Runtime Integration with TensorRT

Need for runtime MXNet-TensorRT integration

TensorRT provides significant acceleration of model inference on NVIDIA GPUs compared to running the full graph in MXNet using unfused GPU operators. In addition to faster fp32 inference, TensorRT optimizes fp16 inference, and is capable of int8 inference (provided the quantization steps are performed). Besides increasing throughput, TensorRT significantly reduces inference latency, especially for small batches. See more here.

Despite its benefits, using pre-trained models with TensorRT typically requires some effort - either re-writing the model using TensorRT's graph building APIs (could be automated using graph-to-graph conversion), or exporting a model to ONNX, followed by an import step. Even if the import is simplified using ONNX, the TensorRT user still needs to provide their own data pipeline, which used to exist in the framework, but no longer does in a stand-alone TensorRT deployment with a client application.

TensorRT is very performant, but does not have the full set of MXNet's operators. While that could be addressed with TensorRT plugins, it's much simpler to reuse already-existing MXNet operators. Also, the user shouldn't care about knowing which operators are supported by TensorRT and which ones aren't - runtime integration allows the graph partitioner to extract subgraphs capable of running inside of TensorRT, place the subgraph in a TensorRT operator in MXNet, execute that operator as part of MXNet's graph execution, and handle non-TensorRT-compatible nodes as regular MXNet operators remaining after the TensorRT subgraph extraction and node substitution. The goal is to accelerate inference without changing user experience.

Design considerations

Since TensorRT can only determine all possible optimizations once the tensor shapes are known, it is imperative that all the shape information be provided. This means that the best time to construct the TensorRT graph is bind time. The PR can selectively apply the TensorRT optimization for inference-only graphs at symbol bind time. This is in fact consistent with the assumptions about TensorRT made on the MXNet Wiki here.

Since as mentioned in #1, TensorRT graph building needs shape information only available at bind time, an important goal was not to disrupt any existing APIs. Even though C++ permits default function arguments, the Python bindings for symbol-related methods (e.g. simple bind) are exposed via a C, not C++, API, wired on the Python side using Ctypes (e.g. see here for the simple bind integration). This precludes the addition of extra arguments without causing breaking changes in the C API. Also, adapting the Python code to such changes wouldn't be enough, since all frontend languages use the C (not C++) API for the FFI. Fortunately, C API changes could be avoided, by simply letting the user enable or disable the TensorRT pass using an environment variable (USE_TENSORRT=1 to enable). This also does not diminish the flexibility of the integration, since the graph pass can read the environment variable each time symbol binding is done, and hence permits turning the graph passes on and off, depending on need. The ability to enable and disable the TensorRT pass at runtime also makes unit testing easier.

TensorRT requires that the workspace size is provided at graph construction time. This value constitutes the upper limit on the amount of memory that TensorRT can use, and does not determine immediate use. Since this amount can be hard for the user to know, its limit should be set to a reasonable value that the user need not concern themselves with. Given that TensorRT integration is applied at bind time and that TensorRT engines wrapped in TensorRT nodes are constructed during the graph pass rather than the memory allocation pass, MXNet will only allocate the amount needed for the nodes remaining after the TensorRT subgraphs have been extracted. This means that no memory will be doubly allocated - first for the complete MXNet subgraph and then for TensorRT. However, the question remains whether the memory used per TensorRT engine should be a configurable parameter, either as a method argument or an environment variable, or whether TensorRT should be able to use the maximum available GPU memory and then reserve only what it needs. I would like to suggest the latter. Since the TensorRT subgraph will typically use less memory than the same subgraph in MXNet (due to more layer fusion), it's extremely unlikely that a model which runs purely as an MXNet graph would fail with an out of memory error when parts or most of the graph run inside TensorRT. Fewer knobs (in this case, not giving the user the ability to tweak the maximum amount of memory availble to TensorRT) would simplify use.

