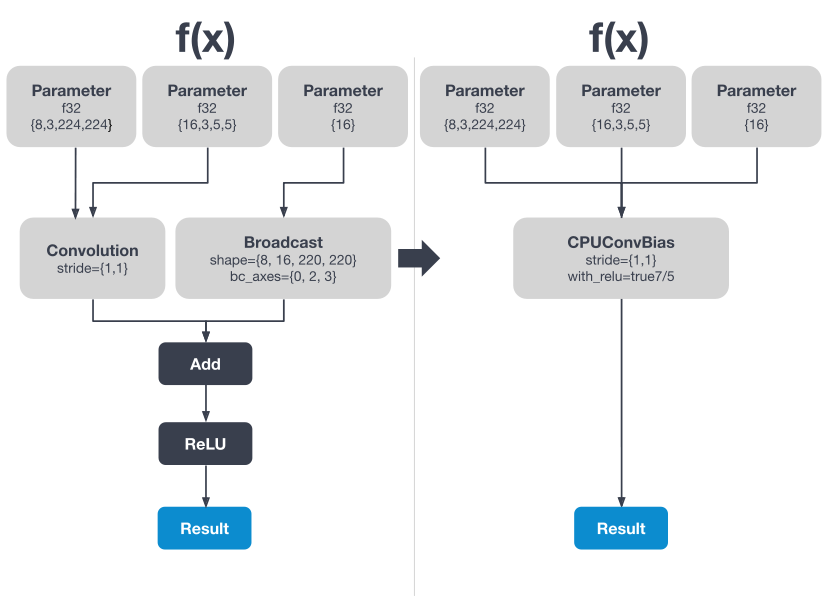
[](index.odt/_images/simple-compiler-passes.svg)

Figure A: On the left side of *Figure A* is a fully-formed function graph prior to fusion. After graph rewrite, the CPU implements a number of custom fusions.

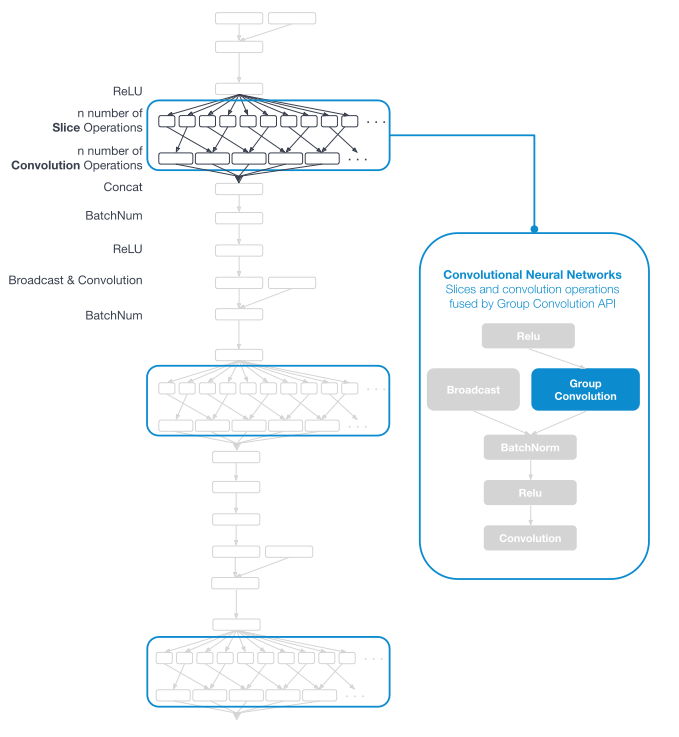
### A complex example[¶](#a-complex-example)

The effectiveness of graph-level optimization with nGraph is more striking to look at in terms of an actual input graph, such as one from the framework bridge. Here is slightly more complicated example drawn from a topology called MobileNet which makes heavy use of group convolution.

In group convolution, sometimes called depthwise convolution, a batch’s different feature channels get divided into groups that are processed independently, rather than every convolution kernel seeing all of the input feature channels.

With “Group Convolution Fusion”, it is possible to optimize a subgraph that has implemented group convolution by many instances of “ordinary” convolution.

*Figure B* shows an excerpt from MobileNet v1, a topology which makes heavy use of group convolution. Here, an image batch and a filter batch first undergo a “preprocessing” phase where segments along the channel axis are sliced out: one per channel group. Next, there are separate convolutions on each channel group before finally concatenating the result back together.

[](index.odt/_images/mobilenet-group-conv.svg)

**Figure B:** Each of these grouped convolution complexes – the operations within the rectangles on the left – is very wide; each is too wide to fit legibly on the illustration.

The group convolution fusion is able to replace each of those giant subgraphs with a single CPU group convolution node. This ends up being beneficial in several ways:

* Reduces sheer node count,
* Provides mappability to MKL-DNN, which has an accelerated group convolution implementation, and
* Eliminates unnecessary temporary nodes.

## Pattern Matcher[¶](#pattern-matcher)

### Overview: Optimize graphs with nGraph Compiler fusions[¶](#overview-optimize-graphs-with-ngraph-compiler-fusions)

The nGraph Compiler is an optimizing compiler. As such, it provides a way to capture a given [function graph](index.odt/index.html" \l "term-function-graph) and perform a series of optimization passes over that graph. The result is a semantically-equivalent graph that, when executed using any [backend](index.odt/index.html" \l "document-backends/index), has hardware-agnostic *and* hardware-specific optimizations, providing superior runtime characteristics to increase training performance or reduce inference latency.

There are several ways to describe what happens when we capture and translate the framework’s output of ops into an nGraph graph. [Fusion](index.odt/index.html" \l "term-fusion) is the term we shall use in our documentation; the action also can be described as: *combining*, *folding*, *squashing*, *collapsing*, or *merging* of graph functions.

Optimization passes may include algebraic simplifications, domain-specific simplifications, and fusion. Most passes share the same mode of operation (or the same operational structure) and consist of various stages (each one a [step](index.odt/index.html" \l "term-step)) where a developer can experiment with the intercepted or dynamic graph. These steps may be cycled or recycled as needed:

1. Locate a list of potentially-transformable subgraphs in the given graph.
2. Transform the selected candidates into semantically-equivalent subgraphs that execute faster, or with less memory (or both).
3. Verify that the optimization pass performs correctly, with any or all expected transformations, with the NGRAPH\_SERIALIZE\_TRACING option, which serializes a graph in the json format after a pass.
4. Measure and evaluate your performance improvements with NGRAPH\_CPU\_TRACING, which produces timelines compatible with chrome://tracing.

Optimizations can be experimented upon without using any backend by registering a pass with pass manager (Manager), calling run\_passes on a function, and then inspecting the transformed graph.

Optimization passes can be programmed ahead of time if you know or can predict what your graph will look like when it’s ready to be executed (in other words: which ops can be automatically translated into [nGraph Core ops](index.odt/index.html" \l "document-ops/index)).

The Interpreter is simply a backend providing reference implementations of ngraph ops in C++, with the focus on simplicity over performance.

### Example[¶](#example)

Let us first consider a simple example. A user would like to execute a graph that describes the following arithmetic expression:

\(a + b \* 1\) or \(Add(a, Mul(b, 1))\)

In the above expressions, 1 is an identity element; any element multiplied by the identity element is equal to itself. This is the same as saying:

\(b \* 1 = b\)

The writer of an optimization pass which uses algebraic simplification would probably want to first locate all multiplication expressions where multiplicands are multiplied by 1 (for stage 1) and to then transform, simplify, or replace those expressions with just their multiplicands (for stage 2).

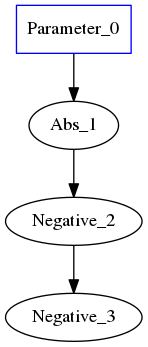
To make the work of an optimization pass writer easier, the nGraph Library includes facilities that enable the *finding* of relevant candidates using pattern matching (via pattern/matcher.hpp), and the *transforming* of the original graph into a condensed version (via pass/graph\_rewrite.hpp).

### Using GraphRewrite to fuse ops[¶](#using-graphrewrite-to-fuse-ops)

* [Exact pattern matching](#exact)
* [Constructing labels from existing nodes](#labels)

#### Exact pattern matching[¶](#exact-pattern-matching)

For the example of -(-A) = A, various graphs of varying complexity can be created and overwritten with recipes for pattern-matching + graph-rewrite. To get started, a simple example for a trivial graph, followed by more complex examples:

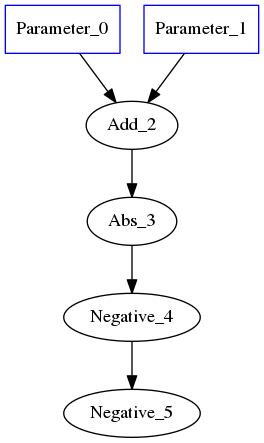


auto a = make\_shared<op::Parameter>(element::i32, shape);

auto absn = make\_shared<op::Abs>(a);

auto neg1 = make\_shared<op::Negative>(absn);

auto neg2 = make\_shared<op::Negative>(neg1);



auto a = make\_shared<op::Parameter>(element::i32, shape);

auto b = make\_shared<op::Parameter>(element::i32, shape);

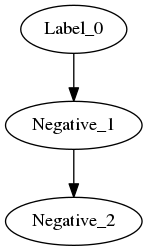
auto c = a + b;

auto absn = make\_shared<op::Abs>(c);

auto neg1 = make\_shared<op::Negative>(absn);

auto neg2 = make\_shared<op::Negative>(neg1);

##### Label AKA . in regexes[¶](#label-aka-in-regexes)



For the code below, element::f32 will still match integer Graph1 and Graph2

//note element::f32, will still match integer Graph1 and Graph2

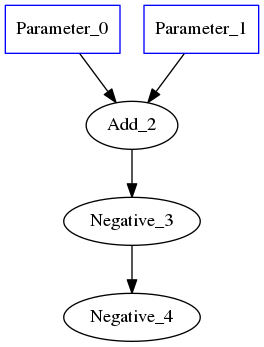
auto lbl = std::make\_shared<pattern::op::Label>(element::f32, Shape{});

auto neg1 = make\_shared<op::Negative>(lbl);

auto neg2 = make\_shared<op::Negative>(neg1);

#### Constructing labels from existing nodes[¶](#constructing-labels-from-existing-nodes)

##### Double Negative w/ Addition[¶](#double-negative-w-addition)



auto a = make\_shared<op::Parameter>(element::i32, shape);

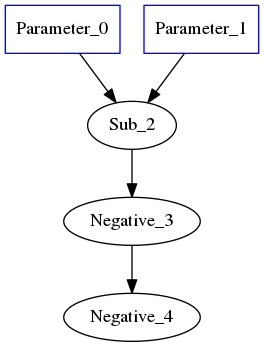
//`lbl` borrows the type and shape information from `a`

auto lbl = std::make\_shared<pattern::op::Label>(a);

auto neg1 = make\_shared<op::Negative>(a);

auto neg2 = make\_shared<op::Negative>(neg1);

##### Double Negative w/ Subtraction[¶](#double-negative-w-subtraction)



Predicates are of type std::function<bool(std::shared\_ptr<Node>)>

//predicates are of type std::function<bool(std::shared\_ptr<Node>)>

auto add\_or\_sub = [](std::shared\_ptr<Node> n) {

return std::dynamic\_pointer\_cast<op::Add>(n) != nullptr ||

std::dynamic\_pointer\_cast<op::Sub>(n) != nullptr

};

auto lbl = std::make\_shared<pattern::op::Label>(

element::f32,

Shape{},

add\_or\_sub

);

auto neg1 = make\_shared<op::Negative>(a);

auto neg2 = make\_shared<op::Negative>(neg1);

The nGraph Compiler is an optimizing compiler. As such, it provides a way to capture a given [function graph](index.odt/index.html" \l "term-function-graph) and perform a series of optimization passes over that graph. The result is a semantically-equivalent graph that, when executed using any [backend](index.odt/index.html" \l "document-backends/index), has optimizations inherent at the hardware level: superior runtime characteristics to increase training performance or reduce inference latency.

*class* Matcher[¶](#_CPPv4N6ngraph7pattern7MatcherE)

[Matcher](index.odt/index.html" \l "classngraph_1_1pattern_1_1Matcher) looks for node patterns in a computation graph. The patterns are described by an automaton that is described by an extended computation graph. The matcher executes by attempting to match the start node of the pattern to a computation graph value (output of a Node). In addition to determing if a match occurs, a pattern node may add graph nodes to a list of matched nodes, associate nodes with graph values, and start submatches. Submatches add match state changes to the enclosing match if the submatch succeeds; otherwise the state is reverted.

The default match behavior of a pattern node with a graph nodes is that the computation graph value is added to the end of the matched value list and the match succeeds if the node/pattern types match and the input values match. In the case of a commutative node, the inputs can match in any order. If the matcher is in strict mode, the graph value element type and shape must also match.

Pattern nodes that have different match behavior are in ngraph::pattern::op and have descriptions of their match behavior.

Public Functions

Matcher(*const* Output<Node> &*pattern\_node*, *const* std::string &*name*, bool *strict\_mode*)[¶](#_CPPv4N6ngraph7pattern7Matcher7MatcherERK6OutputI4NodeERKNSt6stringEb)

Constructs a [Matcher](index.odt/index.html" \l "classngraph_1_1pattern_1_1Matcher) object.

**Parameters**

* pattern\_node: is a pattern sub graph that will be matched against input graphs
* name: is a string which is used for logging and disabling a matcher
* strict\_mode: forces a matcher to consider shapes and ET of nodes

bool match(*const* Output<Node> &*graph\_value*)[¶](#_CPPv4N6ngraph7pattern7Matcher5matchERK6OutputI4NodeE)

Matches a pattern to graph\_node.

**Parameters**

* graph\_value: is an input graph to be matched against

bool match(*const* Output<Node> &*graph\_value*, *const* PatternMap &*previous\_matches*)[¶](#_CPPv4N6ngraph7pattern7Matcher5matchERK6OutputI4NodeERK10PatternMap)

Matches a pattern to graph\_node.

**Parameters**

* graph\_value: is an input graph to be matched against
* previous\_matches: contains previous mappings from labels to nodes to use

size\_t add\_node(Output<Node> *node*)[¶](#_CPPv4N6ngraph7pattern7Matcher8add_nodeE6OutputI4NodeE)

Low-level helper to match recurring patterns.

**Parameters**

* graph: is a graph to be matched against
* pattern: is a recurring pattern
* rpattern: specifies a node to recur from next
* patterns: a map from labels to matches

MatcherState start\_match()[¶](#_CPPv4N6ngraph7pattern7Matcher11start_matchEv)

Try a match.

### Fusion[¶](#fusion)

There are several ways to describe what happens when we capture and translate the framework’s output of ops into an nGraph graph. [Fusion](index.odt/index.html" \l "term-fusion) is the term we shall use in our documentation; the action also can be described as: *combining*, *folding*, *squashing*, *collapsing*, or *merging* of graph functions.

Optimization passes may include algebraic simplifications, domain-specific simplifications, and fusion. Most passes share the same mode of operation (or the same operational structure) and consist of various stages (each one a [step](index.odt/index.html" \l "term-step)) where a developer can experiment with the intercepted or dynamic graph. These steps may be cycled or recycled as needed:

1. Locate a list of potentially-transformable subgraphs in the given graph.
2. Transform the selected candidates into semantically-equivalent subgraphs that execute faster, or with less memory (or both).
3. Verify that the optimization pass performs correctly, with any or all expected transformations, with the NGRAPH\_SERIALIZE\_TRACING option, which serializes a graph in the json format after a pass.
4. Measure and evaluate your performance improvements with NGRAPH\_CPU\_TRACING, which produces timelines compatible with chrome://tracing.

Optimizations can be experimented upon without using any backend by registering a pass with pass manager (Manager), calling run\_passes on a function, and then inspecting the transformed graph.

Optimization passes can be programmed ahead of time if you know or can predict what your graph will look like when it’s ready to be executed (in other words: which ops can be automatically translated into [nGraph Core ops](index.odt/index.html" \l "document-ops/index)).

The Interpreter is simply a backend providing reference implementations of ngraph ops in C++, with the focus on simplicity over performance.

#### Example[¶](#example)

Let us first consider a simple example. A user would like to execute a graph that describes the following arithmetic expression:

\(a + b \* 1\) or \(Add(a, Mul(b, 1))\)

In the above expressions, 1 is an identity element; any element multiplied by the identity element is equal to itself. In other words, the original expression \(a + b \* 1\) is exactly equivalent to the expression \(a + b\), so we can eliminate this extra multiplication step.

The writer of an optimization pass which uses algebraic simplification would probably want to first locate all multiplication expressions where multiplicands are multiplied by 1 (for stage 1) and to then replace, those expressions with just their multiplicands (for stage 2).

To make the work of an optimization pass writer easier, the nGraph Library includes facilities that enable the *finding* of relevant candidates using pattern matching (via pattern/matcher.hpp), and the *transforming* of the original graph into an optimized version (via pass/graph\_rewrite.hpp).

## List of Core ops[¶](#list-of-core-ops)

Some operations are experimental.

[More about Core Ops](#more-about)

|  |  |  |
| --- | --- | --- |
| * [Abs](index.odt/index.html" \l "document-ops/abs) * [Acos](index.odt/index.html" \l "document-ops/acos) * [Add](index.odt/index.html" \l "document-ops/add) * [All](index.odt/index.html" \l "document-ops/all) * [AllReduce](index.odt/index.html" \l "document-ops/allreduce) * [And](index.odt/index.html" \l "document-ops/and) * [Any](index.odt/index.html" \l "document-ops/any) * [Asin](index.odt/index.html" \l "document-ops/asin) * [Atan](index.odt/index.html" \l "document-ops/atan) * [AvgPool](index.odt/index.html" \l "document-ops/avg_pool) * [AvgPoolBackprop](index.odt/index.html" \l "document-ops/avg_pool_backprop) * [BatchNormInference](index.odt/index.html" \l "document-ops/batch_norm_inference) * [BatchNormTraining](index.odt/index.html" \l "document-ops/batch_norm_training) * [BatchNormTrainingBackprop](index.odt/index.html" \l "document-ops/batch_norm_training_backprop) * [Broadcast](index.odt/index.html" \l "document-ops/broadcast) * [BroadcastDistributed](index.odt/index.html" \l "document-ops/broadcast_distributed) * [Ceiling](index.odt/index.html" \l "document-ops/ceiling) * [Concat](index.odt/index.html" \l "document-ops/concat) * [Constant](index.odt/index.html" \l "document-ops/constant) * [Convert](index.odt/index.html" \l "document-ops/convert) * [Convolution](index.odt/index.html" \l "document-ops/convolution) * [Cos](index.odt/index.html" \l "document-ops/cos) * [Cosh](index.odt/index.html" \l "document-ops/cosh) | * [Dequantize](index.odt/index.html" \l "document-ops/dequantize) * [Divide](index.odt/index.html" \l "document-ops/divide) * [Dot](index.odt/index.html" \l "document-ops/dot) * [DropOut](index.odt/index.html" \l "document-ops/drop_out) * [Equal](index.odt/index.html" \l "document-ops/equal) * [Exp](index.odt/index.html" \l "document-ops/exp) * [Floor](index.odt/index.html" \l "document-ops/floor) * [GetOutputElement](index.odt/index.html" \l "document-ops/get_output_element) * [GreaterEq](index.odt/index.html" \l "document-ops/greater_eq) * [Greater](index.odt/index.html" \l "document-ops/greater) * [LessEq](index.odt/index.html" \l "document-ops/less_eq) * [Less](index.odt/index.html" \l "document-ops/less) * [Log](index.odt/index.html" \l "document-ops/log) * [Max](index.odt/index.html" \l "document-ops/max) * [Maximum](index.odt/index.html" \l "document-ops/maximum) * [MaxPool](index.odt/index.html" \l "document-ops/max_pool) * [Min](index.odt/index.html" \l "document-ops/min) * [Minimum](index.odt/index.html" \l "document-ops/minimum) * [Multiply](index.odt/index.html" \l "document-ops/multiply) * [Negative](index.odt/index.html" \l "document-ops/negative) * [NotEqual](index.odt/index.html" \l "document-ops/not_equal) * [Not](index.odt/index.html" \l "document-ops/not) * [OneHot](index.odt/index.html" \l "document-ops/one_hot) | * [Or](index.odt/index.html" \l "document-ops/or) * [Pad](index.odt/index.html" \l "document-ops/pad) * [Parameter](index.odt/index.html" \l "document-ops/parameter) * [Power](index.odt/index.html" \l "document-ops/power) * [Product](index.odt/index.html" \l "document-ops/product) * [Quantize](index.odt/index.html" \l "document-ops/quantize) * [RandomUniform](index.odt/index.html" \l "document-ops/random_uniform) * [Relu](index.odt/index.html" \l "document-ops/relu) * [Result](index.odt/index.html" \l "document-ops/result) * [ShapeOf](index.odt/index.html" \l "document-ops/shape_of) * [Sigmoid](index.odt/index.html" \l "document-ops/sigmoid) * [Sign](index.odt/index.html" \l "document-ops/sign) * [Sin](index.odt/index.html" \l "document-ops/sin) * [Sinh](index.odt/index.html" \l "document-ops/sinh) * [Slice](index.odt/index.html" \l "document-ops/slice) * [Softmax](index.odt/index.html" \l "document-ops/softmax) * [Sqrt](index.odt/index.html" \l "document-ops/sqrt) * [Subtract](index.odt/index.html" \l "document-ops/subtract) * [Tan](index.odt/index.html" \l "document-ops/tan) * [Tanh](index.odt/index.html" \l "document-ops/tanh) * [Transpose](index.odt/index.html" \l "document-ops/transpose) * [Xor](index.odt/index.html" \l "document-ops/xor) |

### Abs[¶](#abs)

Abs // Elementwise absolute value operation

#### Description[¶](#description)

Produces a single output tensor of the same element type and shape as arg, where the value at each coordinate of output is the absolute value of the value at each arg coordinate.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \left|\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}}\right|\]

#### Backprop[¶](#backprop)

\[\overline{\mathtt{arg}} \leftarrow \Delta\ \mathrm{sgn}(\mathtt{arg})\]

#### C++ Interface[¶](#c-interface)

*class* Abs : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v03AbsE)

Elementwise absolute value operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v03Abs13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Abs()[¶](#_CPPv4N6ngraph2op2v03Abs3AbsEv)

Constructs an absolute value operation.

Abs(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v03Abs3AbsERK6OutputI4NodeE)

Constructs an absolute value operation.

Output [d1, ...]

**Parameters**

* arg: Output that produces the input tensor.[d1, ...]

### Acos[¶](#acos)

Acos // Elementwise acos operation

#### Description[¶](#description)

Produces a tensor of the same element type and shape as arg, where the value at each coordinate of output is the inverse cosine of the value at the corresponding coordinate of arg.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \cos^{-1}(\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}})\]

#### Backprop[¶](#backprop)

\[\overline{\mathtt{arg}} \leftarrow -\frac{\Delta}{\sqrt{1-\mathtt{arg}^2}}\]

#### C++ Interface[¶](#c-interface)

*class* Acos : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v04AcosE)

Elementwise inverse cosine (arccos) operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v04Acos13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Acos()[¶](#_CPPv4N6ngraph2op2v04Acos4AcosEv)

Constructs an arccos operation.

Acos(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v04Acos4AcosERK6OutputI4NodeE)

Constructs an arccos operation.

Output [d1, ...]

**Parameters**

* arg: Output that produces the input tensor.[d1, ...]

### Add[¶](#add)

Add // Elementwise add operation

#### Description[¶](#description)

Elementwise add operation.

Produces tensor of the same element type and shape as the two inputs, where the value at each coordinate of output is the sum of the values at the corresponding input coordinates.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg0 | any | any |
| arg1 | same as arg0 | same as arg0 |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | same as arg0 | same as arg0 |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \mathtt{arg0}\_{i\_0, \ldots, i\_{n-1}} + \mathtt{arg1}\_{i\_0, \ldots, i\_{n-1}}\]

#### Backprop[¶](#backprop)

\[\begin{split}\overline{\mathtt{arg0}} &\leftarrow \Delta \\ \overline{\mathtt{arg1}} &\leftarrow \Delta\end{split}\]

#### C++ Interface[¶](#c-interface)

*class* Add : *public* ngraph::op::util::BinaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v03AddE)

Elementwise addition operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v03Add13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Add()[¶](#_CPPv4N6ngraph2op2v03Add3AddEv)

Constructs an uninitialized addition operation.

Add(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, *const* AutoBroadcastSpec &*auto\_broadcast* = AutoBroadcastSpec())[¶](#_CPPv4N2op2v03Add3AddERK6OutputI4NodeERK6OutputI4NodeERK17AutoBroadcastSpec)

Constructs an addition operation.

Output [d0, ...]

**Parameters**

* arg0: Output that produces the first input tensor.[d0, ...]
* arg1: Output that produces the second input tensor.[d0, ...]
* auto\_broadcast: Auto broadcast specification

### All[¶](#all)

All // Boolean "all" reduction operation.

#### Description[¶](#description)

Reduces a tensor of booleans, eliminating the specified reduction axes by taking the logical conjunction (i.e., “AND-reduce”).

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | ngraph::element::boolean | Any |

##### Attributes[¶](#attributes)

| Name | Description |
| --- | --- |
| reduction\_axes | The axis positions (0-based) on which to calculate the conjunction |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg, with reduction\_axes removed. |

#### C++ Interface[¶](#c-interface)

*class* All : *public* ngraph::op::util::LogicalReduction[¶](#_CPPv4N6ngraph2op2v03AllE)

Logical “all” reduction operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v03All13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

All()[¶](#_CPPv4N6ngraph2op2v03All3AllEv)

Constructs an “all” reduction operation.

All(*const* Output<Node> &*arg*, *const* AxisSet &*reduction\_axes*)[¶](#_CPPv4N6ngraph2op2v03All3AllERK6OutputI4NodeERK7AxisSet)

Constructs an “all” reduction operation.

**Parameters**

* arg: The tensor to be reduced.
* reduction\_axes: The axis positions (0-based) to be eliminated.

All(*const* Output<Node> &*arg*, *const* Output<Node> &*reduction\_axes*)[¶](#_CPPv4N6ngraph2op2v03All3AllERK6OutputI4NodeERK6OutputI4NodeE)

Constructs an “all” reduction operation.

**Parameters**

* arg: The tensor to be reduced.
* reduction\_axes: The axis positions (0-based) to be eliminated.

*virtual* std::shared\_ptr<Node> get\_default\_value() *const*[¶](#_CPPv4NK6ngraph2op2v03All17get_default_valueEv)

**Return**

The default value for [All](index.odt/index.html" \l "classngraph_1_1op_1_1v0_1_1All).

### AllReduce[¶](#allreduce)

AllReduce // Collective operation

#### Description[¶](#description)

Combines values from all processes or devices and distributes the result back to all processes or devices.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | element::f32 element::f64 | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | element::f32 element::f64 | Same as arg |

#### C++ Interface[¶](#c-interface)

*class* AllReduce : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v09AllReduceE)

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v09AllReduce13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

void validate\_and\_infer\_types()[¶](#_CPPv4N6ngraph2op2v09AllReduce24validate_and_infer_typesEv)

Throws if the node is invalid.

### And[¶](#and)

And // Elementwise logical-and operation

#### Description[¶](#description)

Produces tensor with boolean element type and shape as the two inputs, which must themselves have boolean element type, where the value at each coordinate of output is 1 (true) if arg0 and arg1 are both nonzero, 0 otherwise.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg0 | ngraph::element::boolean | any |
| arg1 | ngraph::element::boolean | same as arg0 |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | ngraph::element::boolean | same as arg0 |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \mathtt{arg0}\_{i\_0, \ldots, i\_{n-1}}\, \mathtt{&&}\, \mathtt{arg1}\_{i\_0, \ldots, i\_{n-1}}\]

#### C++ Interface[¶](#c-interface)

*class* And : *public* ngraph::op::util::BinaryElementwiseLogical[¶](#_CPPv4N6ngraph2op2v03AndE)

Elementwise logical-and operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v03And13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

And()[¶](#_CPPv4N6ngraph2op2v03And3AndEv)

Constructs a logical-and operation.

And(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, *const* AutoBroadcastSpec &*auto\_broadcast* = AutoBroadcastSpec())[¶](#_CPPv4N2op2v03And3AndERK6OutputI4NodeERK6OutputI4NodeERK17AutoBroadcastSpec)

Constructs a logical-and operation.

Output [d0, ...]

**Parameters**

* arg0: Output that produces the first input tensor.[d0, ...]
* arg1: Output that produces the second input tensor.[d0, ...]
* auto\_broadcast: Auto broadcast specification

### Any[¶](#any)

Any // Boolean "any" reduction operation.

#### Description[¶](#description)

Reduces a tensor of booleans, eliminating the specified reduction axes by taking the logical disjunction (i.e., “OR-reduce”).

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | ngraph::element::boolean | Any |

##### Attributes[¶](#attributes)

| Name | Description |
| --- | --- |
| reduction\_axes | The axis positions (0-based) on which to calculate the disjunction |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg, with reduction\_axes removed. |

#### C++ Interface[¶](#c-interface)

*class* Any : *public* ngraph::op::util::LogicalReduction[¶](#_CPPv4N6ngraph2op3AnyE)

Logical “any” reduction operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op3Any13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Any()[¶](#_CPPv4N6ngraph2op3Any3AnyEv)

Constructs an “any” reduction operation.

Any(*const* Output<Node> &*arg*, *const* AxisSet &*reduction\_axes*)[¶](#_CPPv4N2op3Any3AnyERK6OutputI4NodeERK7AxisSet)

Constructs an “any” reduction operation.

**Parameters**

* arg: The tensor to be reduced.
* reduction\_axes: The axis positions (0-based) to be eliminated.

Any(*const* Output<Node> &*arg*, *const* Output<Node> &*reduction\_axes*)[¶](#_CPPv4N2op3Any3AnyERK6OutputI4NodeERK6OutputI4NodeE)

Constructs an “any” reduction operation.

**Parameters**

* arg: The tensor to be reduced.
* reduction\_axes: The axis positions (0-based) to be eliminated.

shared\_ptr<Node> get\_default\_value() *const*[¶](#_CPPv4NK2op3Any17get_default_valueEv)

**Return**

The default value for [Any](index.odt/index.html" \l "classngraph_1_1op_1_1Any).

### Asin[¶](#asin)

Asin // Elementwise asin operation

#### Description[¶](#description)

Produces a tensor of the same element type and shape as arg, where the value at each coordinate of output is the inverse sine of the value at the corresponding coordinate of arg.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg. |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \sin^{-1}(\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}})\]

#### Backprop[¶](#backprop)

\[\overline{\mathtt{arg}} \leftarrow \frac{\Delta}{\sqrt{1-\mathtt{arg}^2}}\]

#### C++ Interface[¶](#c-interface)

*class* Asin : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v04AsinE)

Elementwise inverse sine (arcsin) operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v04Asin13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Asin()[¶](#_CPPv4N6ngraph2op2v04Asin4AsinEv)

Constructs an arcsin operation.

Asin(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v04Asin4AsinERK6OutputI4NodeE)

Constructs an arcsin operation.

Output [d1, ...]

