nomnigraph

nomnigraph is caffe2's graph transformation subsystem

Usage

The output of `caffe2::convertToNNModule(caffe2::NetDef)` (found in `caffe2/opt`) is an `NNModule`. The output of `caffe2::convertToCaffe2Proto(nom::repr::NNModule*, caffe2::NetDef)` is a `NetDef`. `convertToCaffe2Proto(convertToNNModule(n), n)` should basically return an unchanged network.

An `NNModule` is composed of both `dataFlow` and `controlFlow` graphs.

Creating a new operator is straightforward.

```
auto reluNode = nn.dataFlow.createNode(make_unique<nom::repr::Relu>());
```

The line above does a few things worth talking about.

1. It creates a new node using the graph API (both dataFlow and controlFlow are `Graph` s).
2. It instantiates the node with data, specifically a `unique_ptr` to a neural network operator.
3. This `unique_ptr` contains a type that inherits from `NeuralNetOperator` and forms the fundamental representation described in the IR section below.

Inserting this operator into the graph would look something like this:

```
auto edge = nn.dataFlow.createEdge(convOutputTensorNode, reluNode);
```

Some notes here:

1. Again the graph API is used to insert the node into the graph with an edge.
2. Operators are strictly connected to Tensors, not other operators.

IR

nomnigraph has a *parallel* representation that can contain annotations with caffe2's OperatorDef.

If you call `caffe2::convertToNNModule(caffe2::NetDef)`, every operator in the `NNModule` will be annotated with a reference to the original operator in the net.

This means you should not delete the original protobuf.

```
auto conv = repr::nn::get<repr::Conv>(convNode);
if (conv->getAnnotation()) {
  auto annotation = dyn_cast<caffe2::Caffe2Annotation>(conv-
>getMutableAnnotation());
  OperatorDef* op = annotation->getMutableOperatorDef();
  // Do stuff with the caffe2 protobuf
}
```

If you create a new op, as shown in the example above and copied here:

```
auto reluNode = nn.dataFlow.createNode(make_unique<nom::repr::Relu>());
```

it will not have a caffe2 annotation.

How does `caffe2::convertToCaffe2Proto(nom::repr::NNModule*, caffe2::NetDef)` deal with this?

Operators are either generated manually (see the implementation in `caffe2/opt/converter.cc`) or automatically. The automatic generation is done by simply setting the operator `type` to the name of the operator. If you'd like to add your own operator to a net and need it to be generated (i.e. are writing a transform that inserts new nodes which have attributes likes args) you will need to add your own code to `caffe2/opt/converter.cc`.

Do not create `OperatorDef`s in the transformation itself! This is an anti-pattern as the logic becomes less portable.

API

Below is a subset of selected API calls that are quite useful. Lower level manipulation calls are omitted.

Graph transformation API

Nomnigraph provides a ReplaceSubgraph API to perform graph transformation operations without having to write custom subgraph matching logic. The main header file is SubgraphMatcher.h.

ReplaceSubgraph API takes in

- A subgraph pattern to be matched
- A graph to be scanned for matching patterns
- A ReplaceGraph lambda function that takes in a matched subgraph; callers should implement specific graph transformation operation in the lambda.

The ReplaceSubgraph implementation takes care of the pattern matching part and also provides tools for callers to implement graph transformation logic with less effort.

Example usage of the API can be found in subgraph_matcher_test.cc

Example usage of the API for NNGraph can be found in neural_net_test.cc

Graph API

Nomnigraph's core graph APIs provide a generic graph data structure and basic graph manipulation abilities. The main header file is Graph.h.

```
auto g = Graph<T>(); // Constructor

Graph<T>::NodeRef n = g.createNode(T t); // Returns reference to the node

Graph<T>::EdgeRef e = g.createEdge(n1, n2); // Returns reference to the edge

g.deleteNode(n); // Deletes the node and all of its in/out edges from the graph
// Use g.deleteNode(n, false); to keep the edges around.

g.deleteEdge(e); // Deletes the edge between two nodes.
```

```
auto e = g.getEdge(n1, n2); // Gets the first edge that has n1 as a tail and n2 as
the head.

auto ns = g.getMutableNodes(); // Returns a vector of Graph<T>::NodeRef

auto es = g.getMutableEdges(); // Returns a vector of Graph<T>::EdgeRef

T d = n->data(); // Get the data stored at the node
```

NN API

NN (NeuralNet) extends core Graph with functionalities specific to neural network computation graph. The main header file is NeuralNet.h.

Type checking & data accessing

```
repr::NNModule nn = ...;
using namespace nom;

repr::NNGraph::NodeRef n;  // Canonical node of the neural network

bool b = repr::nn::is<repr::Tensor>(n); // Checks the type stored on the node.
(Works with parent types.)

repr::Conv* c = repr::nn::get<repr::Conv>(n); // Returns a pointer to the
NeuralNetOperator or NeuralNetData in the node
```

Iterate through nodes in a NNGraph.

```
auto pairs = dataIterator(nn); // A useful paradigm for iterating through nodes and
corresponding data in no particular order.
auto nodeRefs = nodeIterator(nn); // Iterate through nodes in no particular order.
// See https://github.com/pytorch/pytorch/blob/master/caffe2/opt/mobile.cc#L106-L109
```

These functions make it easy to check attributes on nodes.

```
// -- Tensor node functions --
bool b = hasProducer(tensorNode);  // Checks for producers.
auto n = getProducer(tensorNode); // Returns the producer of the tensor
bool b = hasConsumer(tensorNode); // Checks for consumers.
std::vector<NNGraph::NodeRef> consumers = getConsumers(tensorNode); // Returns a
vector of all consumers of the tensor.

// -- Operator node functions --
bool b = hasInputs(n); // Checks if there are any input tensors.
std::vector<NNGraph::NodeRef> getInputs(n); // Returns a vector of all the input
tensor nodes.
std::vector<NNGraph::NodeRef> getOutputs(n); // Returns a vector of all the output
tensor nodes.
```

These functions are less commonly useful

```
coalesceInsertedDataDependencies(&nn); // Fixes up all the inserted dependencies in the dataflow graph.

insertOp<repr::Relu>(nn.dataFlow, n1, n2); // Inserts an operator into the dataflow graph and creates a new blob to do so.
// n1 or n2 must be a tensor and the inserted blob inherits the name from that, appending an underscore.

convertNode<repr::ConvRelu>(nn.dataFlow, n);  // Converts the data at the node to a new node by calling the passed in type with the old node's data as the constructor argument.
```