TensorRT can accept graphs constructed using two main approaches: (a) via the TensorRT graph API, (b) using ONNX. Approach (a) seems simple on the surface - one traverses the NNVM graph, finds subgraphs that TensorRT can execute, converts the subgraphs to TensorRT graphs, and substitutes the subgraphs with TensorRT nodes, each of which contain the TensorRT engine corresponding to the subgraph. However, the approach taken by NVIDIA was to use ONNX as tha IR. The reason for this is twofold. First, ONNX is a very well-known IR, which is supported by the entire deep learning software community. This ensures that the design of the IR gets as much feedback as possible as to whether the IR is feature complete, and what the semantics are. NVIDIA already maintains an ONNX-to-TensorRT converter (link), and will continue to do so. Whatever changes that may apply to the TensorRT APIs or the internal features may be nicely hidden behind the well-established ONNX IR. Second, ONNX is growing beyond being merely an IR. As it becomes more of a standard, its adoption will be associated with other benefits, such as the ability to verify standard compliance.

Despite the advantages of using the ONNX route described in #4, there are some costs. The main one is the dependency on Protobuf. This is a valid criticism on the surface, however, since the TensorRT integration requires an opt-in during build time, adding one more dependency is not a problem if it is not a mandatory dependency. Moreover, the same Protobuf dependency already exists for the MXNet ONNX importer, which is now part of the MXNet source tree (link), rather than being located in a separate repository. Just like the use of the ONNX importer is optional and requires ONNX (and hence also Protobuf), the TensorRT build is optional. Finally, Protobuf is already in use for MXNet multi-node for serializing messages, so if the same version of Protobuf is kept, this shouldn't pose problems.

The optional integration of TensorRT will be chosen using a config.mk flag (USE_TENSORRT=1), which will function similarly to other flags, such as USE_CUDA, USE_CUDNN, etc. Needless to say, USE_TENSORRT will depend on CUDA and cuDNN. Setting USE_TENSORRT=1 will enable -DMXNet_USE_TENSORRT=1 to affect code guards in the C++ source, just like USE_CUDNN=1 in config.mk enables -DMXNet_USE_CUDNN=1 to affect the relevant source guards.

In order to simplify evaluation of the TensorRT build, usability and to run unit tests, the PR will come with a Dockerfile, which will be used for CI, but can be used as an example of how to build the integration on bare metal as well.

APIs / user experience

There is no change in the inference APIs, except for the need to set the MXNet_USE_TENSORRT environment variable to 1. For example, in Python, we can simply do:

```
os.environ["MXNet_USE_TENSORRT"] = “1”
```

(env var values are of type string, otherwise the os.environ API will result in Python complaining).

Note that for backward compatibility, if the environment variable is not set, it will default to 0. Also, unlike some other environment variables that are only checked during MXNet initialization, this one gets checked every time graph binding happens. This typically happens only once during the inference application’s life cycle, but since one can re-bind a symbol to say compare a TensorRT and a non-TensorRT run, the check will happen during each bind /re-bind to enable that. Since the TensorRT graph pass is enabled using an environment variable, no break in the C++, C or any frontend language API is needed.

Note that there is one more change required - in calling simple bind. This doesn’t change the simple bind API, but how it’s called relative to the “usual” case, by using some of the arguments which are optional. This has to do with the shared_buffer parameter. Before explaining how the call changes, let’s consider why it’s necessary:

1. The TensorRT graph needs to be constructed during the simple bind call, but before memory gets allocated for the non-TensorRT part of the graph.
2. TensorRT needs the weights, not just the shapes, to be provided before the engine is constructed - it will store them inside the ICudaEngine object. The engine will then be serialized inside the NNVM TensorRT op, and deserialized when the graph executor takes over. This means that the weights need to be provided to the simple bind call to construct the TensorRT engine.
3. The way to provide the weights is to hand them over to the simple bind call via the “shared buffer” argument.

The shared buffer weights can be provided during the bind call and can be freed by the frontend language once binding is complete (e.g. by exiting the relevant scope in Python, or calling del).

Since we need both arg_params (weights) and aux_params (e.g. BatchNorm moments), we need to merge arg_params and aux_params into one dictionary. Here’s a Python example:

```
def merge_dicts(*dict_args):
   """Merge several dictionaries into one"""
   result = {}
   for dictionary in dict_args:
       result.update(dictionary)
   return result
```

`Now let’s see a use example:`

```
device = mx.gpu(0)
sym, arg_params, aux_params =
   mx.model.load_checkpoint(model_name, num_epochs)
executor = sym.simple_bind(ctx=device,
   data=data_shape,
   softmax_label=(batch_size,),
   shared_buffer=merge_dicts(arg_params, aux_params),,
   grad_req='null',
   force_rebind=True)
```

`Now we can simply update data in the executor’s arg dict and run the forward pass:`

```
executor.arg_dict["data"][:] = my_data_batch
executor.forward(is_train=False)
predictions = executor.outputs[0].asnumpy()
```

Limitations and future work

Since the new accelerator API proposal (link) was only published a few days ago and the implementation is still on an MXNet fork, the current TensorRT integration doesn't use that API yet, but could be refactored in a future commit to use it. There is nothing in the current design that would prevent making use of that API in the near future.