**Parameters**

* arg: Output that produces the input tensor.[d1, ...]

### Atan[¶](#atan)

Atan // Elementwise atan operation

#### Description[¶](#description)

Produces a tensor of the same element type and shape as arg, where the value at each coordinate of output is the inverse tangent of the value at the corresponding coordinate of arg.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \tan^{-1}(\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}})\]

#### Backprop[¶](#backprop)

\[\overline{\mathtt{arg}} \leftarrow \frac{\Delta}{1+\mathtt{arg}^2}\]

#### C++ Interface[¶](#c-interface)

*class* Atan : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v04AtanE)

Elementwise inverse tangent (arctan) operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v04Atan13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Atan()[¶](#_CPPv4N6ngraph2op2v04Atan4AtanEv)

Constructs an arctan operation.

Atan(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v04Atan4AtanERK6OutputI4NodeE)

Constructs an arctan operation.

Output [d1, ...]

**Parameters**

* arg: Output that produces the input tensor.[d1, ...]

### AvgPool[¶](#avgpool)

AvgPool // Average Pooling operation

#### Description[¶](#description)

Average pooling windows its input and produces an average for each window.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape | Notes |
| --- | --- | --- | --- |
| data | Any | \((N,C,d\_1,\ldots,d\_n)\) | \(n>0, d\_i>0\) |

##### Attributes[¶](#attributes)

| Name | Type | Notes |
| --- | --- | --- |
| w | Shape[n] | Window shape. \(w\_i\le d\_i\) |
| s | Strides[n] | Window strides. |
| p | Shape[n] | Padding below. |
| q | Shape[n] | Padding above. |
| i | Boolean | Include padding in average. |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Any | \((N,C,d'\_1,\ldots,d'\_n)\) |

Average pooling takes as its input, a batch tensor data of shape \((N,C,d\_1,\ldots,d\_n)\), where where \(N\) is the batch size, and \(C > 0\) is the number of channels (sometimes called features). The dimensions \((d\_1,\ldots,d\_n)\) correspond to the shape of an \(n\)-dimensional data item in a batch. For example, where \(n=2\), the data may represent a two-dimensional image. It also takes four attributes:

1. *window shape*,
2. *window movement strides*, (optional)
3. *padding below*, (optional)
4. *padding above*, (optional)
5. *include padding in average*

The shape of output is \((N,C,d'\_1,\ldots,d'\_n)\), where \(d'\_n = \lceil \frac{p\_i + d\_i + q\_i - w\_i + 1}{s\_i} \rceil\).

**Informal definition:** If \(\textit{i}\) is \(\textit{true}\), then averages are computed as though the padding region contained regular elements of value zero. If \(\textit{i}\) is \(\textit{false}\), then averages are computed using only the non-padding tensor elements that are present in each window.

*Example:* Consider two instances of this operator with the following attributes: \(\textit{w} = (2,2)\), \(\textit{s} = (1,1)\), \(\textit{p} = (1,1)\), and (in one instance) \(\textit{i} = false\) or (in the other instance) \(\textit{i} = true\).

Consider how those two operator instances would handle this input tensor:

\[\begin{split}T\_\textit{in} = \begin{bmatrix} 1 & 3 & 5 & \ldots \\ 7 & 11 & 13 & \ldots \\ 17 & 19 & 23 & \ldots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}\end{split}\]

Applying the padding indicated by the value of \(\textit{p}\), we have the padded image of \(T\_\textit{in}\) as follows:

\[\begin{split}T\_\textit{in,padded} = \begin{bmatrix} (0) & (0) & (0) & (0) & \ldots \\ (0) & 1 & 3 & 5 & \ldots \\ (0) & 7 & 11 & 13 & \ldots \\ (0) & 17 & 19 & 23 & \ldots \\ (0) & \vdots & \vdots & \vdots & \ddots \end{bmatrix}\end{split}\]

Now consider how the two variations of this example’s *AvgPool* operator will compute the “average” value of the top-left window, which contains exactly the elements:

\[\begin{split}\begin{bmatrix} (0) & (0) \\ (0) & 1 \end{bmatrix}\end{split}\]

If \(\textit{i} = false\), then the operator simply ignores the padding elements. It therefore computes the average of the single-element set \(\{ 1 \}\), yielding \(1.0\).

If \(\textit{i} = true\), then the operator computes the average of the set \(\{ 0, 0, 0, 1\}\), yielding 0.25.

*Note:* This operator is ill-defined when *both* of the following conditions hold: (1) \(\textit{i} = false\), and (2) the operator’s other attribute values indicate that at least one window will contain only padding elements.

**Formal definition:** *In the absence of padding*, given an input data batch tensor \(T\_\textit{in}\), the output tensor is defined by the equation

\[T\_\textit{out}[a,c,i\_1,\ldots,i\_n] = \frac{\sum\_{j\_1 = s\_1 i\_1, \ldots, j\_n = s\_n i\_n}^{j\_1 = s\_1 i\_1 + w\_1 - 1, \ldots, j\_n = s\_n i\_n + w\_n - 1} T\_\textit{in}[a,c,j\_1,\ldots,j\_n]}{\prod\_{i=1}^n{w\_n}}\]

*In the presence of padding*, we do not always want to divide by a reciprocal equal to the number of elements in the window, since some of the output points are determined by a window that is partly hanging beyond the edge of the tensor. In this case we can define the output

In this case we can define the output via a few intermediate steps.

First define the *sum tensor* \(T\_\textit{sum}\), with shape \((N,C,d'\_1,\ldots,d'\_n)\), as follows.

\[T\_\textit{sum}[a,c,i\_1,\ldots,i\_n] = \frac{\sum\_{j\_1 = s\_1 i\_1, \ldots, j\_n = s\_n i\_n}^{j\_1 = s\_1 i\_1 + w\_1 - 1, \ldots, j\_n = s\_n i\_n + w\_n - 1} \textit{val}[a,c,j\_1,\ldots,j\_n]}{\prod\_{i=1}^n{w\_n}}\]

where

\[\begin{split}\textit{val}[a,c,j\_1,\ldots,j\_n] = \begin{cases} T\_\textit{in}[a,c,j\_1,\ldots,j\_n]&\text{if for all } k, p\_k \le j\_k < p\_k + d\_k\\ 0&\text{otherwise}. \end{cases}\end{split}\]

Second, define the *divisor tensor* \(T\_\textit{div}\), with shape \((N,C,d'\_1,\ldots,d'\_n)\), as follows.

\[T\_\textit{div}[a,c,i\_1,\ldots,i\_n] = \frac{\sum\_{j\_1 = s\_1 i\_1, \ldots, j\_n = s\_n i\_n}^{j\_1 = s\_1 i\_1 + w\_1 - 1, \ldots, j\_n = s\_n i\_n + w\_n - 1} \textit{val}[a,c,j\_1,\ldots,j\_n]}{\prod\_{i=1}^n{w\_n}}\]

where

\[\begin{split}\textit{val}[a,c,j\_1,\ldots,j\_n] = \begin{cases} 1&\text{if for all }k, p\_k \le j\_k < p\_k + d\_k\\ 0&\text{otherwise}. \end{cases}\end{split}\]

Finally, define \(T\_\textit{out}\) as the result of elementwise dividing \(T\_\textit{sum}\) by \(T\_\textit{div}\). Note that at positions where \(T\_\textit{div}\) is zero, values may be infinity or nan. (This corresponds to a condition where the pooling window is completely out of bounds, encompassing no valid values.)

#### Backprop[¶](#backprop)

#### C++ Interface[¶](#c-interface)

*class* AvgPool : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v07AvgPoolE)

Batched average pooling operation, with optional padding and window stride.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v07AvgPool13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

AvgPool()[¶](#_CPPv4N6ngraph2op2v07AvgPool7AvgPoolEv)

Constructs a batched average pooling operation.

AvgPool(*const* Output<Node> &*arg*, *const* Shape &*window\_shape*, *const* Strides &*window\_movement\_strides*, *const* Shape &*padding\_below*, *const* Shape &*padding\_above*, bool *include\_padding\_in\_avg\_computation*, *const* PadType &*pad\_type*, bool *ceil\_mode*)[¶](#_CPPv4N2op2v07AvgPool7AvgPoolERK6OutputI4NodeERK5ShapeRK7StridesRK5ShapeRK5ShapebRK7PadTypeb)

Constructs a batched average pooling operation.

**Parameters**

* arg: The output producing the input data batch tensor.[d1, dn]
* window\_shape: The window shape.[n]
* window\_movement\_strides: The window movement strides.[n]
* padding\_below: The below-padding shape.[n]
* padding\_above: The above-padding shape.[n]
* include\_padding\_in\_avg\_computation: If true then averages include padding elements, each treated as the number zero. If false, padding elements are entirely ignored when computing averages.
* pad\_type: Padding type to use for additional padded dimensions
* ceil\_mode: Whether to use ceiling while computing output shape.

AvgPool(*const* Output<Node> &*arg*, *const* Shape &*window\_shape*, *const* Strides &*window\_movement\_strides*, *const* Shape &*padding\_below*, *const* Shape &*padding\_above*, bool *include\_padding\_in\_avg\_computation*, *const* PadType &*pad\_type*)[¶](#_CPPv4N2op2v07AvgPool7AvgPoolERK6OutputI4NodeERK5ShapeRK7StridesRK5ShapeRK5ShapebRK7PadType)

Constructs a batched average pooling operation.

**Parameters**

* arg: The output producing the input data batch tensor.[d1, dn]
* window\_shape: The window shape.[n]
* window\_movement\_strides: The window movement strides.[n]
* padding\_below: The below-padding shape.[n]
* padding\_above: The above-padding shape.[n]
* include\_padding\_in\_avg\_computation: If true then averages include padding elements, each treated as the number zero. If false, padding elements are entirely ignored when computing averages.
* pad\_type: Padding type to use for additional padded dimensions

AvgPool(*const* Output<Node> &*arg*, *const* Shape &*window\_shape*, *const* Strides &*window\_movement\_strides*, *const* Shape &*padding\_below*, *const* Shape &*padding\_above*, bool *include\_padding\_in\_avg\_computation* = false)[¶](#_CPPv4N2op2v07AvgPool7AvgPoolERK6OutputI4NodeERK5ShapeRK7StridesRK5ShapeRK5Shapeb)

Constructs a batched average pooling operation.

**Parameters**

* arg: The output producing the input data batch tensor.[d1, dn]
* window\_shape: The window shape.[n]
* window\_movement\_strides: The window movement strides.[n]
* padding\_below: The below-padding shape.[n]
* padding\_above: The above-padding shape.[n]
* include\_padding\_in\_avg\_computation: If true then averages include padding elements, each treated as the number zero. If false, padding elements are entirely ignored when computing averages.

AvgPool(*const* Output<Node> &*arg*, *const* Shape &*window\_shape*, *const* Strides &*window\_movement\_strides*)[¶](#_CPPv4N2op2v07AvgPool7AvgPoolERK6OutputI4NodeERK5ShapeRK7Strides)

Constructs a batched, unpadded average pooling operation (i.e., all padding shapes are set to 0).

**Parameters**

* arg: The output producing the input data batch tensor.[d1, ..., dn]
* window\_shape: The window shape.[n]
* window\_movement\_strides: The window movement strides.[n]

AvgPool(*const* Output<Node> &*arg*, *const* Shape &*window\_shape*)[¶](#_CPPv4N2op2v07AvgPool7AvgPoolERK6OutputI4NodeERK5Shape)

Constructs an unstrided batched convolution operation (i.e., all window movement strides are 1 and all padding shapes are set to 0).

**Parameters**

* arg: The output producing the input data batch tensor.[d1, ..., dn]
* window\_shape: The window shape.[n]

void validate\_and\_infer\_types()[¶](#_CPPv4N2op2v07AvgPool24validate_and_infer_typesEv)

Throws if the node is invalid.

*const* Shape &get\_window\_shape() *const*[¶](#_CPPv4NK2op2v07AvgPool16get_window_shapeEv)

**Return**

The window shape.

*const* Strides &get\_window\_movement\_strides() *const*[¶](#_CPPv4NK2op2v07AvgPool27get_window_movement_stridesEv)

**Return**

The window movement strides.

*const* Shape &get\_padding\_below() *const*[¶](#_CPPv4NK2op2v07AvgPool17get_padding_belowEv)

**Return**

The below-padding shape.

*const* Shape &get\_padding\_above() *const*[¶](#_CPPv4NK2op2v07AvgPool17get_padding_aboveEv)

**Return**

The above-padding shape.

*const* op::PadType &get\_pad\_type() *const*[¶](#_CPPv4NK2op2v07AvgPool12get_pad_typeEv)

**Return**

The pad type for pooling.

shared\_ptr<Node> get\_default\_value() *const*[¶](#_CPPv4NK2op2v07AvgPool17get_default_valueEv)

**Return**

The default value for [AvgPool](index.odt/index.html" \l "classngraph_1_1op_1_1v0_1_1AvgPool).

### AvgPoolBackprop[¶](#avgpoolbackprop)

AvgPoolBackprop // Average Pooling backprop operation.

#### Description[¶](#description)

#### C++ Interface[¶](#c-interface)

*class* AvgPoolBackprop : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v015AvgPoolBackpropE)

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v015AvgPoolBackprop13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

void validate\_and\_infer\_types()[¶](#_CPPv4N2op2v015AvgPoolBackprop24validate_and_infer_typesEv)

Throws if the node is invalid.

#### Python Interface[¶](#python-interface)

### BatchNormInference[¶](#batchnorminference)

BatchNormInference // Adjust input for mean and variance

#### Description[¶](#description)

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| input | real | \((\bullet, C, \ldots)\) |
| gamma | same as input | \((C)\) |
| beta | same as input | \((C)\) |
| mean | same as input | \((C)\) |
| variances | same as input | \((C)\) |

##### Attributes[¶](#attributes)

| Name | Type | Notes |
| --- | --- | --- |
| epsilon | double | Small bias added to variance to avoid division by 0. |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| normalized | same as gamma | Same as input |

#### Mathematical Definition[¶](#mathematical-definition)

The axes of the input fall into two categories: positional and channel, with channel being axis 1. For each position, there are \(C\) channel values, each normalized independently.

Normalization of a channel sample is controlled by two values:

* the mean \(\mu\), and
* the variance \(\sigma^2\);

and by two scaling attributes: \(\gamma\) and \(\beta\).

\[\mathtt{normalized}\_{\bullet, c, \ldots} = \frac{\mathtt{input}\_{\bullet, c, \ldots}-\mu\_c}{\sqrt{\sigma^2\_c+\epsilon}}\gamma\_c+\beta\_c\]

#### C++ Interface[¶](#c-interface)

*class* BatchNormInference : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v018BatchNormInferenceE)

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v018BatchNormInference13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

BatchNormInference(*const* Output<Node> &*input*, *const* Output<Node> &*gamma*, *const* Output<Node> &*beta*, *const* Output<Node> &*mean*, *const* Output<Node> &*variance*, double *epsilon*)[¶](#_CPPv4N6ngraph2op2v018BatchNormInference18BatchNormInferenceERK6OutputI4NodeERK6OutputI4NodeERK6OutputI4NodeERK6OutputI4NodeERK6OutputI4NodeEd)

**Parameters**

* input: [., C, …]
* gamma: gamma scaling for normalized value. [C]
* beta: bias added to the scaled normalized value [C]
* mean: value for mean normalization [C]
* variance: value for variance normalization [C]
* epsilon: Avoids divsion by 0 if input has 0 variance

NGRAPH\_DEPRECATED\_DOC BatchNormInference(double *eps*, *const* Output<Node> &*gamma*, *const* Output<Node> &*beta*, *const* Output<Node> &*input*, *const* Output<Node> &*mean*, *const* Output<Node> &*variance*)[¶](#_CPPv4N6ngraph2op2v018BatchNormInference18BatchNormInferenceEdRK6OutputI4NodeERK6OutputI4NodeERK6OutputI4NodeERK6OutputI4NodeERK6OutputI4NodeE)

In this version of BatchNorm:

MEAN AND VARIANCE: provided by the ‘mean’ and ‘variance’ parameters.

OUTPUT VALUE: a single tensor with the normalized value of ‘input’.

AUTODIFF SUPPORT:

* ’generate\_adjoints(…) may throw an exception.

SHAPE DETAILS: gamma: must have rank 1, with the same span as input’s channel axis. beta: must have rank 1, with the same span as input’s channel axis. input: must have rank >= 2. The second dimension represents the channel axis and must have a span of at least 1. mean: must have rank 1, with the same span as input’s channel axis. variance: must have rank 1, with the same span as input’s channel axis. output: shall have the same shape as ‘input’.

void validate\_and\_infer\_types()[¶](#_CPPv4N6ngraph2op2v018BatchNormInference24validate_and_infer_typesEv)

Throws if the node is invalid.

### BatchNormTraining[¶](#batchnormtraining)

BatchNormTraining // Compute mean and variance from the input.

#### Description[¶](#description)

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| input | real | \((\bullet, C, \ldots)\) |
| gamma | same as input | \((C)\) |
| beta | same as input | \((C)\) |

##### Attributes[¶](#attributes)

| Name | Type | Notes |
| --- | --- | --- |
| epsilon | double | Small bias added to variance to avoid division by 0. |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| normalized | same as gamma | Same as input |
| batch\_mean | same as gamma | \((C)\) |
| batch\_variance | same as gamma | \((C)\) |

The batch\_mean and batch\_variance outputs are computed per-channel from input.

#### Mathematical Definition[¶](#mathematical-definition)

The axes of the input fall into two categories: positional and channel, with channel being axis 1. For each position, there are \(C\) channel values, each normalized independently.

Normalization of a channel sample is controlled by two values:

* the batch\_mean \(\mu\), and
* the batch\_variance \(\sigma^2\);

and by two scaling attributes: \(\gamma\) and \(\beta\).

The values for \(\mu\) and \(\sigma^2\) come from computing the mean and variance of input.

\[\begin{split}\mu\_c &= \mathop{\mathbb{E}}\left(\mathtt{input}\_{\bullet, c, \ldots}\right)\\ \sigma^2\_c &= \mathop{\mathtt{Var}}\left(\mathtt{input}\_{\bullet, c, \ldots}\right)\\ \mathtt{normlized}\_{\bullet, c, \ldots} &= \frac{\mathtt{input}\_{\bullet, c, \ldots}-\mu\_c}{\sqrt{\sigma^2\_c+\epsilon}}\gamma\_c+\beta\_c\end{split}\]

#### Backprop[¶](#backprop)

\[\begin{split}[\overline{\texttt{input}}, \overline{\texttt{gamma}}, \overline{\texttt{beta}}]=\\ \mathop{\texttt{BatchNormTrainingBackprop}}(\texttt{input},\texttt{gamma},\texttt{beta},\texttt{mean},\texttt{variance},\overline{\texttt{normed\_input}}).\end{split}\]

#### C++ Interface[¶](#c-interface)

*class* BatchNormTraining : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v017BatchNormTrainingE)

Batchnorm for training operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v017BatchNormTraining13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

BatchNormTraining(*const* Output<Node> &*input*, *const* Output<Node> &*gamma*, *const* Output<Node> &*beta*, double *epsilon*)[¶](#_CPPv4N6ngraph2op2v017BatchNormTraining17BatchNormTrainingERK6OutputI4NodeERK6OutputI4NodeERK6OutputI4NodeEd)

**Parameters**

* input: Must have rank >= 2, [., C, …]
* gamma: gamma scaling for normalized value. [C]
* beta: bias added to the scaled normalized value [C]
* epsilon: Avoids divsion by 0 if input has 0 variance

NGRAPH\_DEPRECATED\_DOC BatchNormTraining(double *eps*, *const* Output<Node> &*gamma*, *const* Output<Node> &*beta*, *const* Output<Node> &*input*)[¶](#_CPPv4N6ngraph2op2v017BatchNormTraining17BatchNormTrainingEdRK6OutputI4NodeERK6OutputI4NodeERK6OutputI4NodeE)

In this version of BatchNorm:

MEAN AND VARIANCE: computed directly from the content of ‘input’.

OUTPUT VALUE: A tuple with the following structure: [0] - The normalization of ‘input’. [1] - The per-channel means of (pre-normalized) ‘input’. [2] - The per-channel variances of (pre-normalized) ‘input’.

AUTODIFF SUPPORT: yes: ‘generate\_adjoints(…)’ works as expected.

SHAPE DETAILS: gamma: must have rank 1, with the same span as input’s channel axis. beta: must have rank 1, with the same span as input’s channel axis. input: must have rank >= 2. The second dimension represents the channel axis and must have a span of at least 1. output[0]: shall have the same shape as ‘input’. output[1]: shall have rank 1, with the same span as input’s channel axis. output[2]: shall have rank 1, with the same span as input’s channel axis.

void validate\_and\_infer\_types()[¶](#_CPPv4N6ngraph2op2v017BatchNormTraining24validate_and_infer_typesEv)

Throws if the node is invalid.

### BatchNormTrainingBackprop[¶](#batchnormtrainingbackprop)

BatchNormTrainingBackprop // Compute mean and variance backprop from the input.

#### Description[¶](#description)

Computes the input, gamma and beta backprop increments.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| input | real | \((\bullet, C, \ldots)\) |
| gamma | same as input | \((C)\) |
| beta | same as input | \((C)\) |
| mean | same as input | \((C)\) |
| variance | same as input | \((C)\) |
| normalized\_delta | same as input | same as input |

##### Attributes[¶](#attributes)

| Name | Type | Notes |
| --- | --- | --- |
| epsilon | double | Small bias added to variance to avoid division by 0. |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| input\_delta | same as input | Same as input |
| gamma\_delta | same as gamma | \((C)\) |
| beta\_delta | same as beta | \((C)\) |

#### Mathematical Definition[¶](#mathematical-definition)

It is easiest to simplify by looking at a single channel and flattening the remaining axes into a vector; so gamma and beta are scalars, and input is an \(N\)-element vector.

The step by step forward training computation is

\[\begin{split}\mathtt{mean} &= \frac{\sum{\mathtt{input}\_i}}{N}\\ \mathtt{centered}\_i &= \mathtt{input}\_i - \mathtt{mean}\\ \mathtt{square}\_i &= \mathtt{centered}\_i^2\\ \mathtt{variance} &= \frac{\sum \mathtt{square}\_i}{N}\\ \mathtt{invsqrt} &= \frac{1}{\sqrt{\mathtt{variance}+\epsilon}}\\ \mathtt{gmul} &= \texttt{gamma}\cdot \mathtt{invsqrt}\\ \mathtt{normed}\_i &= \mathtt{centered}\_i\mathtt{gmul}+\texttt{beta}\end{split}\]

Using the notation \(\overline{\texttt{name}}\) for \(\texttt{name\_delta}\) and \(\overline{x} \leftarrow y\) to mean the backprop value for \(\texttt{x\_delta}\) is a sum that includes \(y\).

We work backwards

\[\begin{split}\overline{\texttt{beta}}&\leftarrow \overline{\texttt{normed}}\\ \overline{\texttt{gmul}}&\leftarrow \sum \overline{\texttt{normed}}\_i\\ \overline{\texttt{centered}}\_i&\leftarrow\overline{\texttt{normed}}\_i\texttt{gmul}\\ \overline{\texttt{gamma}}&\leftarrow \overline{\texttt{gmul}}\cdot\texttt{invsqrt}\\ \overline{\texttt{invsqrt}}&\leftarrow\texttt{gamma}\cdot\overline{\texttt{gmul}}\\ \overline{\texttt{variance}}&\leftarrow -\frac{\overline{\texttt{invsqrt}}\cdot\texttt{invsqrt}}{2\cdot(\texttt{variance}+\epsilon)}\\ \overline{\texttt{square}}\_i&\leftarrow\frac{\overline{\texttt{variance}}}{N}\\ \overline{\texttt{centered}}\_i&\leftarrow 2\cdot\texttt{centered}\_i\cdot\overline{\texttt{square}}\_i\\ \overline{\texttt{input}}\_i&\leftarrow\overline{\texttt{centered}}\_i\\ \overline{\texttt{mean}}&\leftarrow\sum\overline{\texttt{centered}}\_i\\ \overline{\texttt{input}}\_i&\leftarrow\frac{\overline{\texttt{mean}}}{N}\end{split}\]

#### C++ Interface[¶](#c-interface)

*class* BatchNormTrainingBackprop : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v025BatchNormTrainingBackpropE)

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v025BatchNormTrainingBackprop13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

void validate\_and\_infer\_types()[¶](#_CPPv4N6ngraph2op2v025BatchNormTrainingBackprop24validate_and_infer_typesEv)

Throws if the node is invalid.

### Broadcast[¶](#broadcast)

Broadcast // Operation that produces a tensor based on arg's axes

#### Description[¶](#description)

Operation whose output tensor ignores axes not in the arg tensor.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Attributes[¶](#attributes)

| Name | Type | Notes |
| --- | --- | --- |
| shape | Shape | The shape of the output. |
| broadcast\_axes | AxisSet | Axis positions in shape that are broadcast. |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as shape |

The shape of arg must match shape with elements in broadcast\_axes removed.

For example, if arg is \([a, b, c]\) then

\[\begin{split}\mathtt{Broadcast(arg, Shape{2, 3}, AxisSet{0})} &= \begin{bmatrix} a & b & c\\ a & b & c \end{bmatrix}\\ \mathtt{Broadcast(arg, Shape{3, 2}, AxisSet{1})} &= \begin{bmatrix} a & a\\ b & b\\ c & c \end{bmatrix}\end{split}\]

#### Mathematical Definition[¶](#mathematical-definition)

For a coordinate \(C\), let \(p(C)\) be a coordinate with the axes in broadcast\_axes removed. For example, if \(\mathtt{broadcast\_axes}=\{1,3\}\) then \(p([d\_0, d\_1, d\_2, d\_3, d\_4]) = [d\_0, d\_2, d\_4]\). Then

\[\mathtt{output}\_C = \mathtt{arg}\_{p(C)}.\]

#### Backprop[¶](#backprop)

\[\overline{\mathtt{arg}} \leftarrow \mathtt{Sum}(\Delta, \mathtt{broadcast\_axes}).\]

#### C++ Interface[¶](#c-interface)

*class* Broadcast : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v09BroadcastE)

Operation which “adds” axes to an input tensor, replicating elements from the input as needed along the new axes.

Subclassed by ngraph::op::v0::BroadcastLike

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v09Broadcast13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Broadcast()[¶](#_CPPv4N6ngraph2op2v09Broadcast9BroadcastEv)

Constructs a broadcast operation.

Broadcast(*const* Output<Node> &*arg*, *const* Shape &*shape*, *const* AxisSet &*broadcast\_axes*)[¶](#_CPPv4N2op2v09Broadcast9BroadcastERK6OutputI4NodeERK5ShapeRK7AxisSet)

Constructs a broadcast operation.

**Parameters**

* arg: The input tensor to be broadcast.
* shape: The shape of the output tensor.
* broadcast\_axes: The axis positions (0-based) in the result that are being broadcast. The remaining axes in shape must be the same as the shape of arg.

void validate\_and\_infer\_types()[¶](#_CPPv4N2op2v09Broadcast24validate_and_infer_typesEv)

Throws if the node is invalid.

*const* AxisSet &get\_broadcast\_axes() *const*[¶](#_CPPv4NK6ngraph2op2v09Broadcast18get_broadcast_axesEv)

**Return**

A set containing the indices of the broadcast axes (0-based).

### BroadcastDistributed[¶](#broadcastdistributed)

BroadcastDistributed // Collective operation

#### Description[¶](#description)

Broadcast values from a primary root process or device to other processes or devices within the op communicator.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | element::f32 element::f64 | Any |

##### Outputs (in place)[¶](#outputs-in-place)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | element::f32 element::f64 | Same as arg |

#### C++ Interface[¶](#c-interface)

*class* BroadcastDistributed : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v020BroadcastDistributedE)

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v020BroadcastDistributed13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

void validate\_and\_infer\_types()[¶](#_CPPv4N6ngraph2op2v020BroadcastDistributed24validate_and_infer_typesEv)

Throws if the node is invalid.

### Ceiling[¶](#ceiling)

Ceiling // Elementwise ceiling operation

#### Description[¶](#description)

Produces a single output tensor of the same element type and shape as arg, where the value at each coordinate of output is the ceiling of the value at each arg coordinate.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \lceil \mathtt{arg}\_{i\_0, \ldots, i\_{n-1}}\rceil\]

#### Backprop[¶](#backprop)

Not defined by nGraph.

The backprop would be zero for non-integer input and undefined for integer input; a zero backprop would have no effect on the backprop to arg, so there is no need for Ceiling to define a backprop.

#### C++ Interface[¶](#c-interface)

*class* Ceiling : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v07CeilingE)

Elementwise ceiling operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v07Ceiling13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Ceiling()[¶](#_CPPv4N6ngraph2op2v07Ceiling7CeilingEv)

Constructs a ceiling operation.

Ceiling(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v07Ceiling7CeilingERK6OutputI4NodeE)

Constructs a ceiling operation.

**Parameters**

* arg: Node that produces the input tensor.

### Concat[¶](#concat)

Concat // Concatenation operation

#### Description[¶](#description)

Produce from Nodes of args some outputs with the same attributes

##### Inputs[¶](#inputs)

| Name | Type | Notes |
| --- | --- | --- |
| args | Nodes | All element types the same. All shapes the same except on concatenation\_axis |

##### Attributes[¶](#attributes)

| Name | Notes |
| --- | --- |
| concatenation\_axis | Less than the rank of the shape |

##### Outputs[¶](#outputs)

Name

Element Type

Shape

output

Same as args | Same as arg on non-concatenation\_axis

Sum of concatenation\_axis lengths of args

#### Mathematical Definition[¶](#mathematical-definition)

We map each tensor in args to a segment of output based on the coordinate at coordinate\_axis.

Let

\[\begin{split}s(i) &= \sum\_{j<i} \mathtt{args}[i].\mathtt{shape}\left[\mathtt{concatenation\_axis}\right]\\ t(i) &= \text{The greatest }j\text{ such that }i \ge s(j)\\ p(C)\_i &= \begin{cases} C\_i-s(t(i))&\text{if }i==\mathtt{concatenation\_axis}\\ C\_i&\text{otherwise} \end{cases}\\ \mathtt{output}\_C&=\mathtt{args}[t(C\_i)]\_{p(C)}\end{split}\]

#### Backprop[¶](#backprop)

We slice the backprop value into the backprops associated with the inputs.

#### C++ Interface[¶](#c-interface)

*class* Concat : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v06ConcatE)

Concatenation operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v06Concat13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Concat()[¶](#_CPPv4N6ngraph2op2v06Concat6ConcatEv)

Constructs a concatenation operation.

Concat(*const* OutputVector &*args*, int64\_t *axis*)[¶](#_CPPv4N6ngraph2op2v06Concat6ConcatERK12OutputVector7int64_t)

Constructs a concatenation operation.

**Parameters**

* args: The outputs producing the input tensors.
* axis: The axis along which to concatenate the input tensors.

Concat(*const* NodeVector &*args*, int64\_t *axis*)[¶](#_CPPv4N6ngraph2op2v06Concat6ConcatERK10NodeVector7int64_t)

Constructs a concatenation operation.

**Parameters**

* args: The nodes producing the input tensors.
* axis: The axis along which to concatenate the input tensors.

void validate\_and\_infer\_types()[¶](#_CPPv4N6ngraph2op2v06Concat24validate_and_infer_typesEv)

Throws if the node is invalid.

int64\_t get\_concatenation\_axis() *const*[¶](#_CPPv4NK6ngraph2op2v06Concat22get_concatenation_axisEv)

**Return**

The concatenation axis.

int64\_t get\_axis() *const*[¶](#_CPPv4NK6ngraph2op2v06Concat8get_axisEv)

**Return**

The concatenation axis.

### Constant[¶](#constant)

Constant // Literal constant tensor

#### Description[¶](#description)

The output is a tensor initialized from the values attribute.

##### Attributes[¶](#attributes)

| Name | Type | Notes |
| --- | --- | --- |
| type | ngraph::element::type | The element type of the value in the computation |
| shape | ngraph::Shape | The shape of the constant |
| values | const std::vector<T>& | Constant elements in row-major order. T must be compatible with the element type |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | type | shape |

#### C++ Interface[¶](#c-interface)

*class* Constant : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v08ConstantE)

Class for constants.

Subclassed by ngraph::op::v0::ScalarConstantLike

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v08Constant13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

*template* <*typename* T>  
Constant(*const* element::Type &*type*, Shape *shape*, *const* std::vector<T> &*values*)[¶](#_CPPv4N6ngraph2op2v08Constant8ConstantERKN7element4TypeE5ShapeRKNSt6vectorI1TEE)

Constructs a tensor constant.

**Parameters**

* type: The element type of the tensor constant.
* shape: The shape of the tensor constant.
* values: A vector of literals for initializing the tensor constant. The size of values must match the size of the shape.

Constant(*const* element::Type &*type*, Shape *shape*, *const* std::vector<std::string> &*values*)[¶](#_CPPv4N6ngraph2op2v08Constant8ConstantERKN7element4TypeE5ShapeRKNSt6vectorINSt6stringEEE)

Constructs a tensor constant This constructor is mainly to support deserialization of constants.

**Parameters**

* type: The element type of the tensor constant.
* shape: The shape of the tensor constant.
* values: A list of string values to use as the constant data.

Constant(*const* element::Type &*type*, *const* Shape &*shape*, *const* void \**data*)[¶](#_CPPv4N6ngraph2op2v08Constant8ConstantERKN7element4TypeERK5ShapePKv)

Constructs a tensor constant with the supplied data.

**Parameters**

* type: The element type of the tensor constant.
* shape: The shape of the tensor constant.
* data: A void\* to constant data.

void validate\_and\_infer\_types()[¶](#_CPPv4N6ngraph2op2v08Constant24validate_and_infer_typesEv)

Throws if the node is invalid.

Shape get\_shape\_val() *const*[¶](#_CPPv4NK6ngraph2op2v08Constant13get_shape_valEv)

Returns the value of the constant node as a Shape object Can only be used on element::i64 nodes and interprets negative values as zeros.

Strides get\_strides\_val() *const*[¶](#_CPPv4NK6ngraph2op2v08Constant15get_strides_valEv)

Returns the value of the constant node as a Strides object Can only be used on element::i64 nodes and interprets negative values as zeros.

Coordinate get\_coordinate\_val() *const*[¶](#_CPPv4NK6ngraph2op2v08Constant18get_coordinate_valEv)

Returns the value of the constant node as a Coordinate object Can only be used on element::i64 nodes and interprets negative values as zeros.

CoordinateDiff get\_coordinate\_diff\_val() *const*[¶](#_CPPv4NK6ngraph2op2v08Constant23get_coordinate_diff_valEv)

Returns the value of the constant node as a CoordinateDiff object Can only be used on element::i64 nodes.

AxisVector get\_axis\_vector\_val() *const*[¶](#_CPPv4NK6ngraph2op2v08Constant19get_axis_vector_valEv)

Returns the value of the constant node as an AxisVector object Can only be used on element::i64 nodes and interprets negative values as zeros.

AxisSet get\_axis\_set\_val() *const*[¶](#_CPPv4NK6ngraph2op2v08Constant16get_axis_set_valEv)

Returns the value of the constant node as an AxisSet object Can only be used on element::i64 nodes and interprets negative values as zeros. Repeated values are allowed.

std::vector<std::string> get\_value\_strings() *const*[¶](#_CPPv4NK6ngraph2op2v08Constant17get_value_stringsEv)

**Return**

The initialization literals for the tensor constant.

*template* <*typename* T>  
std::vector<T> cast\_vector() *const*[¶](#_CPPv4NK6ngraph2op2v08Constant11cast_vectorEv)

Return the [Constant](index.odt/index.html" \l "classngraph_1_1op_1_1v0_1_1Constant)’s value as a vector cast to type T.

**Return**

[Constant](index.odt/index.html" \l "classngraph_1_1op_1_1v0_1_1Constant)’s data vector.

**Template Parameters**

* T: Type to which data vector’s entries will be cast.

Public Static Functions

*template* <*typename* T>  
*static* std::shared\_ptr<op::v0::[Constant](index.odt/index.html" \l "_CPPv4N6ngraph2op2v08ConstantE)> create(*const* element::Type &*type*, Shape *shape*, *const* std::vector<T> *values*)[¶](#_CPPv4N6ngraph2op2v08Constant6createERKN7element4TypeE5ShapeKNSt6vectorI1TEE)

Wrapper around constructing a shared\_ptr of a [Constant](index.odt/index.html" \l "classngraph_1_1op_1_1v0_1_1Constant).

**Parameters**

* type: The element type of the tensor constant.
* shape: The shape of the tensor constant.
* values: A vector of values to use as the constant data.

*template* <*typename* T>  
*static* std::shared\_ptr<op::v0::[Constant](index.odt/index.html" \l "_CPPv4N6ngraph2op2v08ConstantE)> create(*const* element::Type &*type*, Shape *shape*, std::initializer\_list<T> *values*)[¶](#_CPPv4N6ngraph2op2v08Constant6createERKN7element4TypeE5ShapeNSt16initializer_listI1TEE)

Wrapper around constructing a shared\_ptr of a [Constant](index.odt/index.html" \l "classngraph_1_1op_1_1v0_1_1Constant).

**Parameters**

* type: The element type of the tensor constant.
* shape: The shape of the tensor constant.
* values: An initializer\_list of values to use as the constant data.

### Convert[¶](#convert)

Convert // Convert a tensor from one element type to another

#### Description[¶](#description)

Long description

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Attributes[¶](#attributes)

| Name | Type | Notes |
| --- | --- | --- |
| element\_type | ngraph::element::type | The element type of the result |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | element\_type | Same as arg |

#### Backprop[¶](#backprop)

\[\overline{\mathtt{arg}} \leftarrow \mathtt{Convert}(\Delta,\mathtt{arg->get\_element\_type()})\]

#### C++ Interface[¶](#c-interface)

*class* Convert : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v07ConvertE)

Elementwise type conversion operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v07Convert13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Convert()[¶](#_CPPv4N6ngraph2op2v07Convert7ConvertEv)

Constructs a conversion operation.

Convert(*const* Output<Node> &*arg*, *const* ngraph::element::Type &*destination\_type*)[¶](#_CPPv4N6ngraph2op2v07Convert7ConvertERK6OutputI4NodeERKN6ngraph7element4TypeE)

Constructs a conversion operation.

**Parameters**

* arg: Node that produces the input tensor.
* destination\_type: Element type for the output tensor.

void validate\_and\_infer\_types()[¶](#_CPPv4N6ngraph2op2v07Convert24validate_and_infer_typesEv)

Throws if the node is invalid.

### Convolution[¶](#convolution)

Convolution // A batched convolution operation

#### Description[¶](#description)

Long description

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| image\_batch | Any | (N, C\_in, d\_1, ..., d\_n) |
| filters | Same as image\_batch | (N, C\_in, df\_1, ..., df\_n) |

##### Attributes[¶](#attributes)

| Name | Type | Notes |
| --- | --- | --- |
| window\_movement\_strides | Strides[n] | How far to slide the window along each axis at each step |
| window\_dilation\_strides | Strides[n] | Per-axis dilation to apply to the filters |
| padding\_below | Shape[n] | How many padding elements to add below the 0-coordinate on each axis |
| padding\_above | Shape[n] | How many padding elements to add above the max-coordinate on each axis |
| image\_dilation\_strides | Strides[n] | Per-axis dilation to apply to the image batch |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| features\_out | Same as image\_batch | (N, C\_in, d\_1 - df\_1 + 1, ..., d\_n - df\_n + 1) |

It must be the case that after dilation and padding are applied, the filter fits within the image.

#### Mathematical Definition[¶](#mathematical-definition)

##### Padding[¶](#padding)

Let \(p\) (the padding below) and \(q\) (the padding above) be a sequence of \(n\) integers, and \(T\) be a tensor of shape \((d\_1,\dots,d\_n)\), such that for all \(i\), \(p\_i + d\_i + q\_i \ge 0\). Then \(\mathit{Pad}[p,q](T)\) is the tensor of shape \((p\_1 + d\_1 + q\_1,\dots,p\_n + d\_n + q\_n)\) such that

\[\begin{split}\mathit{Pad}[p,q](T)\_{i\_1,\dots,i\_n} \triangleq \begin{cases} T\_{i\_1 - p\_1,\dots,i\_n - p\_n} &\mbox{if for all }j, i\_j \ge p\_j\mbox{ and }i\_j < p\_j + d\_j \\ 0 &\mbox{otherwise.} \end{cases}\end{split}\]

##### Dilation[¶](#dilation)

Let \(l\) (the dilation strides) be a sequence of \(n\) positive integers, and \(T\) be a tensor of shape \((d\_1,\dots,d\_n)\). Then \(\mathit{Dilate}[l](T)\) is the tensor of shape \((d'\_1,\dots,d'\_n)\) where \(d'\_i = \mathit{max}(0,l\_i(d\_i - 1) + 1)\) such that

\[\begin{split}\mathit{Dilate}[l](T)\_{i\_1,\dots,i\_n} \triangleq \begin{cases} T\_{i\_1/l\_1,\dots,i\_n/l\_n} &\mbox{if for all }j, i\_j\mbox{ is a multiple of }l\_j \\ 0 &\mbox{otherwise.} \end{cases}\end{split}\]

##### Striding[¶](#striding)

Let \(s\) (the strides) be a sequence of \(n\) positive integers, and \(T\) be a tensor of shape \((d\_1,\dots,d\_n)\). Then \(\mathit{Stride}[s](T)\) is the tensor of shape \((d'\_1,\dots,d'\_n)\) where \(d'\_i = \left\lceil \frac{d\_i}{s\_i} \right\rceil\) such that

\[\mathit{Stride}[s](T)\_{i\_1,\dots,i\_n} \triangleq T\_{s\_1i\_1,\dots,s\_ni\_n}\]

\(s\) is the how far, not the unit of farness.

##### Convolution[¶](#id1)

##### Padded, Dilated, Strided Convolution[¶](#padded-dilated-strided-convolution)

\[\mathit{PDSConv}[g,p,q,l,s](T\_\mathit{image},T\_\mathit{filter} \triangleq \mathit{Stride}[s](\mathit{Conv}(\mathit{Pad}[p,q](\mathit{Dilate}[g](T\_\mathit{batch})),\mathit{Dilate}[l](T\_\mathit{filter})))\]

##### Batched, Padded, Dilated, Strided Convolution[¶](#batched-padded-dilated-strided-convolution)

#### C++ Interface[¶](#c-interface)

*class* Convolution : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v011ConvolutionE)

Batched convolution operation, with optional window dilation and stride.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v011Convolution13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Convolution()[¶](#_CPPv4N6ngraph2op2v011Convolution11ConvolutionEv)

Constructs a batched convolution operation.

Convolution(*const* Output<Node> &*data\_batch*, *const* Output<Node> &*filters*, *const* Strides &*window\_movement\_strides*, *const* Strides &*window\_dilation\_strides*, *const* CoordinateDiff &*padding\_below*, *const* CoordinateDiff &*padding\_above*, *const* Strides &*data\_dilation\_strides*, *const* PadType &*pad\_type* = PadType::EXPLICIT)[¶](#_CPPv4N2op2v011Convolution11ConvolutionERK6OutputI4NodeERK6OutputI4NodeERK7StridesRK7StridesRK14CoordinateDiffRK14CoordinateDiffRK7StridesRK7PadType)

Constructs a batched convolution operation.

Output [N, C\_OUT, R1, ... Rf]

**Parameters**

* data\_batch: The node producing the input data batch tensor.[N, C\_IN, D1, ... Df]
* filters: The node producing the filters tensor.[C\_OUT, C\_IN, F1, ... Ff]
* window\_movement\_strides: The window movement strides.[f]
* window\_dilation\_strides: The window dilation strides.[f]
* padding\_below: The padding-below sizes.[f]
* padding\_above: The padding-above sizes.[f]
* data\_dilation\_strides: The data dilation strides.[f]
* pad\_type: The pad type for automatically computing padding sizes.[f]

Convolution(*const* Output<Node> &*data\_batch*, *const* Output<Node> &*filters*, *const* Strides &*window\_movement\_strides*, *const* Strides &*window\_dilation\_strides*, *const* CoordinateDiff &*padding\_below*, *const* CoordinateDiff &*padding\_above*)[¶](#_CPPv4N2op2v011Convolution11ConvolutionERK6OutputI4NodeERK6OutputI4NodeERK7StridesRK7StridesRK14CoordinateDiffRK14CoordinateDiff)

Constructs a batched convolution operation with no data dilation (i.e., all data dilation strides are 1).

Output [N, C\_OUT, R1, ... Rf]

**Parameters**

* data\_batch: The node producing the input data batch tensor.[N, C\_IN, D1, ... Df]
* filters: The node producing the filters tensor.[C\_OUT, C\_IN, F1, ... Ff]
* window\_movement\_strides: The window movement strides.[f]
* window\_dilation\_strides: The window dilation strides.[f]
* padding\_below: The padding-below sizes.[f]
* padding\_above: The padding-above sizes.[f]

Convolution(*const* Output<Node> &*data\_batch*, *const* Output<Node> &*filters*, *const* Strides &*window\_movement\_strides*, *const* Strides &*window\_dilation\_strides*)[¶](#_CPPv4N2op2v011Convolution11ConvolutionERK6OutputI4NodeERK6OutputI4NodeERK7StridesRK7Strides)

Constructs a batched convolution operation with no padding or data dilation (i.e., padding above and below are 0 everywhere, and all data dilation strides are 1).

Output [N, C\_OUT, R1, ... Rf]

**Parameters**

* data\_batch: The node producing the input data batch tensor.[N, C\_IN, D1, ... Df]
* filters: The node producing the filters tensor.[C\_OUT, C\_IN, F1, ... Ff]
* window\_movement\_strides: The window movement strides.[f]
* window\_dilation\_strides: The window dilation strides.[f]

Convolution(*const* Output<Node> &*data\_batch*, *const* Output<Node> &*filters*, *const* Strides &*window\_movement\_strides*)[¶](#_CPPv4N2op2v011Convolution11ConvolutionERK6OutputI4NodeERK6OutputI4NodeERK7Strides)

Constructs a batched convolution operation with no window dilation, padding, or data dilation (i.e., padding above and below are 0 everywhere, and all window/data dilation strides are 1).

Output [N, C\_OUT, R1, ... Rf]

**Parameters**

* data\_batch: The node producing the input data batch tensor.[N, C\_IN, D1, ... Df]
* filters: The node producing the filters tensor.[C\_OUT, C\_IN, F1, ... Ff]
* window\_movement\_strides: The window movement strides.[f]

Convolution(*const* Output<Node> &*data\_batch*, *const* Output<Node> &*filters*)[¶](#_CPPv4N2op2v011Convolution11ConvolutionERK6OutputI4NodeERK6OutputI4NodeE)

Constructs a batched convolution operation with no window dilation or movement stride (i.e., padding above and below are 0 everywhere, and all window/data dilation strides and window movement strides are 1).

Output [N, C\_OUT, R1, ... Rf]

**Parameters**

* data\_batch: The node producing the input data batch tensor.[N, C\_IN, D1, ... Df]
* filters: The node producing the filters tensor.[C\_OUT, C\_IN, F1, ... Ff]

void validate\_and\_infer\_types()[¶](#_CPPv4N2op2v011Convolution24validate_and_infer_typesEv)

Throws if the node is invalid.

*const* Strides &get\_window\_movement\_strides() *const*[¶](#_CPPv4NK6ngraph2op2v011Convolution27get_window_movement_stridesEv)

**Return**

The window movement strides.

*const* Strides &get\_window\_dilation\_strides() *const*[¶](#_CPPv4NK6ngraph2op2v011Convolution27get_window_dilation_stridesEv)

**Return**

The window dilation strides.

*const* CoordinateDiff &get\_padding\_below() *const*[¶](#_CPPv4NK6ngraph2op2v011Convolution17get_padding_belowEv)

**Return**

The padding-below sizes (possibly negative).

*const* CoordinateDiff &get\_padding\_above() *const*[¶](#_CPPv4NK6ngraph2op2v011Convolution17get_padding_aboveEv)

**Return**

The padding-above sizes (possibly negative).

*const* Strides &get\_data\_dilation\_strides() *const*[¶](#_CPPv4NK6ngraph2op2v011Convolution25get_data_dilation_stridesEv)

**Return**

The input data dilation strides.

*const* PadType &get\_pad\_type() *const*[¶](#_CPPv4NK6ngraph2op2v011Convolution12get_pad_typeEv)

**Return**

The pad type for convolution.

shared\_ptr<Node> get\_default\_value() *const*[¶](#_CPPv4NK2op2v011Convolution17get_default_valueEv)

**Return**

The default value for [Convolution](index.odt/index.html" \l "classngraph_1_1op_1_1v0_1_1Convolution).

### Cos[¶](#cos)

Cos // Elementwise cosine operation

#### Description[¶](#description)

Produces a tensor of the same element type and shape as arg, where the value at each coordinate of output is the cosine of the value at the corresponding coordinate of arg.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \cos(\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}})\]

#### Backprop[¶](#backprop)

\[\overline{\mathtt{arg}} \leftarrow -\Delta\ \sin(\mathtt{arg})\]

#### C++ Interface[¶](#c-interface)

*class* Cos : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v03CosE)

Elementwise cosine operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v03Cos13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Cos()[¶](#_CPPv4N6ngraph2op2v03Cos3CosEv)

Constructs a cosine operation.

Cos(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v03Cos3CosERK6OutputI4NodeE)

Constructs a cosine operation.

**Parameters**

* arg: Node that produces the input tensor.

### Cosh[¶](#cosh)

Cosh // Elementwise hyperbolic cosine operation

#### Description[¶](#description)

Produces a tensor of the same element type and shape as arg, where the value at each coordinate of output is the hyperbolic cosine of the value at the corresponding coordinate of arg.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \cosh(\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}})\]

#### Backprop[¶](#backprop)

\[\overline{\mathtt{arg}} \leftarrow \Delta\ \sinh(\mathtt{arg})\]

#### C++ Interface[¶](#c-interface)

*class* Cosh : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v04CoshE)

Elementwise hyperbolic cosine (cosh) operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v04Cosh13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Cosh()[¶](#_CPPv4N6ngraph2op2v04Cosh4CoshEv)

Constructs a hyperbolic cosine operation.

Cosh(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v04Cosh4CoshERK6OutputI4NodeE)

Constructs a hyperbolic cosine operation.

**Parameters**

* arg: Node that produces the input tensor.

### Dequantize[¶](#dequantize)

Dequantize // Maps quantized input to real output

#### Description[¶](#description)

Produces a tensor of element type type and the same shape as input where the value of each coordinate \(i\) of output is the corresponding coordinate of input minus zero\_point quantity multiplied by scale. The coordinate \(j\) of scale and zero\_point is the coordinate of output projected onto axes.

##### Inputs[¶](#inputs)

Name

Element Type

Shape

input

Any quantized type

Any

scale

Same as output

input shape projected onto axes

zero\_point | Same as input | input shape projected onto axes

##### Attributes[¶](#attributes)

| Name | Description |
| --- | --- |
| type | output element type; any real type |
| axes | Axis positions on which scale and zero\_point are specified |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | type | Same as input |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i,j} = (\mathtt{input}\_{i,j} - \mathtt{zero\_point}\_{j}) \mathtt{scale}\_{j}\]

#### C++ Interface[¶](#c-interface)

*class* Dequantize : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op10DequantizeE)

[Dequantize](index.odt/index.html" \l "classngraph_1_1op_1_1Dequantize) operation Maps quantized input (q) to real output (r) using scale (s) and zero point (z): r = (q - o) \* s.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op10Dequantize13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Dequantize()[¶](#_CPPv4N6ngraph2op10Dequantize10DequantizeEv)

Constructs a [Dequantize](index.odt/index.html" \l "classngraph_1_1op_1_1Dequantize) operation.

Dequantize(*const* Output<Node> &*input*, *const* Output<Node> &*scale*, *const* Output<Node> &*zero\_point*, *const* element::Type &*type*, *const* AxisSet &*axes*)[¶](#_CPPv4N2op10Dequantize10DequantizeERK6OutputI4NodeERK6OutputI4NodeERK6OutputI4NodeERKN7element4TypeERK7AxisSet)

Constructs a [Dequantize](index.odt/index.html" \l "classngraph_1_1op_1_1Dequantize) operation.

**Parameters**

* input: quantized input
* scale: scale used for mapping
* zero\_point: zero point used for mapping
* type: output element type
* axes: axis positions on which scale and zero\_point are specified

void validate\_and\_infer\_types()[¶](#_CPPv4N2op10Dequantize24validate_and_infer_typesEv)

Throws if the node is invalid.

### Divide[¶](#divide)

Divide // Elementwise divide operation

#### Description[¶](#description)

Produces tensor of the same element type and shape as the two inputs, where the value at each coordinate of output is the quotient of the values at the corresponding input coordinates.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg0 | any | any |
| arg1 | same as arg0 | same as arg0 |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | same as arg0 | same as arg0 |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \frac{\mathtt{arg0}\_{i\_0, \ldots, i\_{n-1}}}{\mathtt{arg1}\_{i\_0, \ldots, i\_{n-1}}}\]

#### Backprop[¶](#backprop)

\[\begin{split}\overline{\mathtt{arg0}} &\leftarrow \frac{\Delta}{\mathtt{arg1}}\\ \overline{\mathtt{arg1}} &\leftarrow -\Delta \frac{\mathtt{Output}}{\mathtt{arg1}}\end{split}\]

#### C++ Interface[¶](#c-interface)

*class* Divide : *public* ngraph::op::util::BinaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v06DivideE)

Elementwise division operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v06Divide13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Divide()[¶](#_CPPv4N6ngraph2op2v06Divide6DivideEv)

Constructs a division operation.

Divide(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, bool *pythondiv*, *const* AutoBroadcastSpec &*auto\_broadcast* = AutoBroadcastSpec())[¶](#_CPPv4N2op2v06Divide6DivideERK6OutputI4NodeERK6OutputI4NodeEbRK17AutoBroadcastSpec)

Constructs a division operation.

**Parameters**

* arg0: Node that produces the first input tensor.
* arg1: Node that produces the second input tensor.
* pythondiv: Use Python style rounding for integral type
* auto\_broadcast: Auto broadcast specification

Divide(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, *const* AutoBroadcastSpec &*auto\_broadcast* = AutoBroadcastSpec())[¶](#_CPPv4N2op2v06Divide6DivideERK6OutputI4NodeERK6OutputI4NodeERK17AutoBroadcastSpec)

Constructs a division operation.

**Parameters**

* arg0: Node that produces the first input tensor.
* arg1: Node that produces the second input tensor.
* auto\_broadcast: Auto broadcast specification

### Dot[¶](#dot)

Dot // Generalized dot product operation

#### Description[¶](#description)

Generalized dot product operation, including scalar-tensor product, matrix-vector product, and matrix multiplication.

A few common cases are as follows:

* If \(m = 0\) and \(n = 1\) or \(p = 1\), the operation is a scalar-tensor product.
* If \(m = 1\), \(n = 2\), and \(p = 1\), the operation is a matrix-vector product.
* If \(m = 1\) and \(n = p = 2\), the operation is a matrix multiplication.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg0 | any | \((i\_1,\dots,i\_n,j\_1,\dots,j\_m)\) |
| arg1 | same as arg0 | \((j\_1,\ldots,j\_m,k\_1,\dots,k\_p)\) |

##### Attributes[¶](#attributes)

| Name |  |  |
| --- | --- | --- |
| reduction\_axes\_count | size\_t | The number of axes to reduce through dot-product (corresponds to \(m\) in the formulas above) |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | same as arg0 | \((i\_1,\ldots,i\_n,k\_1,\dots,k\_p)\) |

#### Mathematical Definition[¶](#mathematical-definition)

\[\begin{split}\mathtt{output}\_{i\_1,\dots,i\_n,k\_1,\ldots,k\_p} = \begin{cases} \mathtt{arg0}\_{i\_1,\dots,i\_n} \cdot \mathtt{arg1}\_{k\_1,\dots,k\_p}&\text{if }m=0,\\ \sum\_{j\_1, \ldots, j\_m} \mathtt{arg0}\_{i\_1,\dots,i\_n,j\_1,\dots,j\_m} \cdot \mathtt{arg1}\_{j\_1,\ldots,j\_m,k\_1,\ldots,k\_p} &\text{otherwise}. \end{cases}\end{split}\]

#### Backprop[¶](#backprop)

To be documented.

#### C++ Interface[¶](#c-interface)

*class* Dot : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v03DotE)

Generalized dot product operation, including scalar-tensor product, matrix-vector product, and matrix multiplication.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v03Dot13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Dot()[¶](#_CPPv4N6ngraph2op2v03Dot3DotEv)

Constructs a dot product operation.

Dot(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, size\_t *reduction\_axes\_count*, bool *has\_reduction\_axes\_count* = true)[¶](#_CPPv4N6ngraph2op2v03Dot3DotERK6OutputI4NodeERK6OutputI4NodeE6size_tb)

Constructs a dot product operation.

**Parameters**

* arg0: The node producing the first argument.
* arg1: The node producing the second argument.
* reduction\_axes\_count: The number of axes to dot.

Dot(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*)[¶](#_CPPv4N6ngraph2op2v03Dot3DotERK6OutputI4NodeERK6OutputI4NodeE)

Constructs a dot product operation with default dot-axis selection depending on the inputs.

If arg0 or arg1 is a scalar, there are no dot-axes. Else, there is one dot-axis.

(Note that in particular, this results in scalar-tensor products where one or the other argument is a scalar, a matrix-vector products where arg0 is a matrix and arg1 is a vector, and a matrix multiplication where arg0 and arg1 are both matrices.)

**Parameters**

* arg0: The node producing the first argument.
* arg1: The node producing the second argument.

void validate\_and\_infer\_types()[¶](#_CPPv4N6ngraph2op2v03Dot24validate_and_infer_typesEv)

Throws if the node is invalid.

### DropOut[¶](#dropout)

DropOut // DropOut

#### Description[¶](#description)

##### Inputs[¶](#inputs)

|  |  |  |
| --- | --- | --- |
|  |  |  |

##### Attributes[¶](#attributes)

| Name | Description |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output |  |  |

#### Mathematical Definition[¶](#mathematical-definition)

#### C++ Interface[¶](#c-interface)

### Equal[¶](#equal)

Equal // Elementwise equal operation

#### Description[¶](#description)

Produces tensor of the same element type and shape as the two inputs, where the value at each coordinate of output is 1 (true) if arg0 is equal to arg1, 0 otherwise.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg0 | any | any |
| arg1 | same as arg0 | same as arg0 |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | ngraph::element::boolean | same as arg0 |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \mathtt{arg0}\_{i\_0, \ldots, i\_{n-1}} == \mathtt{arg1}\_{i\_0, \ldots, i\_{n-1}}\]

#### C++ Interface[¶](#c-interface)

*class* Equal : *public* ngraph::op::util::BinaryElementwiseComparison[¶](#_CPPv4N6ngraph2op2v05EqualE)

Elementwise is-equal operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v05Equal13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Equal()[¶](#_CPPv4N6ngraph2op2v05Equal5EqualEv)

Constructs an equal operation.

Equal(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, *const* AutoBroadcastSpec &*auto\_broadcast* = AutoBroadcastSpec())[¶](#_CPPv4N2op2v05Equal5EqualERK6OutputI4NodeERK6OutputI4NodeERK17AutoBroadcastSpec)

Constructs an equal operation.

**Parameters**

* arg0: Node that produces the first input tensor.
* arg1: Node that produces the second input tensor.
* auto\_broadcast: Auto broadcast specification

### Exp[¶](#exp)

Exp // Elementwise expine operation

#### Description[¶](#description)

Produces a tensor of the same element type and shape as arg, where the value at each coordinate of output is the expine of the value at the corresponding coordinate of arg.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \exp(\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}})\]

#### Backprop[¶](#backprop)

\[\overline{\mathtt{arg}} \leftarrow \Delta\ \mathtt{output}\]

#### C++ Interface[¶](#c-interface)

*class* Exp : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v03ExpE)

Elementwise natural exponential (exp) operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v03Exp13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Exp()[¶](#_CPPv4N6ngraph2op2v03Exp3ExpEv)

Constructs an exponential operation.

Exp(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v03Exp3ExpERK6OutputI4NodeE)

Constructs an exponential operation.

**Parameters**

* arg: Node that produces the input tensor.

### Floor[¶](#floor)

Floor // Elementwise floor operation

#### Description[¶](#description)

Produces a single output tensor of the same element type and shape as arg, where the value at each coordinate of output is the floor of the value at each arg coordinate.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \lfloor \mathtt{arg}\_{i\_0, \ldots, i\_{n-1}}\rfloor\]

#### Backprop[¶](#backprop)

Not defined by nGraph.

The backprop would be zero for non-integer input and undefined for integer input; a zero backprop would have no effect on the backprop to arg, so there is no need for Floor to define a backprop.

#### C++ Interface[¶](#c-interface)

*class* Floor : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v05FloorE)

Elementwise floor operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v05Floor13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Floor()[¶](#_CPPv4N6ngraph2op2v05Floor5FloorEv)

Constructs a floor operation.

Floor(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v05Floor5FloorERK6OutputI4NodeE)

Constructs a floor operation.

**Parameters**

* arg: Node that produces the input tensor.

### GetOutputElement[¶](#getoutputelement)

GetOutputElement // Operation to select a unique output from an op

#### Description[¶](#description)

Accesses an output of a node.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Attributes[¶](#attributes)

| Name | Description |
| --- | --- |
| n | The output number from the node arg |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Depends on arg | Depends on arg |

#### C++ Interface[¶](#c-interface)

*class* GetOutputElement : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v016GetOutputElementE)

Operation to get an output from a node.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v016GetOutputElement13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

GetOutputElement(*const* std::shared\_ptr<Node> &*arg*, size\_t *n*)[¶](#_CPPv4N6ngraph2op2v016GetOutputElement16GetOutputElementERKNSt10shared_ptrI4NodeEE6size_t)

Constructs a get-tuple-element operation.

**Parameters**

* arg: The input tuple.
* n: The index of the tuple element to get.