Building the TensorRT engine takes a non-trivial amount of time, because the compiler evaluates performance and the hardware on the system before creating the fused layers on demand, and then needs to actually compile them. For ResNet-50 this may be a few seconds, but larger models also exist which may take longer. TensorRT comes with the ability to serialize the TensorRT engine for a particular hardware platform. This is called the serialization of a TensorRT plan, which is the engine along with the ahead-of-time-compiled fused kernels for a given GPU. The first PR of the TensorRT integration will not provide for TensorRT plan caching, so using TensorRT might have a small start-up cost, but for long-running inference processes, this shouldn't be a problem. Caching the TensorRT plan will be addressed in a future commit.

As mentioned before, the reproducibility of the build will be demonstrated using a Dockerfile that will provide an easy way to evaluate the build. This will be a CI Dockerfile, which could be re-used as an example of building on bare metal, or for building non-CI Docker images. The Docker recipe was tested on Linux on x86_64, but not other platforms supported by TensorRT (Linux on 64-bit ARM (aarch64), Android on aarch64, QNX on aarch64). Supporting other platforms, e.g. Linux on aarch64 (e.g. L4T, i.e. Linux for Tegra, on the NVIDIA Jetson platform) is left for subsequent commits.

The current commit supports many, but not all, of TensorRT operators. For example, this integration can run CNNs such as VGG, or ResNet, but not necessarily everything that TensorRT can support. More operators will be covered in future commits.

TensorRT supports plugins, which can be integrated into the graph pass. However, this was not a priority since the runtime TensorRT integration can always fall back to existing MXNet operators. Supporting plugins is possible, but will be added in future commits.

The upcoming PR will support fp16 and fp32, but not int8. Since int8 support in MXNet is itself very new, figuring out calibration and other details is left for a future commit.

TensorRT 4 has a new feature called BYOM (bring your own memory). This means that instead of telling TensorRT how much memory it can use, the data /scratch space tensors can be provided by MXNet, and can be re-used by MXNet when not running the forward pass. The memory in permanent use will then be limited to TensorRT storing weights. Support for this feature will be added in a future commit.

Subgraph partitioning

The conditions for a subgraph to be valid for TensorRT are:

- all the ops are TensorRT compatible (defined by IsTRTCompatible function in src/executor/tensorrt_pass.cc)

- there should be no cycle created by replacing the subgraph by a TensorRT node

A cycle would imply that there is a path from an output of the subgraph to an input of the subgraph. While respecting those conditions we want to maximize the size of this subgraph.

We first generate a list of pairs of groups of nodes that are incompatible (if the subgraph contains one node of one of the groups (e.g. TensorRT compatible) then it can't contain any node from the other group (non-TensorRT compatible)). This is done by running BFS and reverse BFS starting from all the non-TensorRT compatible nodes.

blocked URL

Figure: Black nodes are not TensorRT compatible. Focusing on node 4: we generate two groups (G1 / G2) by running reverse-BFS / BFS from this node. The properties of those two groups is that if one node of the subgraph is in one of the two groups then you can't add any node from the other group (it will create a cycle that can't be broken by adding more nodes because this non-TensorRT compatible node will stay in between).

We generate a list of pair of groups (one pair for each TensorRT-incompatible node)

We add subgraphs one by one, using BFS:

- start from a node which is not part of a subgraph, instantiate its list of incompatible nodes

- grow the subgraph in DFS order (on both inputs and outputs with priority to inputs), to add a node to the subgraph it has to be:

    1) TensorRT compatible

    2) not in another subgraph

    3) not in the list of current subgraph's incompatible nodes

- Once (and only if) a node is added to the subgraph we:

    1) Update the incompatibility list with new incompatible nodes

2) Add its inputs / outputs to the queue (for future visit)