Output<Node> get\_as\_output() *const*[¶](#_CPPv4NK6ngraph2op2v016GetOutputElement13get_as_outputEv)

Return the equilent Output<Node>

void validate\_and\_infer\_types()[¶](#_CPPv4N6ngraph2op2v016GetOutputElement24validate_and_infer_typesEv)

Throws if the node is invalid.

size\_t get\_n() *const*[¶](#_CPPv4NK6ngraph2op2v016GetOutputElement5get_nEv)

**Return**

The index of the tuple element to get.

### GreaterEq[¶](#greatereq)

GreaterEq // Elementwise greater or equal operation

#### Description[¶](#description)

Produces tensor of the same element type and shape as the two inputs, where the value at each coordinate of output is true (1) if arg0 is greater than or equal to arg1, 0 otherwise.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg0 | any | any |
| arg1 | same as arg0 | same as arg0 |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | ngraph::element::boolean | same as arg0 |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \mathtt{arg0}\_{i\_0, \ldots, i\_{n-1}} \ge \mathtt{arg1}\_{i\_0, \ldots, i\_{n-1}}\]

#### C++ Interface[¶](#c-interface)

*class* GreaterEq : *public* ngraph::op::util::BinaryElementwiseComparison[¶](#_CPPv4N6ngraph2op2v09GreaterEqE)

Elementwise greater-than-or-equal operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v09GreaterEq13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

GreaterEq()[¶](#_CPPv4N6ngraph2op2v09GreaterEq9GreaterEqEv)

Constructs a greater-than-or-equal operation.

GreaterEq(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, *const* AutoBroadcastSpec &*auto\_broadcast* = AutoBroadcastSpec())[¶](#_CPPv4N2op2v09GreaterEq9GreaterEqERK6OutputI4NodeERK6OutputI4NodeERK17AutoBroadcastSpec)

Constructs a greater-than-or-equal operation.

**Parameters**

* arg0: Node that produces the first input tensor.
* arg1: Node that produces the second input tensor.
* auto\_broadcast: Auto broadcast specification

### Greater[¶](#greater)

Greater // Elementwise greater operation

#### Description[¶](#description)

Produces tensor of the same element type and shape as the two inputs, where the value at each coordinate of output is true (1) if arg0 is greater than arg1, 0 otherwise.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg0 | any | any |
| arg1 | same as arg0 | same as arg0 |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | ngraph::element::boolean | same as arg0 |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \mathtt{arg0}\_{i\_0, \ldots, i\_{n-1}} > \mathtt{arg1}\_{i\_0, \ldots, i\_{n-1}}\]

#### C++ Interface[¶](#c-interface)

*class* Greater : *public* ngraph::op::util::BinaryElementwiseComparison[¶](#_CPPv4N6ngraph2op2v07GreaterE)

Elementwise greater-than operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v07Greater13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Greater()[¶](#_CPPv4N6ngraph2op2v07Greater7GreaterEv)

Constructs a greater-than operation.

Greater(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, *const* AutoBroadcastSpec &*auto\_broadcast* = AutoBroadcastSpec())[¶](#_CPPv4N2op2v07Greater7GreaterERK6OutputI4NodeERK6OutputI4NodeERK17AutoBroadcastSpec)

Constructs a greater-than operation.

**Parameters**

* arg0: Node that produces the first input tensor.
* arg1: Node that produces the second input tensor.
* auto\_broadcast: Auto broadcast specification

### LessEq[¶](#lesseq)

LessEq // Elementwise less or equal operation

#### Description[¶](#description)

Produces tensor of the same element type and shape as the two inputs, where the value at each coordinate of output is true (1) if arg0 is less than or equal to arg1, 0 otherwise.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg0 | any | any |
| arg1 | same as arg0 | same as arg0 |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | ngraph::element::boolean | same as arg0 |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \mathtt{arg0}\_{i\_0, \ldots, i\_{n-1}} \le \mathtt{arg1}\_{i\_0, \ldots, i\_{n-1}}\]

#### C++ Interface[¶](#c-interface)

*class* LessEq : *public* ngraph::op::util::BinaryElementwiseComparison[¶](#_CPPv4N6ngraph2op2v06LessEqE)

Elementwise less-than-or-equal operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v06LessEq13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

LessEq()[¶](#_CPPv4N6ngraph2op2v06LessEq6LessEqEv)

Constructs a less-than-or-equal operation.

LessEq(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, *const* AutoBroadcastSpec &*auto\_broadcast* = AutoBroadcastSpec())[¶](#_CPPv4N2op2v06LessEq6LessEqERK6OutputI4NodeERK6OutputI4NodeERK17AutoBroadcastSpec)

Constructs a less-than-or-equal operation.

**Parameters**

* arg0: Node that produces the first input tensor.
* arg1: Node that produces the second input tensor.
* auto\_broadcast: Auto broadcast specification

### Less[¶](#less)

Less // Elementwise less operation

#### Description[¶](#description)

Produces tensor of the same element type and shape as the two inputs, where the value at each coordinate of output is true (1) if arg0 is less than arg1, 0 otherwise.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg0 | any | any |
| arg1 | same as arg0 | same as arg0 |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | ngraph::element::boolean | same as arg0 |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \mathtt{arg0}\_{i\_0, \ldots, i\_{n-1}} < \mathtt{arg1}\_{i\_0, \ldots, i\_{n-1}}\]

#### C++ Interface[¶](#c-interface)

*class* Less : *public* ngraph::op::util::BinaryElementwiseComparison[¶](#_CPPv4N6ngraph2op2v04LessE)

Elementwise less-than operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v04Less13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Less()[¶](#_CPPv4N6ngraph2op2v04Less4LessEv)

Constructs a less-than operation.

Less(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, *const* AutoBroadcastSpec &*auto\_broadcast* = AutoBroadcastSpec())[¶](#_CPPv4N2op2v04Less4LessERK6OutputI4NodeERK6OutputI4NodeERK17AutoBroadcastSpec)

Constructs a less-than operation.

**Parameters**

* arg0: Node that produces the first input tensor.
* arg1: Node that produces the second input tensor.
* auto\_broadcast: Auto broadcast specification

### Log[¶](#log)

Log // Elementwise logine operation

#### Description[¶](#description)

Produces a tensor of the same element type and shape as arg, where the value at each coordinate of output is the logine of the value at the corresponding coordinate of arg.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \log(\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}})\]

#### Backprop[¶](#backprop)

\[\overline{\mathtt{arg}} \leftarrow \frac{\Delta}{\mathtt{input}}\]

#### C++ Interface[¶](#c-interface)

*class* Log : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v03LogE)

Elementwise natural log operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v03Log13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Log()[¶](#_CPPv4N6ngraph2op2v03Log3LogEv)

Constructs a natural log operation.

Log(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v03Log3LogERK6OutputI4NodeE)

Constructs a natural log operation.

**Parameters**

* arg: Node that produces the input tensor.

### Max[¶](#max)

Max // Max reduction

#### Description[¶](#description)

Reduces the tensor, eliminating the specified reduction axes by taking the maximum element.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | \((d\_1,\dots,d\_n)~(n \geq 0)\) |

##### Attributes[¶](#attributes)

| Name | Description |
| --- | --- |
| reduction\_axes | The axis positions (0-based) on which to calculate the max |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | \((d\_i:i\not\in \mathtt{reduction\_axes})\) |

#### C++ Interface[¶](#c-interface)

*class* Max : *public* ngraph::op::util::ArithmeticReduction[¶](#_CPPv4N6ngraph2op2v03MaxE)

Max-reduction operation.

### Maximum[¶](#maximum)

Maximum // Elementwise maximum operation

#### Description[¶](#description)

Produces tensor of the same element type and shape as the two inputs, where the value at each coordinate of output is the maximum of the values at the corresponding input coordinates.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg0 | any | any |
| arg1 | same as arg0 | same as arg0 |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | same as arg0 | same as arg0 |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \max(\mathtt{arg0}\_{i\_0, \ldots, i\_{n-1}}, \mathtt{arg1}\_{i\_0, \ldots, i\_{n-1}})\]

#### Backprop[¶](#backprop)

\[\begin{split}\overline{\mathtt{arg0}} &\leftarrow \mathtt{Greater}(\mathtt{arg0}, \mathtt{arg1})\ \Delta \\ \overline{\mathtt{arg1}} &\leftarrow \mathtt{Greater}(\mathtt{arg1}, \mathtt{arg0})\ \Delta\end{split}\]

#### C++ Interface[¶](#c-interface)

*class* Maximum : *public* ngraph::op::util::BinaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v07MaximumE)

Elementwise maximum operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v07Maximum13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Maximum()[¶](#_CPPv4N6ngraph2op2v07Maximum7MaximumEv)

Constructs a maximum operation.

Maximum(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, *const* AutoBroadcastSpec &*auto\_broadcast* = AutoBroadcastSpec())[¶](#_CPPv4N2op2v07Maximum7MaximumERK6OutputI4NodeERK6OutputI4NodeERK17AutoBroadcastSpec)

Constructs a maximum operation.

**Parameters**

* arg0: Node that produces the first input tensor.
* arg1: Node that produces the second input tensor.
* auto\_broadcast: Auto broadcast specification

### MaxPool[¶](#maxpool)

MaxPool // MaxPool operations

#### Description[¶](#description)

Batched max pooling operation, with optional padding and window stride.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | any | \((N, C, d\_1, \ldots, d\_n)\) |

##### Attributes[¶](#attributes)

| Name | Description |
| --- | --- |
| window\_shape | The window shape. |
| window\_movement\_strides | The window movement strides. (defaults to 1s) |
| padding\_below | The below-padding shape. (defaults to 0s) |
| padding\_above | The above-padding shape. (defaults to 0s) |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | same as arg | \((N,C,d'\_1,\ldots,d'\_n)\) |

The input for max pooling is a data batch tensor of shape \((N,C,d\_1,\dots,d\_n)\) where \(n > 0\), every \(d\_i > 0\), and where \(N\) is the batch size, and \(C > 0\) is the number of channels (sometimes called features). The dimensions \((d\_1,\dots,d\_n)\) correspond to the shape of an \(n\)-dimensional data item in a batch. For example, where \(n=2\), the data may represent a two-dimensional image. It also has two attributes:

1. *the window shape* a size vector \((w\_1,\ldots,w\_n)\) where every \(w\_i \le d\_i\); and
2. *the window movement strides, optional* a vector of positive integers \((s\_1,\dots,s\_n)\).

The output has the shape \((N,C,d'\_1,\ldots,d'\_n)\), where \(d'\_n = \lceil \frac{d\_i - w\_i + 1}{s\_i} \rceil\).

#### Mathematical Definition[¶](#mathematical-definition)

Given an input data batch tensor \(T\_{in}\), the output tensor is defined by the equation

\[T\_{out}[a,c,i\_1,\dots,i\_n] = \max\_{j\_1 = s\_1 i\_1, \dots, j\_n = s\_n i\_n}^{j\_1 = s\_1 i\_1 + w\_1 - 1, \dots, j\_n = s\_n i\_n + w\_n - 1} (T\_{in}[a,c,j\_1,\dots,j\_n])\]

#### C++ Interface[¶](#c-interface)

*class* MaxPool : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v07MaxPoolE)

Batched max pooling operation, with optional padding and window stride.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v07MaxPool13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

MaxPool()[¶](#_CPPv4N6ngraph2op2v07MaxPool7MaxPoolEv)

Constructs a batched max pooling operation.

MaxPool(*const* Output<Node> &*arg*, *const* Shape &*window\_shape*, *const* Strides &*window\_movement\_strides*, *const* Shape &*padding\_below*, *const* Shape &*padding\_above*, *const* PadType &*pad\_type*, bool *ceil\_mode*)[¶](#_CPPv4N2op2v07MaxPool7MaxPoolERK6OutputI4NodeERK5ShapeRK7StridesRK5ShapeRK5ShapeRK7PadTypeb)

Constructs a batched max pooling operation.

**Parameters**

* arg: The node producing the input data batch tensor.
* window\_shape: The window shape.
* window\_movement\_strides: The window movement strides.
* padding\_below: The below-padding shape.
* padding\_above: The above-padding shape.
* pad\_type: The pad type for automatically computing padding sizes
* ceil\_mode: Whether to use ceiling while computing output shape.

MaxPool(*const* Output<Node> &*arg*, *const* Shape &*window\_shape*, *const* Strides &*window\_movement\_strides*, *const* Shape &*padding\_below*, *const* Shape &*padding\_above*, *const* PadType &*pad\_type*)[¶](#_CPPv4N2op2v07MaxPool7MaxPoolERK6OutputI4NodeERK5ShapeRK7StridesRK5ShapeRK5ShapeRK7PadType)

Constructs a batched max pooling operation.

**Parameters**

* arg: The node producing the input data batch tensor.
* window\_shape: The window shape.
* window\_movement\_strides: The window movement strides.
* padding\_below: The below-padding shape.
* padding\_above: The above-padding shape.
* pad\_type: The pad type for automatically computing padding sizes

MaxPool(*const* Output<Node> &*arg*, *const* Shape &*window\_shape*, *const* Strides &*window\_movement\_strides*, *const* Shape &*padding\_below*, *const* Shape &*padding\_above*)[¶](#_CPPv4N2op2v07MaxPool7MaxPoolERK6OutputI4NodeERK5ShapeRK7StridesRK5ShapeRK5Shape)

Constructs a batched max pooling operation.

**Parameters**

* arg: The node producing the input data batch tensor.
* window\_shape: The window shape.
* window\_movement\_strides: The window movement strides.
* padding\_below: The below-padding shape.
* padding\_above: The above-padding shape.

void validate\_and\_infer\_types()[¶](#_CPPv4N2op2v07MaxPool24validate_and_infer_typesEv)

Throws if the node is invalid.

MaxPool(*const* Output<Node> &*arg*, *const* Shape &*window\_shape*, *const* Strides &*window\_movement\_strides*)[¶](#_CPPv4N2op2v07MaxPool7MaxPoolERK6OutputI4NodeERK5ShapeRK7Strides)

Constructs a batched, unpadded max pooling operation (i.e., all padding shapes are set to 0).

**Parameters**

* arg: The node producing the input data batch tensor.
* window\_shape: The window shape.
* window\_movement\_strides: The window movement strides.

MaxPool(*const* Output<Node> &*arg*, *const* Shape &*window\_shape*)[¶](#_CPPv4N2op2v07MaxPool7MaxPoolERK6OutputI4NodeERK5Shape)

Constructs an unstrided batched max pooling operation (i.e., all window movement strides are 1 and all padding shapes are set to 0).

**Parameters**

* arg: The node producing the input data batch tensor.
* window\_shape: The window shape.

*const* Shape &get\_window\_shape() *const*[¶](#_CPPv4NK6ngraph2op2v07MaxPool16get_window_shapeEv)

**Return**

The window shape.

*const* Strides &get\_window\_movement\_strides() *const*[¶](#_CPPv4NK6ngraph2op2v07MaxPool27get_window_movement_stridesEv)

**Return**

The window movement strides.

*const* Shape &get\_padding\_below() *const*[¶](#_CPPv4NK6ngraph2op2v07MaxPool17get_padding_belowEv)

**Return**

The below-padding shape.

*const* Shape &get\_padding\_above() *const*[¶](#_CPPv4NK6ngraph2op2v07MaxPool17get_padding_aboveEv)

**Return**

The above-padding shape.

*const* PadType &get\_pad\_type() *const*[¶](#_CPPv4NK6ngraph2op2v07MaxPool12get_pad_typeEv)

**Return**

The pad type for pooling.

bool get\_ceil\_mode() *const*[¶](#_CPPv4NK6ngraph2op2v07MaxPool13get_ceil_modeEv)

**Return**

The ceiling mode being used for output shape computations

shared\_ptr<Node> get\_default\_value() *const*[¶](#_CPPv4NK2op2v07MaxPool17get_default_valueEv)

**Return**

The default value for [MaxPool](index.odt/index.html" \l "classngraph_1_1op_1_1v0_1_1MaxPool).

### Min[¶](#min)

Min // Min reduction

#### Description[¶](#description)

Reduces the tensor, eliminating the specified reduction axes by taking the minimum element.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | \((d\_1,\dots,d\_n)~(n \geq 0)\) |

##### Attributes[¶](#attributes)

| Name | Description |
| --- | --- |
| reduction\_axes | The axis positions (0-based) on which to calculate the max |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | \((d\_i:i\not\in \mathtt{reduction\_axes})\) |

#### C++ Interface[¶](#c-interface)

*class* Min : *public* ngraph::op::util::ArithmeticReduction[¶](#_CPPv4N6ngraph2op2v03MinE)

Min-reduction operation.

### Minimum[¶](#minimum)

Minimum // Short description.

#### Description[¶](#description)

Produces tensor of the same element type and shape as the two inputs, where the value at each coordinate of output is the minimum of the values at the corresponding input coordinates.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg0 | any | any |
| arg1 | same as arg0 | same as arg0 |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | same as arg0 | same as arg0 |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \min(\mathtt{arg0}\_{i\_0, \ldots, i\_{n-1}}, \mathtt{arg1}\_{i\_0, \ldots, i\_{n-1}})\]

#### Backprop[¶](#backprop)

\[\begin{split}\overline{\mathtt{arg0}} &\leftarrow \mathtt{Less}(\mathtt{arg0}, \mathtt{arg1})\ \Delta \\ \overline{\mathtt{arg1}} &\leftarrow \mathtt{Less}(\mathtt{arg1}, \mathtt{arg0})\ \Delta\end{split}\]

#### C++ Interface[¶](#c-interface)

*class* Minimum : *public* ngraph::op::util::BinaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v07MinimumE)

Elementwise minimum operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v07Minimum13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Minimum()[¶](#_CPPv4N6ngraph2op2v07Minimum7MinimumEv)

Constructs a minimum operation.

Minimum(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, *const* AutoBroadcastSpec &*auto\_broadcast* = AutoBroadcastSpec())[¶](#_CPPv4N2op2v07Minimum7MinimumERK6OutputI4NodeERK6OutputI4NodeERK17AutoBroadcastSpec)

Constructs a minimum operation.

**Parameters**

* arg0: Node that produces the first input tensor.
* arg1: Node that produces the second input tensor.
* auto\_broadcast: Auto broadcast specification

### Multiply[¶](#multiply)

Multiply // Elementwise multiply operation

#### Description[¶](#description)

Produces tensor of the same element type and shape as the two inputs, where the value at each coordinate of output is the product of the values at the corresponding input coordinates.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg0 | any | any |
| arg1 | same as arg0 | same as arg0 |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | same as arg0 | same as arg0 |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \mathtt{arg0}\_{i\_0, \ldots, i\_{n-1}} \mathtt{arg1}\_{i\_0, \ldots, i\_{n-1}}\]

#### Backprop[¶](#backprop)

\[\begin{split}\overline{\mathtt{arg0}} &\leftarrow \Delta\ \mathtt{arg1}\\ \overline{\mathtt{arg1}} &\leftarrow \Delta\ \mathtt{arg0}\end{split}\]

#### C++ Interface[¶](#c-interface)

*class* Multiply : *public* ngraph::op::util::BinaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v08MultiplyE)

Elementwise multiplication operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v08Multiply13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Multiply()[¶](#_CPPv4N6ngraph2op2v08Multiply8MultiplyEv)

Constructs a multiplication operation.

Multiply(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, *const* AutoBroadcastSpec &*auto\_broadcast* = AutoBroadcastSpec())[¶](#_CPPv4N2op2v08Multiply8MultiplyERK6OutputI4NodeERK6OutputI4NodeERK17AutoBroadcastSpec)

Constructs a multiplication operation.

**Parameters**

* arg0: Node that produces the first input tensor.
* arg1: Node that produces the second input tensor.
* auto\_broadcast: Auto broadcast specification

### Negative[¶](#negative)

Negative // Elementwise negative operation

#### Description[¶](#description)

Produces a single output tensor of the same element type and shape as arg, where the value at each coordinate of output is the negative of the value at each arg coordinate.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = -\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}}\]

#### Backprop[¶](#backprop)

\[\overline{\mathtt{arg}} \leftarrow -\Delta\]

#### C++ Interface[¶](#c-interface)

*class* Negative : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v08NegativeE)

Elementwise negative operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v08Negative13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Negative()[¶](#_CPPv4N6ngraph2op2v08Negative8NegativeEv)

Constructs a negative operation.

Negative(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v08Negative8NegativeERK6OutputI4NodeE)

Constructs a negative operation.

**Parameters**

* arg: Node that produces the input tensor.

### NotEqual[¶](#notequal)

NotEqual // Elementwise "not equal" operation

#### Description[¶](#description)

Produces tensor of the same element type and shape as the two inputs, where the value at each coordinate of output is 1 (true) if arg0 is not equal to arg1, 0 otherwise.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg0 | any | any |
| arg1 | same as arg0 | same as arg0 |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | ngraph::element::boolean | same as arg0 |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \mathtt{arg0}\_{i\_0, \ldots, i\_{n-1}} \neq \mathtt{arg1}\_{i\_0, \ldots, i\_{n-1}}\]

#### C++ Interface[¶](#c-interface)

*class* NotEqual : *public* ngraph::op::util::BinaryElementwiseComparison[¶](#_CPPv4N6ngraph2op2v08NotEqualE)

Elementwise not-equal operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v08NotEqual13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

NotEqual()[¶](#_CPPv4N6ngraph2op2v08NotEqual8NotEqualEv)

Constructs a not-equal operation.

NotEqual(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, *const* AutoBroadcastSpec &*auto\_broadcast* = AutoBroadcastSpec())[¶](#_CPPv4N2op2v08NotEqual8NotEqualERK6OutputI4NodeERK6OutputI4NodeERK17AutoBroadcastSpec)

Constructs a not-equal operation.

**Parameters**

* arg0: Node that produces the first input tensor.
* arg1: Node that produces the second input tensor.
* auto\_broadcast: Auto broadcast specification

### Not[¶](#not)

Not // Elementwise negation operation

#### Description[¶](#description)

Produces a single output tensor of boolean type and the same shape as arg, where the value at each coordinate of output is the negation of the value at each arg coordinate.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | element::boolean | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | element::boolean | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \neg\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}}\]

#### C++ Interface[¶](#c-interface)

*class* Not : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v03NotE)

Elementwise logical negation operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v03Not13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Not()[¶](#_CPPv4N6ngraph2op2v03Not3NotEv)

Constructs a logical negation operation.

Not(*const* Output<Node> &*arg*)[¶](#_CPPv4N2op2v03Not3NotERK6OutputI4NodeE)

Constructs a logical negation operation.

**Parameters**

* arg: Node that produces the input tensor.

void validate\_and\_infer\_types()[¶](#_CPPv4N2op2v03Not24validate_and_infer_typesEv)

Throws if the node is invalid.

### OneHot[¶](#onehot)

OneHot // One-hot expansion

#### Description[¶](#description)

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any integral type | \(d\_1,\dots,d\_{m-1},d\_{m+1},\dots,d\_n)~(n \geq 0)\) |

##### Attributes[¶](#attributes)

| Name | Description |
| --- | --- |
| shape | The desired output shape, including the new one-hot axis. |
| one\_hot\_axis | The index within the output shape of the new one-hot axis. |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | shape |

#### Mathematical Definition[¶](#mathematical-definition)

\[\begin{split}\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \begin{cases} 1&\text{if }i\_{\mathtt{one\\_hot\\_axis}} = \mathtt{arg}\_{(i : i\ne \mathtt{one\\_hot\\_axis})}\\ 0&\text{otherwise} \end{cases}\end{split}\]

#### C++ Interface[¶](#c-interface)

*class* OneHot : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v06OneHotE)

One-hot operator.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v06OneHot13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

OneHot()[¶](#_CPPv4N6ngraph2op2v06OneHot6OneHotEv)

Constructs a one-hot operation.

OneHot(*const* Output<Node> &*arg*, *const* PartialShape &*shape*, size\_t *one\_hot\_axis*)[¶](#_CPPv4N2op2v06OneHot6OneHotERK6OutputI4NodeERK12PartialShape6size_t)

Constructs a one-hot operation.

**Parameters**

* arg: Node that produces the input tensor to be one-hot encoded.
* shape: The shape of the output tensor, including the new one-hot axis.
* one\_hot\_axis: The index within the output shape of the new one-hot axis.

void validate\_and\_infer\_types()[¶](#_CPPv4N2op2v06OneHot24validate_and_infer_typesEv)

Throws if the node is invalid.

size\_t get\_one\_hot\_axis() *const*[¶](#_CPPv4NK6ngraph2op2v06OneHot16get_one_hot_axisEv)

**Return**

The index of the one-hot axis.

### Or[¶](#or)

Or // Elementwise logical-or operation

#### Description[¶](#description)

Produces tensor with boolean element type and shape as the two inputs, which must themselves have boolean element type, where the value at each coordinate of output is 1 (true) if arg0 or arg1 is nonzero, 0 otherwise.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg0 | ngraph::element::boolean | any |
| arg1 | ngraph::element::boolean | same as arg0 |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | ngraph::element::boolean | same as arg0 |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \mathtt{arg0}\_{i\_0, \ldots, i\_{n-1}}\, \mathtt{||}\, \mathtt{arg1}\_{i\_0, \ldots, i\_{n-1}}\]

#### C++ Interface[¶](#c-interface)

*class* Or : *public* ngraph::op::util::BinaryElementwiseLogical[¶](#_CPPv4N6ngraph2op2v02OrE)

Elementwise logical-or operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v02Or13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Or(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, *const* AutoBroadcastSpec &*auto\_broadcast* = AutoBroadcastSpec())[¶](#_CPPv4N2op2v02Or2OrERK6OutputI4NodeERK6OutputI4NodeERK17AutoBroadcastSpec)

Constructs a logical-or operation.

Output [d0, ...]

**Parameters**

* arg0: Node that produces the first input tensor.[d0, ...]
* arg1: Node that produces the second input tensor.[d0, ...]
* auto\_broadcast: Auto broadcast specification

### Pad[¶](#pad)

Pad // General padding operation

#### Description[¶](#description)

Adds edge padding.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | \((d\_1, \ldots, d\_n)\) |
| arg\_pad\_value | Same as arg | Scalar |

##### Attributes[¶](#attributes)

| Name | Description |
| --- | --- |
| padding\_below | Padding added before arg. May be negative. |
| padding\_above | Padding added after arg. May be negative. |
| pad\_mode | Padding mode: CONSTANT(default), EDGE or REFLECT. |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | \((d'\_1, \ldots, d'\_n)\) |

\[d'\_i = \mathtt{padding\\_below}\_i+d\_i\cdot(\mathtt{padding\\_interior}\_i)+\mathtt{padding\\_above}\_i\]

Takes an input tensor of shape \((d\_1,\dots,d\_n)\) and pads by inserting a scalar \(x\) supplied as input, in three possible ways:

1. *exterior padding* inserts copies of \(x\) *below or above* the bounds of existing rows, columns, etc.,
2. *interior padding* inserts copies of \(x\) *between* rows, columns, etc., or
3. both of the above.

The number and position of elements to be inserted along a given axis is determined by three attributes:

1. *the padding-below* CoordinateDiff \((p\_1,\ldots,p\_n)\),
2. *the padding-above* CoordinateDiff \((q\_1,\ldots,q\_n)\), and
3. *the interior padding* Shape \((r\_1,\ldots,r\_n)\).

The output tensor will have the shape \((d'\_1,\dots,d'\_n)\) where \(d'\_i = p\_i + (d\_i - 1)(r\_i + 1) + 1 + q\_i\) if \(d\_i > 0\), and \(d'\_i = p\_i + q\_i\) if \(d\_i = 0\).

Example: given a \(3\times 3\) tensor, with interior-padding sizes of \((1,2)\), padding-below of \((1,2)\), padding-above of \((1,0)\), and a pad-value of \(42\), we obtain:

42 42 42 42 42 42 42 42 42

42 42 1 42 42 2 42 42 3

1 2 3 42 42 42 42 42 42 42 42 42

4 5 6 --> 42 42 4 42 42 5 42 42 6

7 8 9 42 42 42 42 42 42 42 42 42

42 42 7 42 42 8 42 42 9

42 42 42 42 42 42 42 42 42

In other words we have inserted one new row between each pair of adjacent rows, two new columns between each pair of adjacent columns, one new row at the top and two new columns on the left, and one new row at the bottom and zero new columns on the right; then filled the new rows and columns with 42.

Note

The terms below and above here refer respectively to lower- or higher-numbered coordinate indices, and numbering starts at the upper-left corner; thus inserting a row “below” actually inserts it at the “top” of the matrix.

#### C++ Interface[¶](#c-interface)

*class* Pad : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v03PadE)

Generic padding operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v03Pad13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Pad()[¶](#_CPPv4N6ngraph2op2v03Pad3PadEv)

Constructs a generic padding operation.

Pad(*const* Output<Node> &*arg*, *const* Output<Node> &*arg\_pad\_value*, *const* CoordinateDiff &*padding\_below*, *const* CoordinateDiff &*padding\_above*, PadMode *pad\_mode* = PadMode::CONSTANT)[¶](#_CPPv4N2op2v03Pad3PadERK6OutputI4NodeERK6OutputI4NodeERK14CoordinateDiffRK14CoordinateDiff7PadMode)

Constructs a padding operation. Padding embeds the values of the input tensor into a larger tensor initialized to arg\_pad\_value.

**Parameters**

* arg: The node producing the input tensor to be padded.
* arg\_pad\_value: The node producing the scalar value to be used outside the are initialized by arg when pad\_mode is CONSTANT.
* padding\_below: How many elements to add on each axis before index 0 of arg. Rank must match arg.
* padding\_above: How many elements to add on each axis after the last element of arg. Rank must match arg.
* pad\_mode: The padding mode: CONSTANT(default), EDGE, REFLECT or SYMMETRIC. CONSTANT initializes new elements with arg\_pad\_value, EDGE uses the nearest value from arg. REFLECT and SYMMETRIC tile the background by flipping arg at the edge (SYMMETRIC) or on the last row/column/etc. (REFLECT).

void validate\_and\_infer\_types()[¶](#_CPPv4N2op2v03Pad24validate_and_infer_typesEv)

Throws if the node is invalid.

*const* CoordinateDiff &get\_padding\_below() *const*[¶](#_CPPv4NK6ngraph2op2v03Pad17get_padding_belowEv)

**Return**

The padding-below sizes.

*const* CoordinateDiff &get\_padding\_above() *const*[¶](#_CPPv4NK6ngraph2op2v03Pad17get_padding_aboveEv)

**Return**

The padding-above sizes.

*const* Shape &get\_padding\_interior() *const*[¶](#_CPPv4NK6ngraph2op2v03Pad20get_padding_interiorEv)

DEPRECATED. This is just a stub for backends that used to implement the interior padding feature, which is no longer supported.

**Return**

Returns a shape full of zeros, with the same rank as [get\_padding\_below()](index.odt/index.html" \l "classngraph_1_1op_1_1v0_1_1Pad_a9ff28a75b443211ac6e8c98c2270b4d3_1a9ff28a75b443211ac6e8c98c2270b4d3).

PadMode get\_pad\_mode() *const*[¶](#_CPPv4NK6ngraph2op2v03Pad12get_pad_modeEv)

**Return**

The padding mode.

*virtual* std::shared\_ptr<Node> get\_default\_value() *const*[¶](#_CPPv4NK6ngraph2op2v03Pad17get_default_valueEv)

**Return**

The default value for [Pad](index.odt/index.html" \l "classngraph_1_1op_1_1v0_1_1Pad).

### Parameter[¶](#parameter)

Parameter // A function parameter.

#### Description[¶](#description)

Parameters are nodes that represent the arguments that will be passed to user-defined functions. Function creation requires a sequence of parameters.

##### Attributes[¶](#attributes)

| Name | Description |
| --- | --- |
| element\_type | The element::Type of the parameter. |
| shape | The Shape of the parameter. |
| cacheable | True if the parameter is not expected to be frequently updated. |

##### Outputs[¶](#outputs)

| Name | Element type | Shape |
| --- | --- | --- |
| output | element\_type | shape |

A Parameter produces the value of the tensor passed to the function in the position of the parameter in the function’s arguments. The passed tensor must have the element type and shape specified by the parameter.

#### Backprop[¶](#backprop)

\[\leftarrow \Delta\]

#### C++ Interface[¶](#c-interface)

*class* Parameter : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v09ParameterE)

A function parameter.

Parameters are nodes that represent the arguments that will be passed to user-defined functions. Function creation requires a sequence of parameters. Basic graph operations do not need parameters attached to a function.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v09Parameter13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Parameter()[¶](#_CPPv4N6ngraph2op2v09Parameter9ParameterEv)

Constructions a tensor-typed parameter node.

Parameter(*const* ngraph::element::Type &*element\_type*, *const* [PartialShape](index.odt/index.html" \l "_CPPv4N6ngraph12PartialShapeE) &*pshape*, *const* bool *cacheable* = false)[¶](#_CPPv4N6ngraph2op2v09Parameter9ParameterERKN6ngraph7element4TypeERK12PartialShapeKb)

Constructions a tensor-typed parameter node.

**Parameters**

* element\_type: The element type of the parameter.
* pshape: The partial shape of the parameter.
* cacheable: True if the parameter is not expected to be frequently updated.

void validate\_and\_infer\_types()[¶](#_CPPv4N6ngraph2op2v09Parameter24validate_and_infer_typesEv)

Throws if the node is invalid.

### Power[¶](#power)

Power // Elementwise exponentiation operation

#### Description[¶](#description)

Elementwise exponentiation operation.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg0 | any | any |
| arg1 | same as arg0 | same as arg0 |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | same as arg0 | same as arg0 |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \mathtt{arg0}\_{i\_0, \ldots, i\_{n-1}} ^ {\mathtt{arg1}\_{i\_0, \ldots, i\_{n-1}}}\]

#### Backprop[¶](#backprop)

\[\begin{split}\overline{\mathtt{arg0}} &\leftarrow \frac{\Delta \cdot \mathtt{arg1}}{\mathtt{arg0}} \\ \overline{\mathtt{arg1}} &\leftarrow \Delta \cdot \mathtt{output} \cdot \log(\mathtt{arg1})\end{split}\]

#### C++ Interface[¶](#c-interface)

*class* Power : *public* ngraph::op::util::BinaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v05PowerE)

Elementwise exponentiation operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v05Power13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Power(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, *const* AutoBroadcastSpec &*auto\_broadcast* = AutoBroadcastSpec())[¶](#_CPPv4N2op2v05Power5PowerERK6OutputI4NodeERK6OutputI4NodeERK17AutoBroadcastSpec)

Constructs an exponentiation operation.

**Parameters**

* arg0: Node that produces the first input tensor.
* arg1: Node that produces the second input tensor.
* auto\_broadcast: Auto broadcast specification

### Product[¶](#product)

Product // Product reduction operation.

#### Description[¶](#description)

Reduces the tensor, eliminating the specified reduction axes by taking the product.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Attributes[¶](#attributes)

| Name | Description |
| --- | --- |
| reduction\_axes | The axis positions (0-based) on which to calculate the product |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg, with reduction\_axes removed. |

#### Mathematical Definition[¶](#mathematical-definition)

\[\begin{split}\mathit{product}\left(\{0\}, \left[ \begin{array}{ccc} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{array} \right]\right) &= \left[ (1 \* 3 \* 5), (2 \* 4 \* 6) \right] = \left[ 15, 48 \right]&\text{ dimension 0 (rows) is eliminated} \\ \mathit{product}\left(\{1\}, \left[ \begin{array}{ccc} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{array} \right]\right) &= \left[ (1 \* 2), (3 \* 4), (5 \* 6) \right] = \left[ 2, 12, 30 \right]&\text{ dimension 1 (columns) is eliminated}\\ \mathit{product}\left(\{0,1\}, \left[ \begin{array}{ccc} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{array} \right]\right) &= (1 \* 2) \* (3 \* 4) \* (5 \* 6) = 720&\text{ both dimensions (rows and columns) are eliminated}\end{split}\]

#### C++ Interface[¶](#c-interface)

*class* Product : *public* ngraph::op::util::ArithmeticReduction[¶](#_CPPv4N6ngraph2op2v07ProductE)

[Product](index.odt/index.html" \l "classngraph_1_1op_1_1v0_1_1Product) reduction operation.

Reduces the tensor, eliminating the specified reduction axes by taking the product.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v07Product13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Product()[¶](#_CPPv4N6ngraph2op2v07Product7ProductEv)

Constructs a product reduction operation.

Product(*const* Output<Node> &*arg*, *const* AxisSet &*reduction\_axes*)[¶](#_CPPv4N2op2v07Product7ProductERK6OutputI4NodeERK7AxisSet)

Constructs a product reduction operation.

**Parameters**

* arg: The tensor to be reduced.
* reduction\_axes: The axis positions (0-based) to be eliminated.

Product(*const* Output<Node> &*arg*, *const* Output<Node> &*reduction\_axes*)[¶](#_CPPv4N2op2v07Product7ProductERK6OutputI4NodeERK6OutputI4NodeE)

Constructs a product reduction operation.

**Parameters**

* arg: The tensor to be reduced.
* reduction\_axes: The axis positions (0-based) to be eliminated.

shared\_ptr<Node> get\_default\_value() *const*[¶](#_CPPv4NK2op2v07Product17get_default_valueEv)

**Return**

The default value for [Product](index.odt/index.html" \l "classngraph_1_1op_1_1v0_1_1Product).

### Quantize[¶](#quantize)

Quantize // Maps real input to quantized output

#### Description[¶](#description)

Produces a tensor of element type type and the same shape as input where the value of each coordinate \(i\) of output is the corresponding coordinate of input divided by scale rounded as specified by round\_mode plus zero\_point. The coordinate \(j\) of scale and zero\_point is the coordinate of output projected onto axes.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| input | Any real type | Any |
| scale | Same as input | input shape projected onto axes |
| zero\_point | Same as output | input shape projected onto axes |

##### Attributes[¶](#attributes)

| Name | Description |
| --- | --- |
| type | output element type; any quantized type |
| axes | Axis positions on which scale and zero\_point are specified |
| round\_mode | *ROUND\_NEAREST\_TOWARD\_INFINITY:* round to nearest integer in case of two equidistant integers round away from zero e.g. 2.5 -> 3 -3.5 -> -4  *ROUND\_NEAREST\_TOWARD\_ZERO:* round to nearest integer in case of two equidistant integers round toward zero e.g. 2.5 -> 2 -3.5 to -3  *ROUND\_NEAREST\_UPWARD:* round to nearest integer in case of two equidistant integers round up e.g. 2.5 to 3 -3.5 to -3  *ROUND\_NEAREST\_DOWNWARD:* round to nearest integer in case of two equidistant integers round down e.g. 2.5 to 2 -3.5 to -4  *ROUND\_NEAREST\_TOWARD\_EVEN:* round to nearest integer in case of two equidistant integers round to even e.g. 2.5 to 2 -3.5 to -4  *ROUND\_TOWARD\_INFINITY:* round to nearest integer away from zero  *ROUND\_TOWARD\_ZERO:* round to nearest integer toward zero  *ROUND\_UP:* round to nearest integer toward infinity (ceiling)  *ROUND\_DOWN:* round to nearest integer toward negative infinity (floor) |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | type | Same as input |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i,j} = \mathtt{round}\left(\frac{\mathtt{input}\_{i,j}}{\mathtt{scale}\_{j}}\right) + \mathtt{zero\_point}\_{j}\]

#### C++ Interface[¶](#c-interface)

*class* Quantize : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v08QuantizeE)

[Quantize](index.odt/index.html" \l "classngraph_1_1op_1_1v0_1_1Quantize) operation Maps real input (r) to quantized output (q) using scale (s), zero point (z) and round mode: q = ROUND(r / s) + o.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v08Quantize13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Quantize(*const* Output<Node> &*input*, *const* Output<Node> &*scale*, *const* Output<Node> &*zero\_point*, *const* ngraph::element::Type &*type*, *const* ngraph::AxisSet &*axes*, RoundMode *round\_mode*)[¶](#_CPPv4N6ngraph2op2v08Quantize8QuantizeERK6OutputI4NodeERK6OutputI4NodeERK6OutputI4NodeERKN6ngraph7element4TypeERKN6ngraph7AxisSetE9RoundMode)

Constructs a [Quantize](index.odt/index.html" \l "classngraph_1_1op_1_1v0_1_1Quantize) operation.

**Parameters**

* input: real input
* scale: scale used for mapping
* zero\_point: zero point used for mapping
* type: output element type
* axes: axis positions on which scale and zero\_point are specified
* round\_mode: describes how to perform ROUND function (see above)

void validate\_and\_infer\_types()[¶](#_CPPv4N6ngraph2op2v08Quantize24validate_and_infer_typesEv)

Throws if the node is invalid.

### RandomUniform[¶](#randomuniform)

RandomUniform // Operation that generates a tensor populated with random

// values of a uniform distribution.

#### Description[¶](#description)

Warning

This op is experimental and subject to change without notice.

##### Inputs[¶](#inputs)

Name

Element Type

Shape | Notes

min\_value

Any floating point type

Scalar | Minimum value for the random distribution

max\_value

Same as max\_value

Scalar

Maximum value for the random distribution

result\_shape

element::i64

Vector of any size

Shape of the output tensor

use\_fixed\_seed

element::boolean

Scalar

Flag indicating whether to use the fixed seed value fixed\_seed (useful for testing)

##### Attributes[¶](#attributes)

| Name | Type | Notes |
| --- | --- | --- |
| fixed\_seed | uint64\_t | Fixed seed value to use if use\_fixed\_seed flag is set to 1. This should be used only for testing; if use\_fixed\_seed is 1, RandomUniform will produce the \_same\_ values at each iteration. |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as min\_value | result\_shape |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_i = \mathtt{uniform\_rand}(\mathtt{min}=\mathtt{min\_value}, \mathtt{max}=\mathtt{max\_value})\]

#### C++ Interface[¶](#c-interface)

*class* RandomUniform : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op13RandomUniformE)

Generates a tensor populated with random values of a uniform distribution.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op13RandomUniform13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

RandomUniform()[¶](#_CPPv4N6ngraph2op13RandomUniform13RandomUniformEv)

Constructs an uninitialized [RandomUniform](index.odt/index.html" \l "classngraph_1_1op_1_1RandomUniform) node.

RandomUniform(*const* Output<Node> &*min\_value*, *const* Output<Node> &*max\_value*, *const* Output<Node> &*result\_shape*, *const* Output<Node> &*use\_fixed\_seed*, uint64\_t *fixed\_seed*)[¶](#_CPPv4N2op13RandomUniform13RandomUniformERK6OutputI4NodeERK6OutputI4NodeERK6OutputI4NodeERK6OutputI4NodeE8uint64_t)

Constructs a [RandomUniform](index.odt/index.html" \l "classngraph_1_1op_1_1RandomUniform) node.

**Parameters**

* min\_value: Output producing the minimum value (inclusive) for the random uniform distribution. Must return a scalar of floating point type, and the type must match that of max\_value.
* max\_value: Output producing the maximum value (inclusive) for the random uniform distribution. Must return a scalar of floating point type, and the type must match that of min\_value.
* result\_shape: Output producing the shape of the output tensor. Must return a vector of type element::i64.
* use\_fixed\_seed: Output producing a boolean scalar Flag indicating whether to use the value supplied in fixed\_seed to re-seed the random number generator at this iteration. Note that whenever use\_fixed\_seed is true, the same values will be generated in the output tensor. This flag is primarily used for debugging. If use\_fixed\_seed is false, the value in fixed\_seed is ignored.
* fixed\_seed: Fixed seed value to be supplied to the random number generator if use\_fixed\_seed is true. If use\_fixed\_seed is false, this value is ignored.

uint64\_t get\_fixed\_seed() *const*[¶](#_CPPv4NK6ngraph2op13RandomUniform14get_fixed_seedEv)

Returns the fixed seed value to be supplied to the random number generator if use\_fixed\_seed is true. If use\_fixed\_seed is false, this value is ignored.

void set\_fixed\_seed(uint64\_t *fixed\_seed*)[¶](#_CPPv4N6ngraph2op13RandomUniform14set_fixed_seedE8uint64_t)

Sets the fixed seed value to be supplied to the random number generator if use\_fixed\_seed is true. If use\_fixed\_seed is false, this value is ignored.

void validate\_and\_infer\_types()[¶](#_CPPv4N6ngraph2op13RandomUniform24validate_and_infer_typesEv)

Throws if the node is invalid.

### Relu[¶](#relu)

Relu // Elementwise relu operation

#### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

#### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

##### Mathematical Definition[¶](#mathematical-definition)

\[\begin{split}\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \begin{cases} 0&\text{if }\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}} \le 0 \\ \mathtt{arg}\_{i\_0, \ldots, i\_{n-1}}&\text{otherwise} \end{cases}\end{split}\]

##### C++ Interface[¶](#c-interface)

*class* Relu : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v04ReluE)

Elementwise [Relu](index.odt/index.html" \l "classngraph_1_1op_1_1v0_1_1Relu) operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v04Relu13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Relu(*const* Output<ngraph::Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v04Relu4ReluERK6OutputIN6ngraph4NodeEE)

Constructs a [Relu](index.odt/index.html" \l "classngraph_1_1op_1_1v0_1_1Relu) operation.

**Parameters**

* arg: Node that produces the input tensor.

### Result[¶](#result)

Result // Allow a value to be a result

#### Description[¶](#description)

Captures a value for use as a function result. The output of the op is the same as the input.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output} = \mathtt{arg}\]

#### C++ Interface[¶](#c-interface)

*class* Result : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v06ResultE)

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v06Result13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Result()[¶](#_CPPv4N6ngraph2op2v06Result6ResultEv)

Allows a value to be used as a function result.

Result(*const* Output<Node> &*arg*, bool *needs\_default\_layout* = false)[¶](#_CPPv4N6ngraph2op2v06Result6ResultERK6OutputI4NodeEb)

Allows a value to be used as a function result.

**Parameters**

* arg: Node that produces the input tensor.

void validate\_and\_infer\_types()[¶](#_CPPv4N6ngraph2op2v06Result24validate_and_infer_typesEv)

Throws if the node is invalid.

### ShapeOf[¶](#shapeof)

ShapeOf // Operation that returns the shape of its input tensor

#### Description[¶](#description)

Warning

This op is experimental and subject to change without notice.

Returns the shape of its input argument as a tensor of element type u64.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | element::u64 | {r} where r is the rank of arg’s shape. |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output} = \mathtt{shapeof}(\mathtt{arg})\]

#### C++ Interface[¶](#c-interface)

*class* ShapeOf : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v07ShapeOfE)

Operation that returns the shape of its input argument as a tensor.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v07ShapeOf13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

ShapeOf(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v07ShapeOf7ShapeOfERK6OutputI4NodeE)

Constructs a shape-of operation.

void validate\_and\_infer\_types()[¶](#_CPPv4N6ngraph2op2v07ShapeOf24validate_and_infer_typesEv)

Throws if the node is invalid.

### Sigmoid[¶](#sigmoid)

Sigmoid // Elementwise sigmoid operation

#### Description[¶](#description)

Produces a single output tensor of the same element type and shape as arg, where the value at each coordinate of output is the sigmoid of arg at the same coordinate.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \frac{1}{1+\exp(-\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}})}\]

#### C++ Interface[¶](#c-interface)

*class* Sigmoid : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v07SigmoidE)

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v07Sigmoid13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

### Sign[¶](#sign)

Sign // Elementwise sign operation

#### Description[¶](#description)

Produces a tensor of the same element type and shape as arg, where the value at each coordinate of output is the sign (-1, 0, 1) of the value at the corresponding coordinate of arg.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \mathtt{sgn}(\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}})\]

#### C++ Interface[¶](#c-interface)

*class* Sign : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v04SignE)

Elementwise sign operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v04Sign13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Sign(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v04Sign4SignERK6OutputI4NodeE)

Constructs an elementwise sign operation.

**Parameters**

* arg: Node that produces the input tensor.

### Sin[¶](#sin)

Sin // Elementwise sine operation

#### Description[¶](#description)

Produces a tensor of the same element type and shape as arg, where the value at each coordinate of output is the sine of the value at the corresponding coordinate of arg.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \sin(\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}})\]

#### Backprop[¶](#backprop)

\[\overline{\mathtt{arg}} \leftarrow \Delta\ \cos(\mathtt{arg})\]

#### C++ Interface[¶](#c-interface)

*class* Sin : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v03SinE)

Elementwise sine operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v03Sin13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Sin(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v03Sin3SinERK6OutputI4NodeE)

Constructs a sine operation.

**Parameters**

* arg: Node that produces the input tensor.

### Sinh[¶](#sinh)

Sinh // Elementwise hyperbolic sine operation

#### Description[¶](#description)

Produces a tensor of the same element type and shape as arg, where the value at each coordinate of output is the hyperbolic sine of the value at the corresponding coordinate of arg.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \sinh(\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}})\]

#### Backprop[¶](#backprop)

\[\overline{\mathtt{arg}} \leftarrow \Delta\ \cosh(\mathtt{arg})\]

#### C++ Interface[¶](#c-interface)

*class* Sinh : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v04SinhE)

Elementwise hyperbolic sine (sinh) operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v04Sinh13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Sinh(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v04Sinh4SinhERK6OutputI4NodeE)

Constructs a hyperbolic sine operation.

**Parameters**

* arg: Node that produces the input tensor.

### Slice[¶](#slice)

Slice // Produces a sub-tensor of its input.

#### Description[¶](#description)

Takes a slice of an input tensor, i.e., the sub-tensor that resides within a bounding box, optionally with a stride.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | \(D=D\_1, D\_2, \ldots, D\_n\). |

##### Attributes[¶](#attributes)

| Name | Description |
| --- | --- |
| lower\_bounds | The (inclusive) lower-bound coordinates \(L=L\_1, L\_2, \ldots, L\_n.\) |
| upper\_bounds | The (exclusive) upper-bound coordinates \(U=U\_1, U\_2, \ldots, U\_n.\) |
| strides | The strides \(S=S\_1, S\_2, \ldots, S\_n\) for the slices. Defaults to 1s. |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | \(D'\_i=\lceil\frac{U\_i-L\_i}{S\_i}\rceil\). |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_I = \mathtt{arg}\_{L+I\*S}\]

where \(I=I\_1, I\_2, \ldots, I\_n\) is a coordinate of the output.

#### C++ Interface[¶](#c-interface)

*class* Slice : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v05SliceE)

Takes a slice of an input tensor, i.e., the sub-tensor that resides within a bounding box, optionally with stride.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v05Slice13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Slice()[¶](#_CPPv4N6ngraph2op2v05Slice5SliceEv)

Constructs a tensor slice operation.

Slice(*const* Output<Node> &*arg*, *const* Coordinate &*lower\_bounds*, *const* Coordinate &*upper\_bounds*, *const* Strides &*strides*)[¶](#_CPPv4N6ngraph2op2v05Slice5SliceERK6OutputI4NodeERK10CoordinateRK10CoordinateRK7Strides)

Constructs a tensor slice operation.

**Parameters**

* arg: The tensor to be sliced.
* lower\_bounds: The axiswise lower bounds of the slice (inclusive).
* upper\_bounds: The axiswise upper bounds of the slice (exclusive).
* strides: The slicing strides; for example, strides of {n,m} means to take every nth row and every mth column of the input matrix.

Slice(*const* Output<Node> &*arg*, *const* Coordinate &*lower\_bounds*, *const* Coordinate &*upper\_bounds*)[¶](#_CPPv4N6ngraph2op2v05Slice5SliceERK6OutputI4NodeERK10CoordinateRK10Coordinate)

Constructs a tensor slice operation with unit strides; i.e., every element inside the bounding box will be copied to the output slice.

**Parameters**

* arg: The tensor to be sliced.
* lower\_bounds: The axiswise lower bounds of the slice (inclusive).
* upper\_bounds: The axiswise upper bounds of the slice (exclusive).

void validate\_and\_infer\_types()[¶](#_CPPv4N6ngraph2op2v05Slice24validate_and_infer_typesEv)

Throws if the node is invalid.

*const* Coordinate &get\_lower\_bounds() *const*[¶](#_CPPv4NK6ngraph2op2v05Slice16get_lower_boundsEv)

**Return**

The inclusive lower-bound coordinates.

*const* Coordinate &get\_upper\_bounds() *const*[¶](#_CPPv4NK6ngraph2op2v05Slice16get_upper_boundsEv)

**Return**

The exclusive upper-bound coordinates.

*const* Strides &get\_strides() *const*[¶](#_CPPv4NK6ngraph2op2v05Slice11get_stridesEv)

**Return**

The slicing strides.

### Softmax[¶](#softmax)

Softmax // Softmax operation

#### Description[¶](#description)

Produces a tensor of the same element type and shape as arg, where the value at each coordinate of output is the expine of the value of the corresponding coordinate of arg divided by the sum of the expine of all coordinates of arg in the specified axes.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Attributes[¶](#attributes)

| Name | Description |
| --- | --- |
| axes | The axis positions (0-based) on which to calculate the softmax |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i} = \frac{\exp(\mathtt{arg}\_{i})}{\sum\_{j} \exp(\mathtt{arg}\_{j})}\]

#### C++ Interface[¶](#c-interface)

*class* Softmax : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v07SoftmaxE)

[Softmax](index.odt/index.html" \l "classngraph_1_1op_1_1v0_1_1Softmax) operation.

### Sqrt[¶](#sqrt)

Sqrt // Elementwise square root operation

#### Description[¶](#description)

Produces a tensor of the same element type and shape as arg, where the value at each coordinate of output is the square root of the value at the corresponding coordinate of arg.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \sqrt{\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}}}\]

#### Backprop[¶](#backprop)

\[\overline{\mathtt{arg}} \leftarrow \frac{\Delta}{2\cdot \mathtt{output}}\]

#### C++ Interface[¶](#c-interface)

*class* Sqrt : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v04SqrtE)

Elementwise square root operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v04Sqrt13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Sqrt(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v04Sqrt4SqrtERK6OutputI4NodeE)

Constructs a square operation.

**Parameters**

* arg: Node that produces the input tensor.

### Subtract[¶](#subtract)

Subtract // Elementwise subtract operation

#### Description[¶](#description)

Elementwise subtract operation.

Produces tensor of the same element type and shape as the two inputs, where the value at each coordinate of output is the difference of the values at the corresponding input coordinates.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg0 | any | any |
| arg1 | same as arg0 | same as arg0 |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | same as arg0 | same as arg0 |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \mathtt{arg0}\_{i\_0, \ldots, i\_{n-1}} - \mathtt{arg1}\_{i\_0, \ldots, i\_{n-1}}\]

#### Backprop[¶](#backprop)

\[\begin{split}\overline{\mathtt{arg0}} &\leftarrow \Delta \\ \overline{\mathtt{arg1}} &\leftarrow -\Delta\end{split}\]

#### C++ Interface[¶](#c-interface)

*class* Subtract : *public* ngraph::op::util::BinaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v08SubtractE)

Elementwise subtraction operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v08Subtract13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Subtract(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, *const* AutoBroadcastSpec &*auto\_broadcast* = AutoBroadcastSpec())[¶](#_CPPv4N2op2v08Subtract8SubtractERK6OutputI4NodeERK6OutputI4NodeERK17AutoBroadcastSpec)

Constructs a subtraction operation.

**Parameters**

* arg0: Node that produces the first input tensor.
* arg1: Node that produces the second input tensor.
* auto\_broadcast: Auto broadcast specification

### Tan[¶](#tan)

Tan // Elementwise tangent operation

#### Description[¶](#description)

Produces a tensor of the same element type and shape as arg, where the value at each coordinate of output is the tangent of the value at the corresponding coordinate of arg.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \tan(\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}})\]

#### Backprop[¶](#backprop)

\[\overline{\mathtt{arg}} \leftarrow \frac{\Delta}{\cos^2(\mathtt{arg})}\]

#### C++ Interface[¶](#c-interface)

*class* Tan : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v03TanE)

Elementwise tangent operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v03Tan13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Tan(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v03Tan3TanERK6OutputI4NodeE)

Constructs a tangent operation.

**Parameters**

* arg: Node that produces the input tensor.

### Tanh[¶](#tanh)

Tanh // Elementwise hyperbolic tangent operation.

#### Description[¶](#description)

Produce a tensor with the same shape and element typye as arg, where the value at each coordinate of output is the hyperbolic tangent of the value of arg at the same coordinate.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | Same as arg |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \tanh(\mathtt{arg}\_{i\_0, \ldots, i\_{n-1}})\]

#### Backprop[¶](#backprop)

\[\overline{\mathtt{arg}} \leftarrow \Delta\ (1 - \mathtt{output}^2)\]

#### C++ Interface[¶](#c-interface)

*class* Tanh : *public* ngraph::op::util::UnaryElementwiseArithmetic[¶](#_CPPv4N6ngraph2op2v04TanhE)

Elementwise hyperbolic tangent operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v04Tanh13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Tanh(*const* Output<Node> &*arg*)[¶](#_CPPv4N6ngraph2op2v04Tanh4TanhERK6OutputI4NodeE)

Constructs a hyperbolic tangent operation.

**Parameters**

* arg: Node that produces the input tensor.

### Transpose[¶](#transpose)

Transpose // Operation that transposes axes of a tensor

#### Description[¶](#description)

Warning

This op is not yet implemented in any backend.

Warning

This op is experimental and subject to change without notice.

Operation that transposes axes of an input tensor. This operation covers matrix transposition, and also more general cases on higher-rank tensors.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg | Any | Any |
| input\_order | element::i64 | [n], where n is the rank of arg. |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | Same as arg | P(ShapeOf(arg)), where *P* is the permutation supplied for input\_order. |

The input input\_order must be a vector of shape [n], where n is the rank of arg, and must contain every integer in the range [0,n-1]. This vector represents a permutation of arg’s dimensions. For example,

| arg Shape | input\_order Value | output Shape | Comment |
| --- | --- | --- | --- |
| [3,4] | [1,0] | [4,3] | Transposes the arg matrix. |
| [3,3] | [1,0] | [3,3] | Transposes the arg matrix. |
| [3,3] | [1,0] | [3,3] | Transposes the arg matrix. |
| [3,4,8] | [2,0,1] | [8,3,4] | Moves the “last” dimension to the “first” position. |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0,i\_1,...,i\_n} = \mathtt{arg}\_{i\_{\mathtt{input\_order}[0]},i\_\mathtt{input\_order}[1],...,i\_\mathtt{input\_order}[n]}.\]

#### Backprop[¶](#backprop)

Not yet implemented.

#### C++ Interface[¶](#c-interface)

*class* Transpose : *public* ngraph::op::Op[¶](#_CPPv4N6ngraph2op2v19TransposeE)

Tensor transpose operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v19Transpose13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Transpose(*const* Output<Node> &*arg*, *const* Output<Node> &*input\_order*)[¶](#_CPPv4N2op2v19Transpose9TransposeERK6OutputI4NodeERK6OutputI4NodeE)

Constructs a transpose operation.

**Parameters**

* arg: Node producing the tensor to be transposed.
* input\_order: Node producing the permutation to apply to the axes of the input shape. Must be a vector of element type element::i64, with shape [n], where n is the rank of arg. The tensor’s value must contain every integer in the range [0,n-1].

void validate\_and\_infer\_types()[¶](#_CPPv4N2op2v19Transpose24validate_and_infer_typesEv)

Throws if the node is invalid.

### Xor[¶](#xor)

Xor // Elementwise logical-xor operation

#### Description[¶](#description)

Produces tensor with boolean element type and shape as the two inputs, which must themselves have boolean element type, where the value at each coordinate of output is 0 (true) if arg0 or arg1 both zero or both nonzero, or 1 otherwise.

##### Inputs[¶](#inputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| arg0 | ngraph::element::boolean | any |
| arg1 | ngraph::element::boolean | same as arg0 |

##### Outputs[¶](#outputs)

| Name | Element Type | Shape |
| --- | --- | --- |
| output | ngraph::element::boolean | same as arg0 |

#### Mathematical Definition[¶](#mathematical-definition)

\[\mathtt{output}\_{i\_0, \ldots, i\_{n-1}} = \mathtt{arg0}\_{i\_0, \ldots, i\_{n-1}}\, \mathtt{XOR}\, \mathtt{arg1}\_{i\_0, \ldots, i\_{n-1}}\]

#### C++ Interface[¶](#c-interface)

*class* Xor : *public* ngraph::op::util::BinaryElementwiseLogical[¶](#_CPPv4N6ngraph2op2v03XorE)

Elementwise logical-xor operation.

Public Functions

*const* NodeTypeInfo &get\_type\_info() *const*[¶](#_CPPv4NK6ngraph2op2v03Xor13get_type_infoEv)

Returns the NodeTypeInfo for the node’s class. During transition to type\_info, returns a dummy type\_info for Node if the class has not been updated yet.

Xor(*const* Output<Node> &*arg0*, *const* Output<Node> &*arg1*, *const* AutoBroadcastSpec &*auto\_broadcast* = AutoBroadcastSpec())[¶](#_CPPv4N2op2v03Xor3XorERK6OutputI4NodeERK6OutputI4NodeERK17AutoBroadcastSpec)

Constructs a logical-xor operation.

Output [d0, ...]

**Parameters**

* arg0: Node that produces the first input tensor.[d0, ...]
* arg1: Node that produces the second input tensor.[d0, ...]
* auto\_broadcast: Auto broadcast specification

### More about Core Ops[¶](#more-about-core-ops)

An Op’s primary role is to function as a node in a directed acyclic computation graph.

*Core ops* are ops that are available and generally useful to all framework bridges and that can be compiled by all transformers. A framework bridge may define framework-specific ops to simplify graph construction, provided that the bridge can enable every transformer to replace all such ops with equivalent clusters or subgraphs composed of core ops. In a similar manner, transformers may define transformer-specific ops to represent kernels or other intermediate operations.

The input and output ports of ops are any of the functions which work with Output<Node>/Input<Node>. Previous functions that worked at the level of ops are deprecated, like:

Node::get\_element\_type()

as it does not take any input. This function has been replaced with new functions like:

Node::get\_output\_element\_type(index)

where there is no ambiguity.

If a framework supports extending the set of ops it offers, a bridge may even expose transformer-specific ops to the framework user.

Important

Our design philosophy is that the graph is not a script for running kernels; rather, our compilation will match ops to appropriate kernels for the backend(s) in use. Thus, we expect that adding of new Core ops should be infrequent and that most functionality instead gets added with new functions that build sub-graphs from existing core ops.

It is easiest to define a new op by adapting an existing op. Some of the tasks that must be performed are:

* Op constructor:
  + Checking type-consistency of arguments
  + Specifying the result type for a call
* Serializer/Deserializer
* Transformer handlers:
  + Interpreter (reference) implementation of behavior. The implementation should favor clarity over efficiency.

## Provenance[¶](#provenance)

### Basic concepts[¶](#basic-concepts)

The term [provenance](index.odt/index.html" \l "term-provenance) refers to the matching of device code to framework subgraphs; it is analogous to source code locators in conventional compilers, which associate regions of object code with source files and line numbers. Provenance is *extensible* in that it may also include the chain of passes that lead from the framework graph to the executing code.

It can associate device code with specific tags added by a framework bridge which correspond to the framework ops that create the nGraph nodes. This works only for those transformations that take place in nGraph: the information stored in the nodes can include additional details about how the device code was chosen. For example, whenever a graph transformation is performed with one of the nGraph core [Ops](index.odt/index.html" \l "document-ops/index), a lower level of abstraction can record information about the transformation that may be useful to anyone wondering why a kernel was “chosen”; a complete description of the steps leading to the device kernels being used, as well as all of the framework nodes that led to the kernel, can be obtained.

#### Existing use cases[¶](#existing-use-cases)

Currently, every node nGraph touches can optionally have a set of provenance tags, which are strings set by a framework bridge. When a set of nodes is replaced by a new set of nodes, a combination of heuristics and special casing is used to set the tags on the new nodes based on the tags from the old nodes.

A [builder](index.odt/index.html" \l "term-builder) is a function that creates a sub-graph and returns a root node to the bridge. The bridge is not necessarily aware of the subgraph, only of the returned node, where it sets tags. The remaining nodes’ tags are set by associating a set of nodes, called a *provenance group*, with the node. Any tags added to the node are also added to the nodes in the provenance group.

An updated implementation of the functionality of builders is the *fused op*, a node that can replace itself with a subgraph. When the node is expanded into a subgraph, a vector of values is returned, corresponding to outputs of the original fused op; the tags of the fused op are added to all nodes in the values in reverse dataflow direction, up to (though not including) the input values of the fused op.

## Quantization[¶](#quantization)

[Quantization](index.odt/index.html" \l "term-quantization) refers the process of reducing the number of bits that represent a number. In a DL context, weights and activations can be represented using 8-bit integers (INT8) to compress the model size of a trained neural network without any significant loss in model accuracy. INT8 is one kind of quantization. Compared with 32-bit floating point (FP32), using arithmetic with lower precision, such as INT8, to calculate weights and activation requires less memory.

### Implementing a quantized model with nGraph[¶](#implementing-a-quantized-model-with-ngraph)

To implement a quantized model with nGraph, provide a partially (or fully) quantized model (where the convolution layer in the model is replaced with a quantized convolution, for example) to the nGraph Library along with quantized parameters: weights, activations, scale, and zero point.

Note

As of version 0.29, only quantization for inference is supported.

### nGraph Quantized Operators (Ops)[¶](#ngraph-quantized-operators-ops)

nGraph uses scale and zero point (also used by ONNX) to map real values to quantized values. All quantized ops use scale and zero point and can be used just like any other nGraph op.

**Scale**: the quantization scale of the tensor

**Zero point**: the zero point of the tensor

**Round mode**: used in combination with scale and zero point to round real values to quantized values

| Op | Description |
| --- | --- |
| [Quantize](index.odt/index.html" \l "document-ops/quantize) | Maps real values (r) to quantized values (q) using scale (s), zero point (z), and round mode; produces a quantized tensor. |
| [Dequantize](index.odt/index.html" \l "document-ops/dequantize) | Maps quantized values (q) to real values (r) using scale (s) and zero point (z); converts a quantized tensor to a floating-point tensor. |
| [FakeQuantize](index.odt/index.html" \l "ngraph.ops.fake_quantize) | Performs element-wise linear quantization. |
| [QuantizedConvolution](index.odt/index.html" \l "ngraph.ops.quantized_convolution) | Performs 8-bit convolution. |
| [QuantizedDot](index.odt/index.html" \l "ngraph.ops.quantized_dot) | Performs 8-bit dot. |

Quantization Ops[¶](#id2)

Some frameworks such as TensorFlow\* have fused ops. nGraph provides optional operations to help users easily translate (map) any quantized model created from frameworks with fused ops to nGraph. Unlike builders, experimental ops take scale and zero point instead of min and max.

| Operator | Description |
| --- | --- |
| QuantizedConvolutionBias | This experimental op can be fused with a ReLU op. |
| QuantizedConvolutionBiasAdd | This experimental op constructs a quantized convolution with bias and optional ReLU. And then takes input for the add operation. |
| QuantizedConvolutionBiasSignedAdd | Same as QuantizedConvolutionBiasAdd but with signed add. |
| QuantizedConvolutionRelu | This experimental op is designed for a particular use case that would require convolution and ReLU to be combined. |
| QuantizedDotBias | This experimental op can be fused with a ReLU op. |

Experimental Quantized Ops (optional)[¶](#id3)

### nGraph Quantization Design[¶](#ngraph-quantization-design)

The goal of nGraph quantization is to flexibly support a wide variety of frameworks and users. The use of scale and zero point as well as quantized builders in the nGraph design helps to achieve this goal.

#### Scale and Zero Point[¶](#scale-and-zero-point)

Using scale and zero point allows nGraph to be framework agnostic (i.e., it can equally support all deep learning frameworks). nGraph Bridges will automatically convert min and max (provided by a DL framework) to scale and zero point as needed. Quantized builders are available to help the bridges perform this calculation. However, if users are directly using nGraph (and not using a bridge), they are required to provide scale and zero point for quantized ops.

Another advantage of using scale and zero point to express quantization parameters is that users can flexibly implement quantized ops into various nGraph backends. When implementing quantized ops, all current nGraph backends will directly use scale and zero point (and not min and max) to perform the quantized computation.

#### Quantized Builders[¶](#quantized-builders)

Quantized builders are helper utilities to assist framework integrators to enable quantized models with nGraph. They serve as an API (interface) between framework bridges and nGraph, allowing framework bridges to directly construct ops in the nGraph Abstraction Layer.

Quantized builders help nGraph framework bridges by:

* Breaking down a fused quantized operator in the framework to a subgraph (of quantized and non-quantized operators) in the nGraph core IR
* Converting from min and max to scale and zero point based on the quantization mode described by the DL framework

**Note:** Fused ops and quantized builders serve the same purpose. In the future, fused ops will replace quantized builders.

nGraph Quantized Builders[¶](#id4)

Category

Builder

Description

Scaled Mode Min / Max Builders

ScaledQuantize

Converts min and max to scale and zero point using a scaled mode calculation and then constructs and returns an nGraph Quantize operator.

ScaledDequantize

Converts min and max to scale and zero point using a scaled mode calculation and then constructs and returns an nGraph Dequantize operator.

Quantized Convolution and Variants

ScaledQuantizedConvolution

Constructs a quantized convolution with an optional ReLU.

ScaledQuantizedConvolutionBias

Constructs a quantized convolution with bias and an optional ReLU.

ScaledQuantizedConvolutionBiasAdd

Constructs a quantized convolution with bias and an optional ReLU, where the output is added to the output of another convolution (sum\_input).

Quantized Dot (Matmul) and Variants

ScaledQuantizedDot

Constructs a quantized dot (Matmul) with an optional ReLU.

ScaledQuantizedDotBias

Constructs a quantized dot (Matmul) with bias and an optional ReLU.

Quantized Concat

ScaledQuantizedConcat

Constructs a quantized concatenation.

## Dynamic Shapes[¶](#dynamic-shapes)

For an example on how to use dynamic shapes, see the [Scenario Two: Known Partial Shape](index.odt/index.html" \l "scenario-two) documentation.

### Runtime Error Checking[¶](#runtime-error-checking)

Static type-checking in the presence of dynamic shapes will make optimistic assumptions about things like shape mismatches. For example, if an elementwise op is provided inputs of shapes (2,?) and (?,5), the type checker will proceed under the assumption that the user is not going to pass tensors with inconsistent shape at runtime, and therefore infer an output shape of (2,5). That means that shape mismatches can now occur at runtime.

### PartialShape, Dimension, and Rank Classes[¶](#partialshape-dimension-and-rank-classes)

Partial shape information is expressed via the PartialShape, Dimension, and Rank classes.

Note

Rank is an alias for Dimension, used when the value represents the number of axes in a shape, rather than the size of one dimension in a shape.

*class* PartialShape[¶](#_CPPv4N6ngraph12PartialShapeE)

Class representing a shape that may be partially or totally dynamic.

A [PartialShape](index.odt/index.html" \l "classngraph_1_1PartialShape) may have:

* Dynamic rank. (Informal notation: ?)
* Static rank, but dynamic dimensions on some or all axes. (Informal notation examples: {1,2,?,4}, {?,?,?})
* Static rank, and static dimensions on all axes. (Informal notation examples: {1,2,3,4}, {6}, {})

Public Functions

PartialShape(std::initializer\_list<[Dimension](index.odt/index.html" \l "_CPPv4N6ngraph9DimensionE)> *init*)[¶](#_CPPv4N6ngraph12PartialShape12PartialShapeENSt16initializer_listI9DimensionEE)

Constructs a shape with static rank from an initializer list of [Dimension](index.odt/index.html" \l "classngraph_1_1Dimension).

Examples:

**Parameters**

* init: The [Dimension](index.odt/index.html" \l "classngraph_1_1Dimension) values for the constructed shape.

PartialShape s{2,3,4}; // rank=3, all dimensions static

PartialShape s{}; // rank=0

PartialShape s{2,Dimension::dynamic(),3}; // rank=3, dimension 1 dynamic

PartialShape(*const* std::vector<[Dimension](index.odt/index.html" \l "_CPPv4N6ngraph9DimensionE)> &*dimensions*)[¶](#_CPPv4N6ngraph12PartialShape12PartialShapeERKNSt6vectorI9DimensionEE)

Constructs a [PartialShape](index.odt/index.html" \l "classngraph_1_1PartialShape) with static rank from a vector of [Dimension](index.odt/index.html" \l "classngraph_1_1Dimension).

**Parameters**

* dimensions: The [Dimension](index.odt/index.html" \l "classngraph_1_1Dimension) values for the constructed shape.

PartialShape()[¶](#_CPPv4N6ngraph12PartialShape12PartialShapeEv)

Constructs a static [PartialShape](index.odt/index.html" \l "classngraph_1_1PartialShape) with zero rank (the shape of a scalar).

PartialShape(*const* Shape &*shape*)[¶](#_CPPv4N12PartialShape12PartialShapeERK5Shape)

Constructs a static [PartialShape](index.odt/index.html" \l "classngraph_1_1PartialShape) from a Shape.

**Parameters**

* shape: The Shape to convert into [PartialShape](index.odt/index.html" \l "classngraph_1_1PartialShape).

bool is\_static() *const*[¶](#_CPPv4NK6ngraph12PartialShape9is_staticEv)

Check if this shape is static.

A shape is considered static if it has static rank, and all dimensions of the shape are static.

**Return**

true if this shape is static, else false.

bool is\_dynamic() *const*[¶](#_CPPv4NK6ngraph12PartialShape10is_dynamicEv)

Check if this shape is dynamic.

A shape is considered static if it has static rank, and all dimensions of the shape are static.

**Return**

false if this shape is static, else true.

Rank rank() *const*[¶](#_CPPv4NK6ngraph12PartialShape4rankEv)

Get the rank of the shape.

**Return**

The rank of the shape. This will be Rank::dynamic() if the rank of the shape is dynamic.

bool compatible(*const* PartialShape &*s*) *const*[¶](#_CPPv4NK12PartialShape10compatibleERK12PartialShape)

Check whether this shape is compatible with the argument, i.e., whether it is possible to merge them.

Two shapes are compatible if

* one or both of them has dynamic rank, or
* both shapes have dynamic and equal rank, and their dimensions are elementwise compatible (see [Dimension::compatible()](index.odt/index.html" \l "classngraph_1_1Dimension_a5db79a6648dfe51d196170b299c51e06_1a5db79a6648dfe51d196170b299c51e06)).

**Return**

true if this shape is compatible with s, else false.

**Parameters**

* s: The shape to be checked for compatibility with this shape.

bool same\_scheme(*const* PartialShape &*s*) *const*[¶](#_CPPv4NK12PartialShape11same_schemeERK12PartialShape)

Check whether this shape represents the same scheme as the argument.

Two shapes

s1 and s2 represent the same scheme if

* they both have dynamic rank, or
* they both have static and equal rank r, and for every i from 0 to r-1, s1[i] represents the same scheme as s2[i] (see [Dimension::same\_scheme()](index.odt/index.html" \l "classngraph_1_1Dimension_aa98e5e906ee63dde88133798e7760949_1aa98e5e906ee63dde88133798e7760949)).

**Return**

true if this shape represents the same scheme as s, else false.

**Parameters**

* s: The shape whose scheme is being compared with this shape.

bool relaxes(*const* PartialShape &*s*) *const*[¶](#_CPPv4NK12PartialShape7relaxesERK12PartialShape)

Check whether this shape is a relaxation of the argument.

Intuitively, a

[PartialShape](index.odt/index.html" \l "classngraph_1_1PartialShape) s1 is said to *relax* s2 (or *is a relaxation* of s2) if it is “more permissive” than s2. In other words, s1 is a relaxation of s2 if anything you can form by plugging things into the dynamic dimensions of s2 is also something you can form by plugging things into the dynamic dimensions of s1, but not necessarily the other way around.

**Return**

true if this shape relaxes s, else false.

**Parameters**

* s: The shape which is being compared against this shape.

s1.relaxes(s2) is equivalent to s2.refines(s1).

Formally, [PartialShape](index.odt/index.html" \l "classngraph_1_1PartialShape) s1 is said to *relax* [PartialShape](index.odt/index.html" \l "classngraph_1_1PartialShape) s2 if:

* s1 has dynamic rank, or
* s1 and s2 both have static rank r, and for every i from 0 to r-1, either s1[i] is dynamic, or s1[i] == s2[i].

bool refines(*const* PartialShape &*s*) *const*[¶](#_CPPv4NK12PartialShape7refinesERK12PartialShape)

Check whether this shape is a refinement of the argument.

Intuitively, a

[PartialShape](index.odt/index.html" \l "classngraph_1_1PartialShape) s1 is said to *relax* s2 (or *is a relaxation* of s2) if it is “less permissive” than s2. In other words, s1 is a relaxation of s2 if anything you can form by plugging things into the dynamic dimensions of s1 is also something you can form by plugging things into the dynamic dimensions of s2, but not necessarily the other way around.

**Return**

true if this shape refines s, else false.

**Parameters**

* s: The shape which is being compared against this shape.

s1.refines(s2) is equivalent to s2.relaxes(s1).

Formally, [PartialShape](index.odt/index.html" \l "classngraph_1_1PartialShape) s1 is said to *refine* [PartialShape](index.odt/index.html" \l "classngraph_1_1PartialShape) s2 if:

* s2 has dynamic rank, or
* s1 and s2 both have static rank r, and for every i from 0 to r-1, either s2[i] is dynamic, or s1[i] == s2[i].

bool merge\_rank(Rank *r*)[¶](#_CPPv4N12PartialShape10merge_rankE4Rank)

Checks that this shape’s rank is compatible with r, and, if this shape’s rank is dynamic and r is static, updates this shape to have a rank of r with dimensions all dynamic.

**Return**

true if this shape’s rank is compatible with r, else false.

Shape to\_shape() *const*[¶](#_CPPv4NK12PartialShape8to_shapeEv)

Convert a static [PartialShape](index.odt/index.html" \l "classngraph_1_1PartialShape) to a Shape.

**Return**

A new Shape s where s[i] = size\_t((\*this)[i]).

**Exceptions**

* std::invalid\_argument: If this [PartialShape](index.odt/index.html" \l "classngraph_1_1PartialShape) is dynamic.

bool all\_non\_negative() *const*[¶](#_CPPv4NK12PartialShape16all_non_negativeEv)

Returns true if all static dimensions of the tensor are non-negative, else false.

*const* [Dimension](index.odt/index.html" \l "_CPPv4N6ngraph9DimensionE) &operator[](size\_t *i*) *const*[¶](#_CPPv4NK6ngraph12PartialShapeixE6size_t)

Index operator for [PartialShape](index.odt/index.html" \l "classngraph_1_1PartialShape).

**Return**

A reference to the ith [Dimension](index.odt/index.html" \l "classngraph_1_1Dimension) of this shape.

**Parameters**

* i: The index of the dimension being selected.

[Dimension](index.odt/index.html" \l "_CPPv4N6ngraph9DimensionE) &operator[](size\_t *i*)[¶](#_CPPv4N6ngraph12PartialShapeixE6size_t)

Index operator for [PartialShape](index.odt/index.html" \l "classngraph_1_1PartialShape).

**Return**

A reference to the ith [Dimension](index.odt/index.html" \l "classngraph_1_1Dimension) of this shape.

**Parameters**

* i: The index of the dimension being selected.

operator std::vector<Dimension>() *const*[¶](#_CPPv4NK6ngraph12PartialShapecvNSt6vectorI9DimensionEEEv)

Returns a vector of the dimensions. This has no meaning if dynamic.

Public Static Functions

PartialShape dynamic(Rank *r* = Rank::dynamic())[¶](#_CPPv4N12PartialShape7dynamicE4Rank)

Construct a [PartialShape](index.odt/index.html" \l "classngraph_1_1PartialShape) with the given rank and all dimensions (if any) dynamic.

**Return**

A [PartialShape](index.odt/index.html" \l "classngraph_1_1PartialShape) with the given rank, and all dimensions (if any) dynamic.

bool merge\_into(PartialShape &*dst*, *const* PartialShape &*src*)[¶](#_CPPv4N12PartialShape10merge_intoER12PartialShapeRK12PartialShape)

Try to merge one shape into another.

Merges

src into dst, returning true on success and false on failure. If false is returned, the effect on dst is unspecified.

**Return**

true if merging succeeds, else false.

**Parameters**

* dst: The shape that src will be merged into.
* src: The shape that will be merged into dst.

To merge two partial shapes s1 and s2 is to find the most permissive partial shape s that is no more permissive than s1 or s2, if s exists. For example:

merge(?,?) -> ?

merge(?,{?,?}) -> {?,?}

merge({?,?},{?,?}) -> {?,?}

merge({1,2,3,4},?) -> {1,2,3,4}

merge({1,2},{1,?}) -> {1,2}

merge({1,2,?,?},{1,?,3,?}) -> {1,2,3,?}

merge({1,2,3},{1,2,3}) -> {1,2,3}

merge({1,?},{2,?}) fails [dimension 0 constraints are inconsistent]

merge({?,?},{?,?,?}) fails [ranks are inconsistent]

This function (merge\_into) performs the “merge” operation described above on dst and src, but overwrites dst with the result and returns true if merging is successful; if merging is unsuccessful, the function returns false and may make unspecified changes to dst.

bool broadcast\_merge\_into(PartialShape &*dst*, *const* PartialShape &*src*, *const* op::AutoBroadcastSpec &*autob*)[¶](#_CPPv4N12PartialShape20broadcast_merge_intoER12PartialShapeRK12PartialShapeRKN2op17AutoBroadcastSpecE)

Try to merge one shape into another along with implicit broadcasting.

*class* Dimension[¶](#_CPPv4N6ngraph9DimensionE)

Class representing a dimension, which may be dynamic (undetermined until runtime), in a shape or shape-like object.

Static dimensions may be implicitly converted from int64\_t. A dynamic dimension is constructed with [Dimension()](index.odt/index.html" \l "classngraph_1_1Dimension_1a5142d8915846b85e9eda226441ff2a3d) or [Dimension::dynamic()](index.odt/index.html" \l "classngraph_1_1Dimension_a456844bbf9d1aee5709cf9dbd7675e9d_1a456844bbf9d1aee5709cf9dbd7675e9d).

XXX: THIS CLASS IS NOT IN USE YET AND THE ENTIRE DESIGN IS SUBJECT TO CHANGE.

Public Functions

Dimension(int64\_t *dimension*)[¶](#_CPPv4N9Dimension9DimensionE7int64_t)

Construct a static dimension.

**Parameters**

* dimension: Value of the dimension. Must not be equal to [Dimension::s\_dynamic\_val](index.odt/index.html" \l "classngraph_1_1Dimension_1a6c786d89b58b6561c9b4e02e0fd99e4d).

**Exceptions**

* std::invalid\_argument: If dimension == [Dimension::s\_dynamic\_val](index.odt/index.html" \l "classngraph_1_1Dimension_1a6c786d89b58b6561c9b4e02e0fd99e4d).

Dimension()[¶](#_CPPv4N6ngraph9Dimension9DimensionEv)

Construct a dynamic dimension.

bool is\_static() *const*[¶](#_CPPv4NK6ngraph9Dimension9is_staticEv)

Check whether this dimension is static.

**Return**

true if the dimension is static, else false.

bool is\_dynamic() *const*[¶](#_CPPv4NK6ngraph9Dimension10is_dynamicEv)

Check whether this dimension is dynamic.

**Return**

false if the dimension is static, else true.

operator int64\_t() *const*[¶](#_CPPv4NK6ngraph9Dimensioncv7int64_tEv)

Convert this dimension to int64\_t. This dimension must be static.

**Exceptions**

* std::invalid\_argument: If this dimension is dynamic.

operator size\_t() *const*[¶](#_CPPv4NK6ngraph9Dimensioncv6size_tEv)

Convert this dimension to size\_t. This dimension must be static and non-negative.

**Exceptions**

* std::invalid\_argument: If this dimension is dynamic or negative.

bool same\_scheme(*const* [Dimension](index.odt/index.html" \l "_CPPv4N6ngraph9DimensionE) &*dim*) *const*[¶](#_CPPv4NK6ngraph9Dimension11same_schemeERK9Dimension)

Check whether this dimension represents the same scheme as the argument (both dynamic, or equal).

**Return**

true if this dimension and dim are both dynamic, or if they are both static and equal; otherwise, false.

**Parameters**

* dim: The other dimension to compare this dimension to.

bool compatible(*const* Dimension &*d*) *const*[¶](#_CPPv4NK9Dimension10compatibleERK9Dimension)

Check whether this dimension is capable of being merged with the argument dimension.

Two dimensions are considered compatible if it is possible to merge them. (See

[Dimension::merge](index.odt/index.html" \l "classngraph_1_1Dimension_a55f16d5168f62489ecffac7dd9c21f91_1a55f16d5168f62489ecffac7dd9c21f91).)

**Return**

true if this dimension is compatible with d, else false.

**Parameters**

* d: The dimension to compare this dimension with.

bool relaxes(*const* Dimension &*d*) *const*[¶](#_CPPv4NK9Dimension7relaxesERK9Dimension)

Check whether this dimension is a relaxation of the argument.

A dimension

d1 *relaxes* (or *is a relaxation of*) d2 if d1 and d2 are static and equal, or d1 is dynamic.

**Return**

true if this dimension relaxes d, else false.

**Parameters**

* d: The dimension to compare this dimension with.

d1.relaxes(d2) is equivalent to d2.refines(d1).

bool refines(*const* Dimension &*d*) *const*[¶](#_CPPv4NK9Dimension7refinesERK9Dimension)

Check whether this dimension is a refinement of the argument.

A dimension

d2 *refines* (or *is a refinement of*) d1 if d1 and d2 are static and equal, or d2 is dynamic.

**Return**

true if this dimension relaxes d, else false.

**Parameters**

* d: The dimension to compare this dimension with.

d1.refines(d2) is equivalent to d2.relaxes(d1).

Dimension operator+(*const* Dimension &*dim*) *const*[¶](#_CPPv4NK9DimensionplERK9Dimension)

Addition operator for [Dimension](index.odt/index.html" \l "classngraph_1_1Dimension).

**Return**

[Dimension::dynamic()](index.odt/index.html" \l "classngraph_1_1Dimension_a456844bbf9d1aee5709cf9dbd7675e9d_1a456844bbf9d1aee5709cf9dbd7675e9d) if either of \*this or dim is dynamic; else, a static dimension with value int64\_t(\*this)+in64\_t(dim).

**Parameters**

* dim: Right operand for addition.

Dimension operator-(*const* Dimension &*dim*) *const*[¶](#_CPPv4NK9DimensionmiERK9Dimension)

Subtraction operator for [Dimension](index.odt/index.html" \l "classngraph_1_1Dimension).

**Return**

[Dimension::dynamic()](index.odt/index.html" \l "classngraph_1_1Dimension_a456844bbf9d1aee5709cf9dbd7675e9d_1a456844bbf9d1aee5709cf9dbd7675e9d) if either of \*this or dim is dynamic; else, a static dimension with value int64\_t(\*this)-int64\_t(dim).

**Parameters**

* dim: Right operand for subtraction.

Dimension operator\*(*const* Dimension &*dim*) *const*[¶](#_CPPv4NK9DimensionmlERK9Dimension)

Multiplication operator for [Dimension](index.odt/index.html" \l "classngraph_1_1Dimension).

**Return**

0 if either of \*this or dim is static and 0; else, [Dimension::dynamic()](index.odt/index.html" \l "classngraph_1_1Dimension_a456844bbf9d1aee5709cf9dbd7675e9d_1a456844bbf9d1aee5709cf9dbd7675e9d) if either of \*this or dim is dynamic; else, a static dimension with value int64\_t(\*this)\*int64\_t(dim).

**Parameters**

* dim: Right operand for multiplicaiton.

[Dimension](index.odt/index.html" \l "_CPPv4N6ngraph9DimensionE) &operator+=(*const* [Dimension](index.odt/index.html" \l "_CPPv4N6ngraph9DimensionE) &*dim*)[¶](#_CPPv4N6ngraph9DimensionpLERK9Dimension)

Add-into operator for [Dimension](index.odt/index.html" \l "classngraph_1_1Dimension).

**Return**

A reference to \*this, after updating \*this to the value \*this + dim.

**Parameters**

* dim: Right operand for addition.

[Dimension](index.odt/index.html" \l "_CPPv4N6ngraph9DimensionE) &operator\*=(*const* [Dimension](index.odt/index.html" \l "_CPPv4N6ngraph9DimensionE) &*dim*)[¶](#_CPPv4N6ngraph9DimensionmLERK9Dimension)

Multiply-into operator for [Dimension](index.odt/index.html" \l "classngraph_1_1Dimension).

**Return**

A reference to \*this, after updating \*this to the value \*this \* dim.

**Parameters**

* dim: Right operand for multiplication.

Public Static Functions

bool merge(Dimension &*dst*, *const* Dimension *d1*, *const* Dimension *d2*)[¶](#_CPPv4N9Dimension5mergeER9DimensionK9DimensionK9Dimension)

Try to merge two [Dimension](index.odt/index.html" \l "classngraph_1_1Dimension) objects together.

* If d1 is dynamic, writes d2 to dst and returns true.
* If d2 is dynamic, writes d1 to dst and returns true.
* If d1 and d2 are static and equal, writes d1 to dst and returns true.
* If d1 and d2 are both static and unequal, leaves dst unchanged and returns false.

**Return**

true if merging succeeds, else false.

**Parameters**

* dst: Reference to write the merged [Dimension](index.odt/index.html" \l "classngraph_1_1Dimension) into.
* d1: First dimension to merge.
* d2: Second dimension to merge.

bool broadcast\_merge(Dimension &*dst*, *const* Dimension *d1*, *const* Dimension *d2*)[¶](#_CPPv4N9Dimension15broadcast_mergeER9DimensionK9DimensionK9Dimension)

Try to merge two [Dimension](index.odt/index.html" \l "classngraph_1_1Dimension) objects together with implicit broadcasting of unit-sized dimension to non unit-sized dimension.

*static* [Dimension](index.odt/index.html" \l "_CPPv4N6ngraph9DimensionE) dynamic()[¶](#_CPPv4N6ngraph9Dimension7dynamicEv)

Create a dynamic dimension.

**Return**

A dynamic dimension.

Public Static Attributes

*const* int64\_t s\_dynamic\_val = {(std::numeric\_limits<int64\_t>::max())}[¶](#_CPPv4N6ngraph9Dimension13s_dynamic_valE)

Constant for the value used internally to represent a dynamic dimension.

## Working with Backends[¶](#working-with-backends)

* [What is a backend?](#what-is-backend)
* [How to use?](#how-to-use)
* [nGraph bridge](#ngraph-bridge)
* [OpenCL](#opencl)

### What is a backend?[¶](#what-is-a-backend)

In the nGraph Compiler stack, what we call a *backend* is responsible for function execution and value allocation. A backend can be used to [carry out computations](index.odt/index.html" \l "document-core/constructing-graphs/execute) from a framework on a CPU, GPU, or ASIC; it can also be used with an *Interpreter* mode, which is primarily intended for testing, to analyze a program, or to help a framework developer customize targeted solutions.

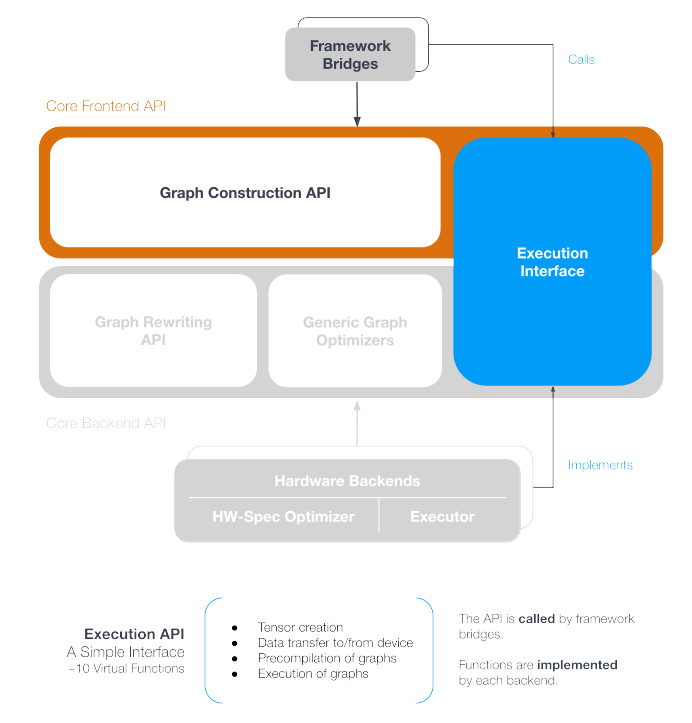
nGraph also provides a way to use the advanced tensor compiler PlaidML as a backend; you can learn more about this backend and how to build it from source in our documentation: [Building nGraph-PlaidML from source](index.odt/index.html" \l "ngraph-plaidml-backend).

| Backend | Current nGraph support | Future nGraph support |
| --- | --- | --- |
| Intel® Architecture Processors (CPUs) | Yes | Yes |
| Intel® Nervana™ Neural Network Processor™ (NNPs) | Yes | Yes |
| AMD\* GPUs | Yes | Some |

Each backend must define a function ngraph\_register\_${backend}\_backend that registers a backend constructor function and ensures that initializations are performed. An example that includes initializations can be found in the ngraph/src/runtime/cpu/cpu\_backend.cpp file. See also: [Backend APIs](index.odt/index.html" \l "backend-api-macros).

### How to use?[¶](#how-to-use)

1. Create a Backend; think of it as a compiler.
2. A Backend can then produce an Executable by calling compile.
3. A single iteration of the executable is executed by calling the call method on the Executable object.

[](index.odt/_images/execution-interface-run-graph.svg)

The execution interface for nGraph

The nGraph execution API for Executable objects is a simple, five-method interface; each backend implements the following five functions:

* The create\_tensor() method allows the bridge to create tensor objects in host memory or an accelerator’s memory.
* The write() and read() methods are used to transfer raw data into and out of tensors that reside in off-host memory.
* The compile() method instructs the backend to prepare an nGraph function for later execution.
* And, finally, the call() method is used to invoke an nGraph function against a particular set of tensors.

### How to display ngraph-related passes executed during runtime?[¶](#how-to-display-ngraph-related-passes-executed-during-runtime)

One easy way to get info about passes is to set the environment variable NGRAPH\_PROFILE\_PASS\_ENABLE=1. With this set, the pass manager will dump the name and execution time of each pass.

### nGraph bridge[¶](#ngraph-bridge)

When specified as the generic backend – either manually or automatically from a framework – NGRAPH defaults to CPU, and it also allows for additional device configuration or selection.

Because nGraph can select backends, specifying the INTELGPU backend as a runtime environment variable also works if one is present in your system:

NGRAPH\_TF\_BACKEND="INTELGPU"

An [axpy.py example](https://github.com/tensorflow/ngraph-bridge/blob/master/examples/axpy.py) is optionally available to test; outputs will vary depending on the parameters specified.

NGRAPH\_TF\_BACKEND="INTELGPU" python3 axpy.py

* NGRAPH\_INTELGPU\_DUMP\_FUNCTION – dumps nGraph’s functions in dot format.

### OpenCL[¶](#opencl)

OpenCL is only needed for the [PlaidML from nGraph](index.odt/index.html" \l "document-backends/plaidml-ng-api/index); if you have only a CPU backend, it is not needed.

1. Install the latest Linux driver for your system. You can find a list of drivers at <https://software.intel.com/en-us/articles/opencl-drivers>; You may need to install [OpenCL SDK](https://software.intel.com/en-us/opencl-sdk) in case of an libOpenCL.so absence.
2. Any user added to “video” group:

sudo usermod –a –G video <user\_id>

may, for example, be able to find details at the /sys/module/[system]/parameters/ location.

## Backend APIs[¶](#backend-apis)

### Backend APIs[¶](#backend-apis)

Each backend BACKEND needs to define the macro ${BACKEND}\_API appropriately to import symbols referenced from outside the library, and to export them from within the library. See any of the ${backend}\_backend\_visibility header files for an example; see also [What is a backend?](index.odt/index.html" \l "what-is-backend)

### DynamicBackend[¶](#dynamicbackend)

*class* DynamicBackend : *public* Backend[¶](#_CPPv4N6ngraph7runtime7dynamic14DynamicBackendE)

Wrapper class used to provide dynamic tensor support on backends that otherwise do not support dynamic tensors.

The main function of this class is to intercept create\_dynamic\_tensor and compile:

* create\_dynamic\_tensor will return a special DynamicTensor object whose shape can be updated after creation. Internally, DynamicTensor wraps static tensors managed by the wrapped backend.
* compile will return a special DynamicExecutable object, which allows dynamic shapes to be supported via graph cloning.

This class is instantiated by ngraph::runtime::Backend::create.

### PlaidML from nGraph[¶](#plaidml-from-ngraph)

*class* PlaidML\_Backend : *public* Backend[¶](#_CPPv4N6ngraph7runtime7plaidml15PlaidML_BackendE)

As of version 0.15, there is a new backend API to work with functions that can be compiled as a runtime Executable. Where previously Backend used a shared\_ptr<Function> as the handle passed to the call method to execute a compiled object, the addition of the shared\_ptr<Executable> object has more direct methods to actions such as validate, call, get\_performance\_data, and so on. This new API permits any executable to be saved or loaded *into* or *out of* storage and makes it easier to distinguish when a Function is compiled, thus making the internals of the Backend and Executable easier to implement.

## Distributed Training[¶](#distributed-training)

TBD WIP

## Quantization-Aware Training[¶](#quantization-aware-training)

Quantization-Aware Training is a technique used to quantize models during the training process. The main idea is that the quantization is emulated in the forward path by inserting some “Quantization” and “De-Quantization” nodes (Q-DQ) several places in the network to emulate the inference quantization noise. The expectation is the backward propagation will alter the weights so that they will adapt to this noise, and the result loss will be much better than traditional Post-Training Quantization.

For the weights, it is also common to take different quantization functions that cut off outliers. Some examples are available in the [Distiller guide](https://nervanasystems.github.io/distiller/algo_quantization.html" \l "quantization-aware-training). Distiller is an open-source Python package for neural network compression research. Network compression can reduce the footprint of a neural network, increase its inference speed, and save energy. Additionally, a framework for pruning, regularization and quantization algorithms is provided. A set of tools for analyzing and evaluating compression performance on previously-known State-of-the-Art (SotA) algorithms

When using QAT techniques, the position in which the Q-DQ ops are placed needs to align with the fusions hardware does for inference.

## Validated Workloads[¶](#validated-workloads)

We have validated performance [[1]](#f1) for the following workloads:

* [CPU Tensorflow](#cpu-tensorflow)
* [CPU ONNX](#cpu-onnx)
* [GPU TensorFlow](#gpu-tensorflow)
* [GPU ONNX](#gpu-onnx)

### [CPU Tensorflow](#id6)[¶](#cpu-tensorflow)

| TensorFlow Workload | Genre of Deep learning |
| --- | --- |
| Resnet50 v1 | Image recognition |
| Resnet50 v2 | Image recognition |
| Inception V3 | Image recognition |
| Inception V4 | Image recognition |
| Inception-ResNetv2 | Image recognition |
| MobileNet v1 | Image recognition |
| Faster RCNN | Object detection |
| VGG16 | Image recognition |
| SSD-VGG16 | Object detection |
| SSD-MobileNetv1 | Object detection |
| R-FCN | Object detection |
| Yolo v2 | Object detection |
| Transformer-LT | Language translation |
| Wide & Deep | Recommender system |
| NCF | Recommender system |
| U-Net | Image segmentation |
| DCGAN | Generative adversarial network |
| DRAW | Image generation |
| A3C | Reinforcement learning |

### [CPU ONNX](#id7)[¶](#cpu-onnx)

Additionally, we validated the following workloads are functional through [nGraph ONNX importer](https://github.com/NervanaSystems/ngraph-onnx/blob/master/README.md). ONNX models can be downloaded from the [ONNX Model Zoo](https://github.com/onnx/models).

| ONNX Workload | Genre of Deep Learning |
| --- | --- |
| DenseNet-121 | Image recognition |
| Inception-v1 | Image recognition |
| Inception-v2 | Image recognition |
| ResNet-50 | Image recognition |
| Mobilenet | Image recognition |
| Shufflenet | Image recognition |
| SqueezeNet | Image recognition |
| VGG-16 | Image recognition |
| ZFNet-512 | Image recognition |
| MNIST | Image recognition |
| Emotion-FERPlus | Image recognition |
| BVLC AlexNet | Image recognition |
| BVLC GoogleNet | Image recognition |
| BVLC CaffeNet | Image recognition |
| BVLC R-CNN ILSVRC13 | Object detection |
| ArcFace | Face Detection and Recognition |

### [GPU TensorFlow](#id8)[¶](#gpu-tensorflow)

| TensorFlow Workload | Genre of Deep Learning |
| --- | --- |
| Resnet50 v2 | Image recognition |
| Inception V3 | Image recognition |
| Inception V4 | Image recognition |
| Inception-ResNetv2 | Image recognition |
| VGG-16 | Image recognition |

### [GPU ONNX](#id9)[¶](#gpu-onnx)

| ONNX Workload | Genre of Deep Learning |
| --- | --- |
| Inception V1 | Image recognition |
| Inception V2 | Image recognition |
| ResNet-50 | Image recognition |
| SqueezeNet | Image recognition |

Important

Please see Intel’s [Optimization Notice](https://software.intel.com/en-us/articles/optimization-notice) for details on disclaimers.

Footnotes

|  |  |
| --- | --- |
| [[1]](#id1) | Benchmarking performance of DL systems is a young discipline; it is a good idea to be vigilant for results based on atypical distortions in the configuration parameters. Every topology is different, and performance changes can be attributed to multiple causes. Also watch out for the word “theoretical” in comparisons; actual performance should not be compared to theoretical performance. |

Contents

* [Diagnostics](#diagnostics)
  + [Compile Flags](#compile-flags)
  + [Environment Variables](#environment-variables)
    - [Debug Tracer](#debug-tracer)
    - [Intra-op and inter-op parallelism](#intra-op-and-inter-op-parallelism)
    - [Looking at graph objects](#looking-at-graph-objects)

## [Diagnostics](#id2)[¶](#diagnostics)

Important

Many of the following flags may be experimental only and subject to change.

Build nGraph with various compile flags and environment variables to diagnose performance and memory issues. See also [Performance testing with nbench](index.odt/index.html" \l "document-inspection/profiling).

### [Compile Flags](#id3)[¶](#compile-flags)

| Compile Flag | Description | Default Value |
| --- | --- | --- |
| NGRAPH\_CODE\_COVERAGE\_ENABLE | Enable code coverage data collection | FALSE |
| NGRAPH\_DEBUG\_ENABLE | Enable output for NGRAPH\_DEBUG statements | FALSE |
| NGRAPH\_DEPRECATED\_ENABLE | Enable compiler deprecation pragmas for deprecated APIs (recommended only for development use) | FALSE |
| NGRAPH\_DEX\_ONLY | Build CPU DEX without codegen | FALSE |
| NGRAPH\_DISTRIBUTED\_ENABLE | Enable distributed training using MLSL/OpenMPI | OFF |
| NGRAPH\_DISTRIBUTED\_MLSL\_ENABLE | Use MLSL | OFF |
| NGRAPH\_DOC\_BUILD\_ENABLE | Automatically build documentation | OFF |
| NGRAPH\_FAST\_MATH\_ENABLE | Enable fast math | ON |
| NGRAPH\_HALIDE |  | OFF |
| NGRAPH\_INTERPRETER\_ENABLE | Control the building of the INTERPRETER backend | TRUE |
| NGRAPH\_INTERPRETER\_STATIC\_LIB\_ENABLE | Enable build INTERPRETER backend static library | FALSE |
| NGRAPH\_JSON\_ENABLE | Enable JSON based serialization and tracing features | TRUE |
| NGRAPH\_LIB\_VERSIONING\_ENABLE | Enable shared library versioning | FALSE |
| NGRAPH\_MLIR\_ENABLE | Control the building of MLIR backend | FALSE |
| NGRAPH\_NOP\_ENABLE | Control the building of the NOP backend | TRUE |
| NGRAPH\_ONNX\_IMPORT\_ENABLE | Enable ONNX importer | FALSE |
| NGRAPH\_PLAIDML\_ENABLE | Enable the PlaidML backend | ${PLAIDML\_FOUND} |
| NGRAPH\_PYTHON\_BUILD\_ENABLE | Enable build of NGRAPH python package wheel | FALSE |
| NGRAPH\_STATIC\_LIB\_ENABLE | Enable build NGRAPH static library | FALSE |
| NGRAPH\_TBB\_ENABLE | Only if (NGRAPH\_CPU\_ENABLE) Control usage of TBB for CPU backend | TRUE |
| NGRAPH\_TOOLS\_ENABLE | Control the building of tools | TRUE |
| NGRAPH\_UNIT\_TEST\_ENABLE | Control the building of unit tests | TRUE |
| NGRAPH\_USE\_PREBUILT\_LLVM | Use a precompiled LLVM | FALSE |
| NGRAPH\_USE\_PREBUILT\_MLIR | Use the [precompiled MLIR](https://github.com/IntelAI/mlir) | FALSE |

### [Environment Variables](#id4)[¶](#environment-variables)

Important

Many of the following flags may be experimental only and subject to change.

| Environment Variable | Description |
| --- | --- |
| NGRAPH\_DISABLE\_LOGGING | Disable printing all logs irrespective of build type |
| NGRAPH\_DISABLED\_FUSIONS | Disable specified fusions. Specified as ; separated list and supports regex |
| NGRAPH\_ENABLE\_REPLACE\_CHECK | Enables strict type checking in copy constructor copy\_with\_new\_args |
| NGRAPH\_ENABLE\_SERIALIZE\_TRACING | generates 1 json file per pass to run with nbench for localized execution rather than whole stack execution |
| NGRAPH\_ENABLE\_TRACING | Enables creating graph execution timelines to be viewed in chrome://tracing see also [General Visualization Tools](index.odt/index.html" \l "document-inspection/viz_tools). |
| NGRAPH\_ENABLE\_VISUALIZE\_TRACING | Enables creating visual graph for each pass .svg files by default; see also [General Visualization Tools](index.odt/index.html" \l "document-inspection/viz_tools) |
| NGRAPH\_FAIL\_MATCH\_AT | Allows one to specify node name patterns to abort pattern matching at particular nodes. Helps debug an offending fusion |
| NGRAPH\_GTEST\_INFO | Enables printing info about a specific test |
| NGRAPH\_INTER\_OP\_PARALLELISM | See [Intra-op and inter-op parallelism](#interop-intraop) |
| NGRAPH\_INTRA\_OP\_PARALLELISM | See [Intra-op and inter-op parallelism](#interop-intraop) |
| NGRAPH\_PASS\_ATTRIBUTES | Specify pass-specific attributes as a semi-colon separated list to be enabled or disabled. Naming of pass attributes is up to the backends and see also [pass config](https://github.com/NervanaSystems/ngraph/blob/a4a3031bb40f19ec28704f76de39762e1f27e031/src/ngraph/pass/pass_config.cpp" \l "L54) |
| NGRAPH\_PASS\_ENABLES | Specify a semi-colon separated list to enable or disable a pass on core or backend. This will override the default enable/disable values |
| NGRAPH\_PROFILE\_PASS\_ENABLE | Dump the name and execution time of each pass; shows per-pass time taken to compile |
| NGRAPH\_PROVENANCE\_ENABLE | Enable adding provenance info to nodes. This will also be added to serialized files. |
| NGRAPH\_SERIALIZER\_OUTPUT\_SHAPES | Enable adding output shapes in the serialized graph |
| NGRAPH\_VISUALIZE\_EDGE\_JUMP\_DISTANCE | Calculated in code; helps prevent *long* edges between two nodes very far apart |
| NGRAPH\_VISUALIZE\_EDGE\_LABELS | Set it to 1 in /.bashrc; adds label to a graph edge when NGRAPH\_ENABLE\_VISUALIZE\_TRACING=1 |
| NGRAPH\_VISUALIZE\_TREE\_OUTPUT\_SHAPES | Set it to 1 in /.bashrc; adds output shape of a node when NGRAPH\_ENABLE\_VISUALIZE\_TRACING=1 |
| NGRAPH\_VISUALIZE\_TREE\_OUTPUT\_TYPES | Set it to 1 in /.bashrc; adds output type of a node when NGRAPH\_ENABLE\_VISUALIZE\_TRACING=1 |
| NGRAPH\_VISUALIZE\_TRACING\_FORMAT | Default format is .svg. See also [General Visualization Tools](index.odt/index.html" \l "document-inspection/viz_tools) |
| OMP\_NUM\_THREADS | See: [OpenMPI Runtime Library Documentation](https://www.openmprtl.org/documentation) |

#### [Debug Tracer](#id5)[¶](#debug-tracer)

Another diagnostic configuration option is to activate NGRAPH\_CPU\_DEBUG\_TRACER, a runtime environment variable that supports extra logging and debug detail.

This is a useful tool for data scientists interested in outputs from logtrace files that can, for example, help in tracking down model convergences. It can also help engineers who might want to add their new Backend to an existing framework to compare intermediate tensors/values to references from a CPU backend.

To activate this tool, set the env var NGRAPH\_CPU\_DEBUG\_TRACER=1. It will dump trace\_meta.log and trace\_bin\_data.log. The names of the logfiles can be customized.

To specify the names of logs with those flags:

NGRAPH\_TRACER\_LOG = "meta.log"

NGRAPH\_BIN\_TRACER\_LOG = "bin.log"

#### [Intra-op and inter-op parallelism](#id6)[¶](#intra-op-and-inter-op-parallelism)

* intra\_op\_parallelism\_threads
* inter\_op\_parallelism\_threads

Some frameworks, like TensorFlow\*, use these settings to improve performance; however, they are often not sufficient for optimal performance. Framework-based adjustments cannot access the underlying NUMA configuration in multi-socket Intel® Xeon® processor-based platforms, which is a key requirement for many kinds of inference-engine computations.

The meta\_log contains:

kernel\_name, serial\_number\_of\_op, tensor\_id, symbol\_of\_in\_out, num\_elements, shape, binary\_data\_offset, mean\_of\_tensor, variance\_of\_tensor

A line example from a unit-test might look like:

K=Add S=0 TID=0\_0 >> size=4 Shape{2, 2} bin\_data\_offset=8 mean=1.5 var=1.25

The binary\_log line contains:

tensor\_id, binary data (tensor data)

A reference for the implementation of parsing these logfiles can also be found in the unit test for this feature.

#### [Looking at graph objects](#id7)[¶](#looking-at-graph-objects)

A number of nGraph objects can print themselves on streams. For example,``cerr << a + b`` produces v0::Add Add\_2(Parameter\_0[0]:f32{2,3}, Parameter\_1[0]:f32{2,3}):(f32{2,3}) indicating the specific version of the op, its name, arguments, and outputs.

## Debug TensorFlow\*[¶](#debug-tensorflow)

Note

These flags are all disabled by default

For profiling with TensorFlow\* and nbench, see [Use nbench to ease end-to-end debugging for TensorFlow\*](index.odt/index.html" \l "nbench-tf).

| Flag | Description |
| --- | --- |
| NGRAPH\_ENABLE\_SERIALIZE=1 | Generate nGraph-level serialized graphs |
| NGRAPH\_TF\_VLOG\_LEVEL=5 | Generate ngraph-tf logging info for different passes |
| NGRAPH\_TF\_LOG\_PLACEMENT=1 | Generate op placement log at stdout |
| NGRAPH\_TF\_DUMP\_CLUSTERS=1 | Dump Encapsulated TF Graphs formatted as NGRAPH\_cluster\_<cluster\_num> |
| NGRAPH\_TF\_DUMP\_GRAPHS=1 | Dump TF graphs for different passes: precapture, capture, unmarked, marked, clustered, declustered, encapsulated |
| TF\_CPP\_MIN\_VLOG\_LEVEL=1 | Enable TF CPP logs |
| NGRAPH\_TF\_DUMP\_DECLUSTERED\_GRAPHS=1 | Dump graphs with final clusters assigned. Use this to view TF computation graph with colored nodes indicating clusters |
| NGRAPH\_TF\_USE\_LEGACY\_EXECUTOR | This flag will be obsolete soon. |

## Debug ONNX[¶](#debug-onnx)

Note

These flags are all disabled by default

| Flag | Description |
| --- | --- |
| ONNXRUNTIME\_NGRAPH\_DUMP\_OPS | Dumps ONNX ops |
| ONNXRUNTIME\_NGRAPH\_LRU\_CACHE\_SIZE | Modify LRU cache size (NGRAPH\_EP\_LRU\_CACHE\_DEFAULT\_SIZE 500) |

## Debug PaddlePaddle\*[¶](#debug-paddlepaddle)

PaddlePaddle has its [own env vars](https://github.com/PaddlePaddle/Paddle/blob/cdd46d7e022add8de56995e681fa807982b02124/python/paddle/fluid/__init__.py" \l "L161-L227).

## General Visualization Tools[¶](#general-visualization-tools)

nGraph provides serialization and deserialization facilities, along with the ability to create image formats or a PDF.

NGRAPH\_ENABLE\_VISUALIZE\_TRACING=1 enables visualization and generates graph visualization files.

Note

Using NGRAPH\_ENABLE\_VISUALIZE\_TRACING=1 will affect performance.

When visualization is enabled, svg files for your graph get generated. The default format can be adjusted by setting the NGRAPH\_VISUALIZE\_TRACING\_FORMAT flag to another format, like PNG or PDF.

Note

Large graphs are usually not legible with formats like PDF.

Large graphs may require additional work to get into a human-readable format. On the back end, very long edges will need to be cut to make (for example) a hard-to-render training graph tractable. This can be a tedious process, so incorporating the help of a rendering engine or third-party tool like one listed below may be useful.

1. [Gephi](https://gephi.org/)
2. [Cytoscape](https://cytoscape.org/)
3. [Netron](https://github.com/lutzroeder/netron/blob/master/README.md)

## Performance testing with nbench[¶](#performance-testing-with-nbench)

The nGraph Compiler stack includes the nbench tool which provides additional methods of assessing or debugging performance issues.

If you follow the build process under [Build and Test](index.odt/index.html" \l "document-buildlb), the NGRAPH\_TOOLS\_ENABLE flag defaults to ON and automatically builds nbench. As its name suggests, nbench can be used to benchmark any nGraph-serialized model with a given backend.

To benchmark an already-serialized nGraph .json model with, for example, a CPU backend, run nbench as follows.

$ cd ngraph/build/src/tools

$ nbench/nbench -b CPU - i 1 -f <serialized\_json file>

Samples for testing can be found under ngraph/test/models.

### nbench[¶](#nbench)

Benchmark and nGraph JSON model with a given backend.

SYNOPSIS

nbench [-f <filename>] [-b <backend>] [-i <iterations>]

OPTIONS

-f|--file Serialized model file

-b|--backend Backend to use (default: CPU)

-d|--directory Directory to scan for models. All models are benchmarked.

-i|--iterations Iterations (default: 10)

-s|--statistics Display op statistics

-v|--visualize Visualize a model (WARNING: requires Graphviz installed)

--timing\_detail Gather detailed timing

-w|--warmup\_iterations Number of warm-up iterations

--no\_copy\_data Disable copy of input/result data every iteration

--dot Generate Graphviz dot file

--double\_buffer Double buffer inputs and outputs

#### Use nbench to ease end-to-end debugging for TensorFlow\*[¶](#use-nbench-to-ease-end-to-end-debugging-for-tensorflow)

Rather than run a TensorFlow\* model “end-to-end” all the time, developers who notice a problem with performance or memory usage can generate a unique serialized model for debugging by using NGRAPH\_ENABLE\_SERIALIZE=1. This serialized model can then be run and re-run with nbench to efficiently experiment with any changes in ngraph space; developers can make changes and test changes without the overhead of a complete end-to-end compilation for each change.

#### Find or display version[¶](#find-or-display-version)

If you’re working with the [Python API](index.odt/index.html" \l "document-python_api/index), the following command may be useful:

python3 -c "import ngraph as ng; print('nGraph version: ',ng.\_\_version\_\_)";

To manually build a newer version than is available from the latest [PyPI](https://pypi.org/project/ngraph-core/) (Python Package Index), see our nGraph Python API [BUILDING.md](https://github.com/NervanaSystems/ngraph/blob/master/python/BUILDING.md) documentation.

## [Contribution Guide](#id3)[¶](#contribution-guide)

Contents

* [Contribution Guide](#contribution-guide)
  + [License](#license)
  + [Code formatting](#code-formatting)
    - [Adding ops to nGraph Core](#adding-ops-to-ngraph-core)
    - [Coding style](#coding-style)
      * [Names](#names)
      * [Namespaces](#namespaces)
      * [File Names](#file-names)
      * [Formatting](#formatting)
  + [Contributing to documentation](#contributing-to-documentation)
    - [Documenting source code examples](#documenting-source-code-examples)
    - [Adding captions to code blocks](#adding-captions-to-code-blocks)
    - [How to build the documentation](#how-to-build-the-documentation)

### [License](#id4)[¶](#license)

All contributed code must be compatible with the [Apache 2](https://www.apache.org/licenses/LICENSE-2.0) license, preferably by being contributed under the Apache 2 license. Code contributed with another license will need the license reviewed by Intel before it can be accepted.

### [Code formatting](#id5)[¶](#code-formatting)

All C/C++ source code in the repository, including the test code, must adhere to the source-code formatting and style guidelines described here. The coding style described here applies to the nGraph repository. Related repositories may make adjustments to better match the coding styles of libraries they are using.

#### [Adding ops to nGraph Core](#id6)[¶](#adding-ops-to-ngraph-core)

Our design philosophy is that the graph is not a script for running optimized kernels; rather, the graph is a specification for a computation composed of basic building blocks which we call ops. Compilation should match groups of ops to appropriate optimal semantically equivalent groups of kernels for the backend(s) in use. Thus, we expect that adding of new Core ops should be infrequent and that most functionality instead gets added with new functions that build sub-graphs from existing core ops.

#### [Coding style](#id7)[¶](#coding-style)

We have a coding standard to help us to get development done. If part of the standard is impeding progress, we either adjust that part or remove it. To this end, we employ coding standards that facilitate understanding of *what nGraph components are doing*. Programs are easiest to understand when they can be understood locally; if most local changes have local impact, you do not need to dig through multiple files to understand what something does and if it is safe to modify.

##### [Names](#id8)[¶](#names)

Names should *briefly* describe the thing being named and follow these casing standards:

* Define C++ class or type names with CamelCase.
* Assign template parameters with UPPER\_SNAKE\_CASE.
* Case variable and function names with lower\_snake\_case.

Method names for basic accessors are prefixed by get\_, is\_, or set\_ and should have simple \(\mathcal{O}(1)\) implementations:

* A get\_ method should be externally idempotent. It may perform some simple initialization and cache the result for later use. Trivial get\_ methods can be defined in a header file. If a method is non-trivial, that is often a sign that it is not a basic accessor.
* An is\_ may be used instead of get\_ for boolean accessors.
* A set\_ method should change the value returned by the corresponding get\_ method.
  + Use set\_is\_ if using is\_ to get a value.
  + Trivial set\_ methods may be defined in a header file.
* Names of variables should indicate the use of the variable.
  + Member variables should be prefixed with m\_.
  + Static member variables should be rare and be prefixed with s\_.
* Do not use using to define a type alias at top-level in header file. If the abstraction is useful, give it a class.
  + C++ does not enforce the abstraction. For example if X and Y are aliases for the same type, you can pass an X to something expecting a Y.
  + If one of the aliases were later changed, or turned into a real type, many callers could require changes.

##### [Namespaces](#id9)[¶](#namespaces)

* ngraph is for the public API, although this is not currently enforced.
  + Use a nested namespace for implementation classes.
  + Use an unnamed namespace or static for file-local names. This helps prevent unintended name collisions during linking and when using shared and dynamically-loaded libraries.
  + Never use using at top-level in a header file.
    - Doing so leaks the alias into users of the header, including headers that follow.
    - It is okay to use using with local scope, such as inside a class definiton.
  + Be careful of C++’s implicit namespace inclusions. For example, if a parameter’s type is from another namespace, that namespace can be visible in the body.
  + Only use using std and/or using ngraph in .cpp files. using a nested namespace has can result in unexpected behavior.

##### [File Names](#id10)[¶](#file-names)

* Do not use the same file name in multiple directories. At least one IDE/debugger ignores the directory name when setting breakpoints.
* Use .hpp for headers and .cpp for implementation.
* Reflect the namespace nesting in the directory hierarchy.
* Unit test files are in the tests directory.
  + Transformer-dependent tests are tests running on the default transformer or specifying a transformer. For these, use the form

TEST(file\_name, test\_name)

* + Transformer-independent tests:
    - File name is file\_name.in.cpp
    - Add #include "test\_control.hpp" to the file’s includes
    - Add the line static std::string s\_manifest = "${MANIFEST}"; to the top of the file.
    - Use

NGRAPH\_TEST(${BACKEND\_NAME}, test\_name)

for each test. Files are generated for each transformer and the ${BACKEND\_NAME} is replaced with the transformer name.

Individual unit tests may be disabled by adding the name of the test to the unit\_test.manifest file found in the transformer’s source file directory.

##### [Formatting](#id11)[¶](#formatting)

Things that look different should look different because they are different. We use **clang format** to enforce certain formatting. Although not always ideal, it is automatically enforced and reduces merge conflicts.

* The .clang-format file located in the root of the project specifies our format. Simply run:

$ make style-check

$ make style-apply

* Formatting with #include files:
  + Put headers in groups separated by a blank line. Logically order the groups downward from system-level to 3rd-party to ngraph.
  + Formatting will keep the files in each group in alphabetic order.
  + Use this syntax for files that **do not change during nGraph development**; they will not be checked for changes during builds. Normally this will be everything but the ngraph files:

#include <file>

* + Use this syntax for files that **are changing during nGraph development**; they will be checked for changes during builds. Normally this will be ngraph headers:

#include "file"

* + Use this syntax for system C headers with C++ wrappers:

#include <c...>

* To guard against multiple inclusion, use:

#pragma once

* + The syntax is a compiler extension that has been adopted by all supported compilers.
* The initialization

Foo x{4, 5};

is preferred over

Foo x(4, 5);

* Indentation should be accompanied by braces; this includes single-line bodies for conditionals and loops.
* Exception checking:
  + Throw an exception to report a problem.
  + Nothing that calls abort, exit or terminate should be used. Remember that ngraph is a guest of the framework.
  + Do not use exclamation points in messages!
  + Be as specific as practical. Keep in mind that the person who sees the error is likely to be on the other side of the framework and the message might be the only information they see about the problem.
* If you use auto, know what you are doing. auto uses the same type-stripping rules as template parameters. If something returns a reference, auto will strip the reference unless you use auto&:
  + Don’t do things like

auto s = Shape{2,3};

Instead, use

Shape s{2, 3};

* + Indicate the type in the variable name.
* One variable declaration/definition per line
  + Don’t use the C-style

int x, y, \*z;

Instead, use:

int x;

int y;

int\* z;

To contribute documentation for your code, please see the [Contributing to documentation](index.odt/index.html" \l "document-project/doc-contributor-README).

|  |  |
| --- | --- |
| orphan: |  |

### [Contributing to documentation](#id12)[¶](#contributing-to-documentation)

Note

Tips for contributors who are new to the highly-dynamic environment of documentation in AI software:

* A good place to start is “document something you figured out how to get working”. Content changes and additions should be targeted at something more specific than “developers”. If you don’t understand how varied and wide the audience is, you’ll inadvertently break or block things.
* There are experts who work on all parts of the stack; try asking how documentation changes ought to be made in their respective sections.
* Start with something small. It is okay to add a “patch” to fix a typo or suggest a word change; larger changes to files or structure require research and testing first, as well as some logic for why you think something needs changed.
* Most documentation should wrap at about 80. We do our best to help authors source-link and maintain their own code and contributions; overwriting something already documented doesn’t always improve it.
* Be careful editing files with links already present in them; deleting links to papers, citations, or sources is discouraged.
* Please do not submit Jupyter\* notebook code to the nGraph Library or core repos; best practice is to maintain any project-specific examples, tests, or walk-throughs in a separate repository and to link back to the stable op or Ops that you use in your project.

For updates within the nGraph Library /doc repo, please submit a PR with any changes or ideas you’d like integrated. This helps us maintain trackability with respect to changes made, additions, deletions, and feature requests.

If you prefer to use a containerized application, like Jupyter\* notebooks, Google Docs\*, the GitHub\* GUI, or MS Word\* to explain, write, or share documentation contributions, you can convert the doc/sphinx/source/\*.rst files to another format with a tool like pypandoc and share a link to your efforts on our [wiki](https://github.com/NervanaSystems/ngraph/wiki/).

Another option is to fork the [ngraph repo](https://github.com/NervanaSystems/ngraph/), essentially snapshotting it at that point in time, and to build a Jupyter\* notebook or other set of docs around it for a specific use case. Add a note on our wiki to show us what you did; new and novel applications may have their projects highlighted on an upcoming [ngraph.ai](https://www.ngraph.ai/) release.

Note

Please do not submit Jupyter\* notebook code to the nGraph Library or core repos; best practice is to maintain any project-specific examples, tests, or walk-throughs in a separate repository.

#### [Documenting source code examples](#id13)[¶](#documenting-source-code-examples)

When **verbosely** documenting functionality of specific sections of code – whether they are entire code blocks within a file, or code strings that are **outside** the nGraph Library’s [documentation repo](https://github.com/NervanaSystems/ngraph/tree/master/doc), here is an example of best practice:

Say a file has some interesting functionality that could benefit from more explanation about one or more of the pieces in context. To keep the “in context” navigable, write something like the following in your .rst documentation source file:

.. literalinclude:: ../../../examples/abc/abc.cpp

:language: cpp

:lines: 20-31

And the raw code will render as follows

using namespace ngraph;

int main()

{

// Build the graph

Shape s{2, 3};

auto a = std::make\_shared<op::Parameter>(element::f32, s);

auto b = std::make\_shared<op::Parameter>(element::f32, s);

auto c = std::make\_shared<op::Parameter>(element::f32, s);

auto t0 = std::make\_shared<op::Add>(a, b);

You can now verbosely explain the code block without worrying about breaking the code. The trick here is to add the file you want to reference relative to the folder where the Makefile is that generates the documentation you’re writing.

See the **note** at the bottom of this page for more detail about how this works in the current 0.29 version of nGraph Library documentation.

#### [Adding captions to code blocks](#id14)[¶](#adding-captions-to-code-blocks)

One more trick to helping users understand exactly what you mean with a section of code is to add a caption with content that describes your parsing logic. To build on the previous example, let’s take a bigger chunk of code, add some line numbers, and add a caption:

.. literalinclude:: ../../../examples/abc/abc.cpp

:language: cpp

:lines: 48-56

:caption: "caption for a block of code that initializes tensors"

and the generated output will show readers of your helpful documentation

“caption for a block of code that initializes tensors”[¶](#id2)

// Initialize tensors

float v\_a[2][3] = {{1, 2, 3}, {4, 5, 6}};

float v\_b[2][3] = {{7, 8, 9}, {10, 11, 12}};

float v\_c[2][3] = {{1, 0, -1}, {-1, 1, 2}};

t\_a->write(&v\_a, sizeof(v\_a));

t\_b->write(&v\_b, sizeof(v\_b));

t\_c->write(&v\_c, sizeof(v\_c));

Our documentation practices are designed around “write once, reuse” that we can use to prevent code bloat. See the [Contribution Guide](index.odt/index.html" \l "document-project/contribution-guide) for our code style guide.

#### [How to build the documentation](#id15)[¶](#how-to-build-the-documentation)

Note

Stuck on how to generate the html? Run these commands; they assume you start at a command line running within a clone (or a cloned fork) of the ngraph repo. You do **not** need to run a virtual environment to create documentation if you don’t want; running $ make clean in the doc/sphinx folder removes any generated files.

Right now the minimal version of Sphinx needed to build the documentation is Sphinx v. 1.7.5. This can be installed with **pip3**, either to a virtual environment, or to your base system if you plan to contribute much core code or documentation. For C++ API docs that contain inheritance diagrams and collaboration diagrams which are helpful for framework integratons, building bridge code, or creating a backend UI for your own custom framework, be sure you have a system capable of running [doxygen](http://www.doxygen.org/index.html).

To build documentation locally, run:

$ sudo apt-get install python3-sphinx

$ pip3 install Sphinx==1.7.5

$ pip3 install breathe numpy

$ cd doc/sphinx/

$ make html

$ cd build/html

$ python3 -m http.server 8000

Then point your browser at localhost:8000.

To build documentation in a python3 virtualenv, try:

$ python3 -m venv py3doc

$ . py3doc/bin/activate

(py3doc)$ pip install Sphinx breathe numpy

(py3doc)$ cd doc/sphinx

(py3doc)$ make html

(py3doc)$ cd build/html

(py3doc)$ python -m http.server 8000

Then point your browser at localhost:8000.

Note

For docs built in a virtual env, Sphinx latest changes may break documentation; try building with a specific version of Sphinx.

For tips on writing reStructuredText-formatted documentation, see the [sphinx](http://www.sphinx-doc.org/en/stable/rest.html) stable reST documentation.

## Glossary[¶](#glossary)

ANN

Artificial Neural Network, often abbreviated as [NN](#term-nn).

backend

A component that can execute computations.

bridge

A component of nGraph that acts as a backend for a framework, allowing the framework to define and execute computations.

builder

A builder is a function that creates a sub-graph and returns a root node to the bridge. Fused ops are preferred over builders See also: [Basic concepts](index.odt/index.html" \l "document-provenance/overview).

data-flow graph

Data-flow graphs are used to implement deep learning models. In a data-flow graph, nodes represent operations on data and edges represent data flowing between those operations.

dynamic tensor

A tensor whose shape can change from one “iteration” to the next. When created, a framework [bridge](#term-bridge) might supply only *partial* shape information: it might be **all** the tensor dimensions, **some** of the tensor dimensions, or **none** of the tensor dimensions; furthermore, the rank of the tensor may be left unspecified.

export

The serialized version of a trained model that can be passed to one of the nGraph backends for computation.

framework

Frameworks provide expressive user-facing APIs for constructing, training, validating, and deploying DL/ML models: TensorFlow\*, PaddlePaddle\*, MXNet\*, PyTorch\*, and Caffe\* are all examples of well-known frameworks.

function graph

The nGraph Library uses a function graph to represent an op’s parameters and results.

fusion

Fusion is the fusing, combining, merging, collapsing, or refactoring of a graph’s functional operations (ops) into one or more of nGraph’s core ops.

ISA

An acronym for “Instruction Set Architecture,” an ISA is machine code that is compatible with the underlying silicon architecture. A realization of an ISA is called an *implementation*. An ISA permits multiple implementations that may vary in performance, physical size, memory use or reuse, and monetary cost among other things. An ISA defines everything a machine-language programmer needs to know in order to program a particular backend device. What an ISA defines will differ among ISAs; in general, it defines things like:

* supported *data types*;
* physical *states* available, such as the main memory and registers;
* *semantics*, such as the memory consistency and addressing modes;
* *low-level machine instructions* that comprise a machine language;
* and the *input/output model*.

Be careful to not confuse ISAs with microarchitectures.

LSTM

LSTM is an acronym for “Long Short-Term Memory”. LSTMs extend on the traditional RNN by providing a number of ways to “forget” the memory of the previous time step via a set of learnable gates. These gates help avoid the problem of exploding or vanishing gradients that occur in the traditional RNN.

model description

A description of a program’s fundamental operations that are used by a framework to generate inputs for computation.

NN

NN is an acronym for “Neural Network”. NN models are used to simulate possible combinations of binary logic processing and multi-layer (multi-dimensional) paths through which a [data-flow graph](#term-data-flow-graph) may be mapped or computed. A NN does not have centralized storage; rather, a NN manifests as information stored as patterns throughout the network structure. NNs may be **Recurrent** (feedback loop) or **Nonrecurrent** (feed-forward) with regard to the network vector.

op

An op represents an operation. Ops are stateless and have zero or more inputs and zero or more outputs. Some ops have additional constant attributes. Every output of an op corresponds to a tensor and has an element type and a shape. The element types and shapes of the outputs of an op are determined by the inputs and attributes of the op.

parameter

In the context of a function graph, a “parameter” refers to what “stands in” for an argument in an op definition.

provenance

The term provenance refers to the matching of device code to framework sub-graphs; it is analogous to source code locators in conventional compilers, which associate regions of object code with source files and line numbers.

quantization

Quantization refers to the conversion of numerical data into a lower-precision representation. Quantization is often used in deep learning to reduce the time and energy needed to perform computations by reducing the size of data transfers and the number of steps needed to perform a computation. This improvement in speed and energy usage comes at a cost in terms of numerical accuracy, but deep learning models are often able to function well in spite of this reduced accuracy.

RANN

Recurrent Artificial Neural Network, often abbreviated as [RNN](#term-rnn).

result

In the context of a function graph, the term “result” refers to what stands in for the returned value.

RNN

A Recurrent Neural Network is a variety of [NN](#term-nn) where output nodes from a layer on a data-flow graph have loopback to nodes that comprise an earlier layer. Since the RNN has no “centralized” storage, this loopback is the means by which the ANN can “learn” or be trained. There are several sub-categories of RNNs. The traditional RNN looks like:

\(s\_t = tanh(dot(W,x\_{t-1}) + dot(U, s\_{t-1})\)

where \(x\) is the input data, \(s\) is the memory, and output is \(o\_t = softmax(dot(V, s\_t))\). [Tanh](index.odt/index.html" \l "document-ops/tanh), [Dot](index.odt/index.html" \l "document-ops/dot), and [Softmax](index.odt/index.html" \l "document-ops/softmax) are all nGraph [core Ops](index.odt/index.html" \l "document-ops/index).

SGD

Stochastic Gradient Descent, also known as incremental gradient descent, is an iterative method for optimizing a differentiable objective function.

shape

The shape of a tensor is a tuple of non-negative integers that represents an exclusive upper bound for coordinate values.

shape propagation

The static process by which assignment of every tensor (or, equivalently, every node output) in the graph is assigned **complete shape information**.

shared pointer

The C++ standard template library has the template std::shared\_ptr<X>. A shared pointer is used like an X\* pointer, but maintains a reference count to the underlying object. Each new shared pointer to the object increases the count. When a shared pointer goes out of scope, the reference count is decremented, and, when the count reaches 0, the underlying object is deleted. The function template std::make\_shared<X>(...) can be used similarly to new X(...), except it returns a std::shared\_ptr<X> instead of an X\*.

If there is a chain of shared pointers from an object back to itself, every object in the chain is referenced, so the reference counts will never reach 0 and the objects will never be deleted.

If a referenced b and b wanted to track all references to itself and shared pointers were used both directions, there would be a chain of pointers form a to itself. We avoid this by using shared pointers in only one direction, and raw pointers for the inverse direction. std::enabled\_shared\_from\_this is a class template that defines a method shared\_from\_this that provides a shared pointer from a raw pointer.

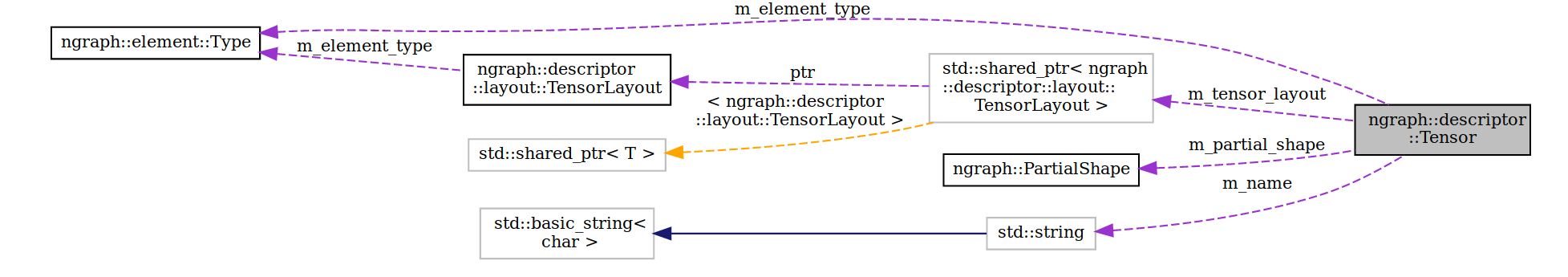
nGraph makes use of shared pointers for objects whose lifetime is hard to determine when they are allocated.

step

An abstract “action” that produces zero or more tensor outputs from zero or more tensor inputs. Steps correspond to *ops* that connect *nodes*.

tensors

Tensors are maps from *coordinates* to scalar values, all of the same type, called the *element type* of the tensor.

[](index.odt/_images/classngraph_tensor_descriptor.png)

Tensorview

The interface backends implement for tensor use. When there are no more references to the tensor view, it will be freed when convenient for the backend.

validated

To provide optimizations with nGraph, we first confirm that a given workload is “validated” as being functional; that is, we can successfully load its serialized graph as an nGraph [function graph](#term-function-graph)

## Release Notes[¶](#release-notes)

nGraph is provided as source code, APIs, build scripts, and some binary formats for various Compiler stack configurations and use cases.

For downloads formatted as .zip and tar.gz, see <https://github.com/NervanaSystems/ngraph/releases>.

This page includes additional documentation updates.

We are pleased to announce the release of version 0.29.

### Core updates for 0.29[¶](#core-updates-for-version)

* Constant folding improvements
* Pattern refactoring
* Serialization bug-fixes
* Build improvements

### Latest documentation updates[¶](#latest-documentation-updates)

* Improved documentation on Pad op

Important

Pre-releases (-rc-0.\*) have newer features, and are less stable.

#### Changelog on Previous Releases[¶](#changelog-on-previous-releases)

### 0.28[¶](#id2)

* Better debugging documentation
* Dynamic Shapes and APIs
* Provenance
* Add linkages and overview for quantization APIs
* New ngraph.ai themed illustrations

### 0.27.1[¶](#id3)

* Fixes broken serializer for Sum and Product
* New ops
* Provenance improvements from 0.25.1
* More dynamic shape ops
* More informative errors
* Additional details on quantization
* Index updates
* API updates
* All ops support Output<Node> arguments
* Additional ops
* ONNX handling unknown domains
* Provenance works with builders and fused ops
* RPATH for finding openmpi
* Negative indices/axes fixes
* Migrate some get\_argument removals
* Negative indices/axes fixes
* Better support for MKL-DNN 1.0 (DNNL)
* Additional constant element types
* Add new Sphinx-friendly theme (can be built natively for an alternative to ngraph.ai docs).
* Update PaddlePaddle documentation to reflect demo directories instead of example directory.
* Update doc regarding the validation of Sum op.

### 0.26.1[¶](#id4)

* Performance increase for ConstantFolding pass

### 0.25.1[¶](#id5)

* Allow DLLs that link nGraph statically to load backends
* Add rank id to trace file name
* Allow provenance merging to be disabled
* Remove some white-listed compiler warnings
* Provenance, builders, ops that make ops, and fused op expansions
* Note the only support for nGPU is now through PlaidML; nGraph support for nGPU (via cuDNN) has been deprecated.
* iGPU works only with nGraph version 0.24.

### 0.25.0[¶](#id6)

* Better PlaidML support
* Double-buffering support
* Constant folding
* Support for static linking
* Additional ops
* Preliminary static linking support
* Known issue: No PlaidML training support
* Doc: Add instructions how to build NGRAPH\_PLAIDML backend
* Published interim version of doc navigation for updates at ngraph.ai
* GPU validations: added 5 functional TensorFlow workloads and 4 functional ONNX workloads

### 0.24[¶](#id7)

* Fixes reshape sink/swim issue
* More ONNX ops
* Elementwise divide defaults to Python semantics
* GenerateMask seed optional
* Graph visualization improvements
* Preserve control dependencies in more places
* GetOutputElement has single input

### 0.23[¶](#id8)

* More ONNX ops
* Elementwise divide defaults to Python semantics
* GenerateMask seed optional
* Document new debug tool
* Graph visualization improvements
* Note deprecation of MXNet’s ngraph-mxnet PyPI
* Note default change to svg files for graphs and visualization
* Add more prominent tips for contributors who find the doc-contributor-README
* Better GSG / Install Guide structure.
* Added group edits and new illustrations from PR 2994 to introduction.rst.
* Ensure ngraph-bridge link in README goes to right place.
* Make project extras their own subdirectory with index to help organize them.
* **Known Issues**
  + When using TensorFlow\* v1.14.0 with `ngraph-bridge v0.16.0rc0 and CPU backend, we saw notable to severe decreases in throughput in many models.

### 0.22[¶](#id9)

* More ONNX ops
* Optimizations
* Don’t reseed RNG on each use
* Initial doc and API for IntelGPU backend
* DynamicBackend API

### 0.21[¶](#id10)

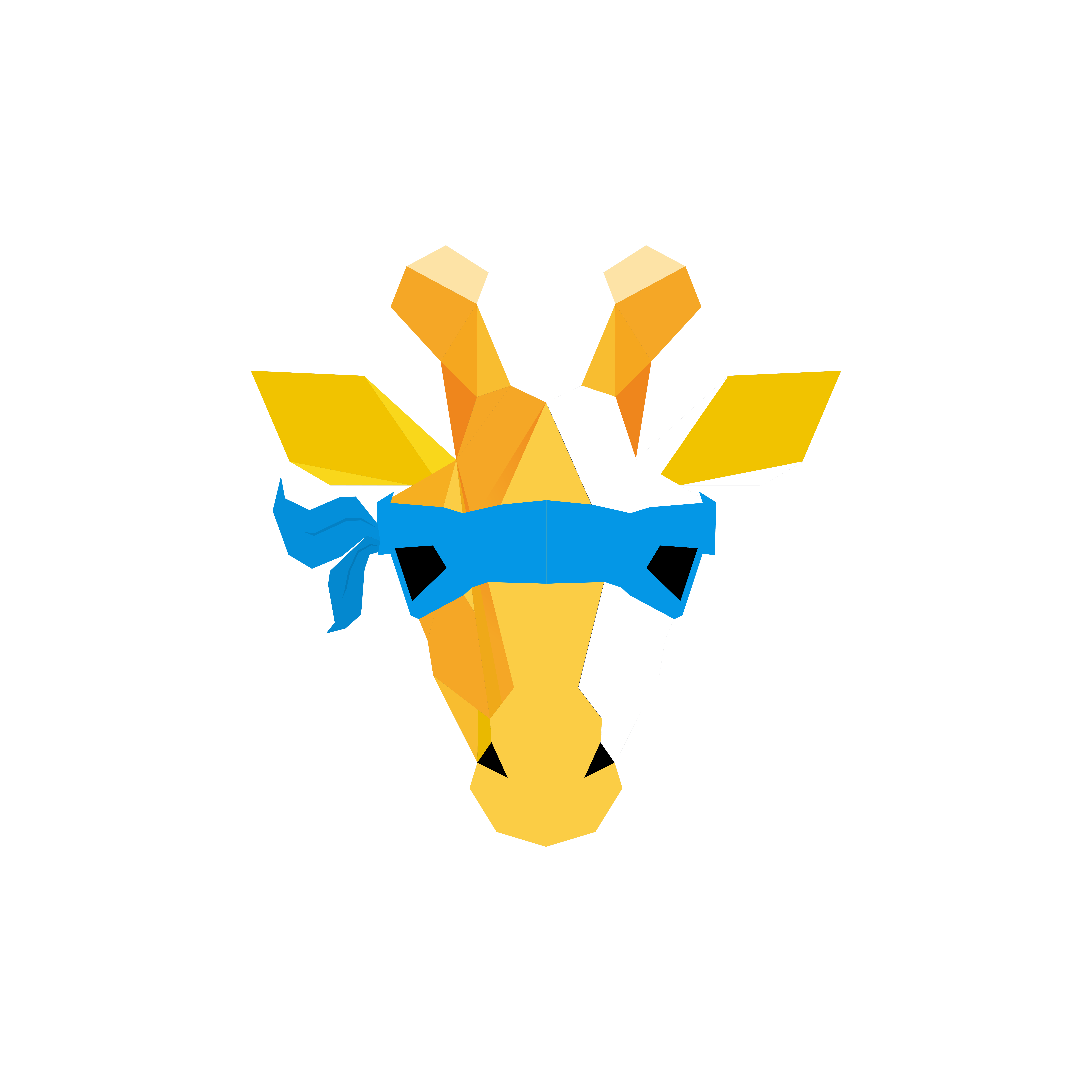
* The offset argument in tensor reads and writes has been removed
* Save/load API
* More ONNX ops
* Better tensor creation
* More shape support
* Provenance improvements
* offset arg for tensor creation is deprecated
* static linking support
* Initial test of 0.21-doc
* Updated doc-contributor-README for new community-based contributions.
* Added instructions on how to test or display the installed nGraph version.
* Added instructions on building nGraph bridge (ngraph-bridge).
* Updated Backend Developer Guides and ToC structure.
* Tested documentation build on Clear Linux OS; it works.
* Fixed a few links and redirs affected by filename changes.
* Some coding adjustments for options to render math symbols, so they can be documented more clearly and without excessive JS (see replacements.txt).
* Consistent filenaming on all BE indexes.
* Removed deprecated TensorAPI.

### 0.20[¶](#id11)

* Save/load API
* More ONNX ops
* Better tensor creation
* More shape support
* Provenance improvements

### pre-0.20[¶](#pre-0-20)

* More dynamic shape preparation
* Distributed interface factored out
* fp16 and bfloat16 types
* codegen execution parameterized by context
* NodeMap, NodeVector, ParameterVector, ResultVector now vectors
  + node\_vector.hpp replaced by node.hpp
  + op/parameter\_vector.hpp replaced by op/parameter.hpp
  + op/result\_vector.hpp replaced by op/result.hpp
* Additional ONNX ops
* Add graph visualization tools to doc
* Update doxygen to be friendlier to frontends
* Python formatting issue
* mkl-dnn work-around
* Event tracing improvements
* Gaussian error function
* Begin tracking framework node names
* ONNX quantization
* More fusions
* Allow negative padding in more places
* Add code generation for some quantized ops
* Preliminary dynamic shape support
* initial distributed ops
* Pad op takes CoordinateDiff instead of Shape pad values to allow for negative padding.
* NodeInput and NodeOutput classes prepare for simplifications of Node
* Test improvements
* Additional quantization ops
* Performance improvements
* Fix memory leak
* Concat optimization
* Doc updates

[](index.odt/_images/nGraph_mask_1-1.png)

The Intel Homomorphic Encryption (HE) transformer for nGraph enables deep learning on encrypted data using homomorphic encryption.

### Distributed training with nGraph[¶](#distributed-training-with-ngraph)

Important

Distributed training is not officially supported as of version 0.29; however, some configuration options have worked for nGraph devices in testing environments.

#### How? (Generic frameworks)[¶](#how-generic-frameworks)

See also: [Distribute training across multiple nGraph backends](index.odt/index.html" \l "document-core/constructing-graphs/distribute-train)

To synchronize gradients across all workers, the essential operation for data parallel training, due to its simplicity and scalability over parameter servers, is allreduce. The AllReduce op is one of the nGraph Library’s core ops. To enable gradient synchronization for a network, we simply inject the AllReduce op into the computation graph, connecting the graph for the autodiff computation and optimizer update (which then becomes part of the nGraph graph). The nGraph Backend will handle the rest.

Data scientists with locally-scalable rack or cloud-based resources will likely find it worthwhile to experiment with different modes or variations of distributed training. Deployments using nGraph Library with supported backends can be configured to train with data parallelism and will soon work with model parallelism. Distributing workloads is increasingly important, as more data and bigger models mean the ability to [Distribute training across multiple nGraph backends](index.odt/index.html" \l "document-core/constructing-graphs/distribute-train) work with larger and larger datasets, or to work with models having many layers that aren’t designed to fit to a single device.

Distributed training with data parallelism splits the data and each worker node has the same model; during each iteration, the gradients are aggregated across all workers with an op that performs “allreduce”, and applied to update the weights.

Using multiple machines helps to scale and speed up deep learning. With large mini-batch training, one could train ResNet-50 with Imagenet-1k data to the *Top 5* classifier in minutes using thousands of CPU nodes. See [arxiv.org/abs/1709.05011](https://arxiv.org/format/1709.05011).

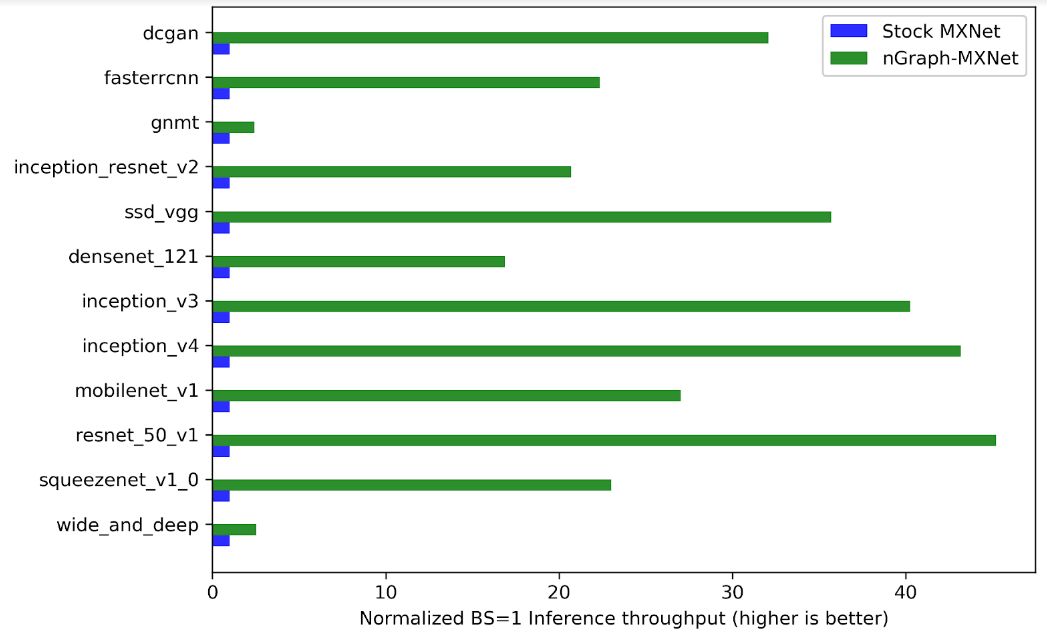
### Testing latency[¶](#testing-latency)

Important

This tutorial was tested using previous versions. While it is not currently or officially supported in the latest nGraph Compiler stack 0.29, some configuration options may still work.

Many open-source DL frameworks provide a layer where experts in data science can make use of optimizations contributed by machine learning engineers. Having a common API benefits both: it simplifies deployment and makes it easier for ML engineers working on advanced deep learning hardware to bring highly-optimized performance to a wide range of models, especially in inference.

One DL framework with advancing efforts on graph optimizations is Apache MXNet\*, where [Intel has contributed efforts showing](https://cwiki.apache.org/confluence/display/MXNET/MXNet+nGraph+integration+using+subgraph+backend+interface) how to work with our nGraph Compiler stack as an [experimental backend](https://github.com/apache/incubator-mxnet/pull/12502). Our approach provides **more opportunities** to start working with different kinds of graph optimizations **than would be available to the MXNet framework alone**, for reasons outlined in our [introduction](http://ngraph.nervanasys.com/docs/latest/project/introduction.html) documentation. Note that the MXNet bridge requires trained models only; it does not support distributed training.

[](index.odt/_images/ngraph-mxnet-models.png)

Up to 45X faster compilation with nGraph backend

#### Tutorial: Testing inference latency of ResNet-50-V2 with MXNet[¶](#tutorial-testing-inference-latency-of-resnet-50-v2-with-mxnet)

This tutorial supports compiling MXNet with nGraph’s CPU backend.

Begin by cloning MXNet from GitHub:

git clone --recursive https://github.com/apache/incubator-mxnet

To compile run:

cd incubator-mxnet

make -j USE\_NGRAPH=1

MXNet’s build system will automatically download, configure, and build the nGraph library, then link it into libmxnet.so. Once this is complete, we recommend building a python3 virtual environment for testing, and then install MXNet to the virtual environment:

python3 -m venv .venv

. .venv/bin/activate

cd python

pip install -e .

cd ../

Now we’re ready to use nGraph to run any model on a CPU backend. Building MXNet with nGraph automatically enabled nGraph on your model scripts, and you shouldn’t need to do anything special. If you run into trouble, you can disable nGraph by setting

MXNET\_SUBGRAPH\_BACKEND=

If you do see trouble, please report it and we’ll address it as soon as possible.

#### Running ResNet-50-V2 Inference[¶](#running-resnet-50-v2-inference)

To show a working example, we’ll demonstrate how MXNet may be used to run ResNet-50 Inference. For ease, we’ll consider the standard MXNet ResNet-50-V2 model from the [gluon model zoo](https://github.com/apache/incubator-mxnet/blob/master/python/mxnet/gluon/model_zoo/vision/resnet.py" \l "L499), and we’ll test with batch\_size=1. Note that the nGraph-MXNet bridge supports static graphs only (dynamic graphs are in the works); so for this example, we begin by converting the gluon model into a static graph. Also note that any model with a saved checkpoint can be considered a “static graph” in nGraph. For this example, we’ll presume that the model is pre-trained.

import mxnet as mx

# Convert gluon model to a static model

from mxnet.gluon.model\_zoo import vision

import time

batch\_shape = (1, 3, 224, 224)

input\_data = mx.nd.zeros(batch\_shape)

resnet\_gluon = vision.resnet50\_v2(pretrained=True)

resnet\_gluon.hybridize()

resnet\_gluon.forward(input\_data)

resnet\_gluon.export('resnet50\_v2')

resnet\_sym, arg\_params, aux\_params = mx.model.load\_checkpoint('resnet50\_v2', 0)

To load the model into nGraph, we simply bind the symbol into an Executor.

model = resnet\_sym.simple\_bind(ctx=mx.cpu(), data=batch\_shape, grad\_req='null')

model.copy\_params\_from(arg\_params, aux\_params)

At binding, the MXNet Subgraph API finds nGraph, determines how to partition the graph, and in the case of Resnet, sends the entire graph to nGraph for compilation. This produces a single call to an [NNVM](https://github.com/dmlc/nnvm) NGraphSubgraphOp embedded with the compiled model. At this point, we can test the model’s performance.

dry\_run = 5

num\_batches = 100

for i in range(dry\_run + num\_batches):

if i == dry\_run:

start\_time = time.time()

outputs = model.forward(data=input\_data, is\_train=False)

for output in outputs:

output.wait\_to\_read()

print("Average Latency = ", (time.time() - start\_time)/num\_batches \* 1000, "ms")

## Indices and tables[¶](#indices-and-tables)

* [Search Page](index.odt/search.html)
* [Index](index.odt/genindex.html)

© Copyright 2017-2020, Intel Corporation.

Intel nGraph Library contains trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. \* Other names and brands may be claimed as the property of others; see [branding notice](http://ngraph.nervanasys.com/docs/latest/branding-notice.html) for more information. Last updated on Feb 05, 2020.

Documentation built with [Sphinx](http://sphinx-doc.org/). Find our code on [GitHub](https://www.github.com/NervanaSystems/).