Dynamic CustomOp Support

Link to dev List discussion

TBD

Feature Shepherd

TBD

Problem

Previously MXNet only supported Custom operators written in higher level langaugues (ie. Python, Java/Scala, etc.) via the Custom Op interface: https://mxnet.incubator.apache.org/versions/master/tutorials/gluon/customop.html?highlight=customop. This makes it complicated to add high performance routines written in C++ and CUDA. One solution was the MobulaOp project: https://github.com/wkcn/MobulaOP which enabled a seamless experience for loading these high performance C++ and CUDA routines built on-top of the Custom Op interface. This project was very successful and we propose to integrate the concepts and design directly into MXNet in this project. But in this project we will implement a CustomOp and dynamic library loader into the MXNet engine, enabling custom high performance ops to be leveraged from all language bindings and without the overhead of the engine using callbacks at runtime.

UserExperience

Similar to the ideas presented in the Bring Your Own Accelerator proposal and the current user experience that MobulaOp provides, we want to provide a similar user experience where its easy to load operator libraries dynamically at runtime. Similarly, one benefit to writing custom ops is that you do not need to recompile MXNet. So we want to provide an easy to use build flow to compile custom operators into libraries without a TON of external dependencies.

However, we will aim to balance between "simplified build/limited dependencies" and "ease of writing custom operators". For example, many custom operators may need to execute basic tensor operations like addition, dot, etc. and it would be redundant and complicated for custom op authors to have to rewrite these core routines.

Lastly, we want custom operators to be first-class operators and have access to all the capabilities that internal MXNet operators do. One example is enabling custom operators to leverage the MXNet resource manager for storage and memory.

Approach

Compiling Custom Operators

*To support compiling custom operators, we need to construct a simple API/file-set that users will compile their custom operators with. The result of this compilation will be a dynamic library (Linux: *.so, Windows: *.dll). We will need to provide unit tests that allow users to test their operator registration outside of MXNet to ease debugging.*

Just how operators are registered in MXNet with NNVM, we propose a similar lightweight approach that doesnt require compiling custom operators with NNVM.

Dynamically Loading Operator Libraries

After a user compiles custom operator(s) into a library, we need to construct an API to

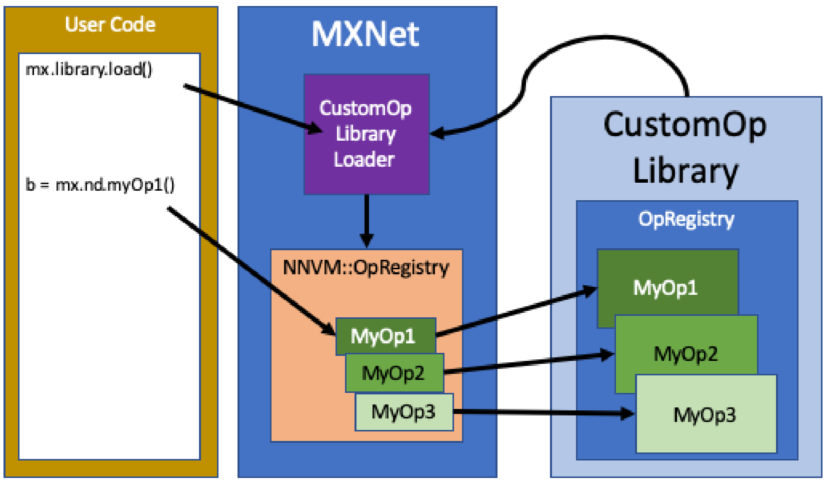
- load user-specified libraries
- register operators from each library in MXNet so that they can be called/executed

Calling Custom Operators

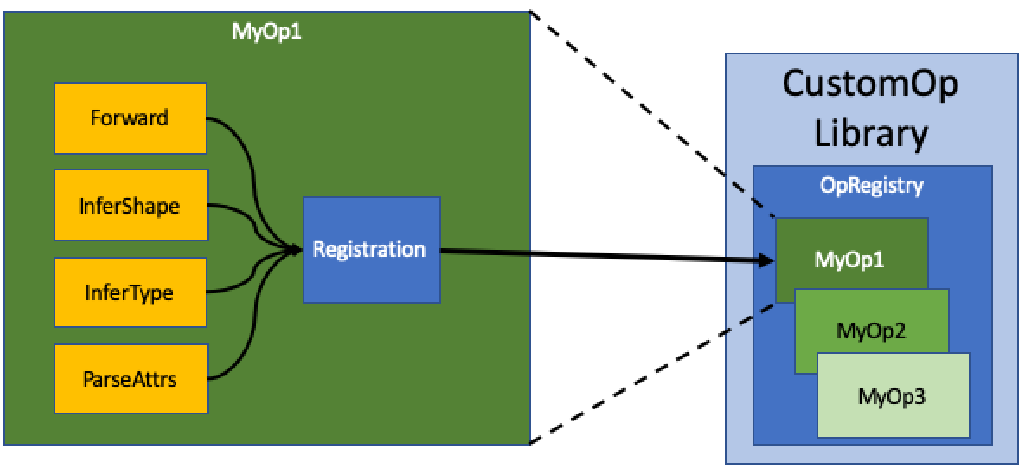
After a library is loaded, users need to call their operators from their application. We'll register custom operators in the same ndarray and symbol namespaces that regular operators use here to provide a similar user experience.

Architecture

The figure below shows the high-level architecture. The user will call the mx.library.load *API to load their custom operator library. This will result in the operators being discovered from the so/dll and re-registered into MXNet's NNVM registry. Then the user will call their operator directly just like they would for any regular MXNet operator.*



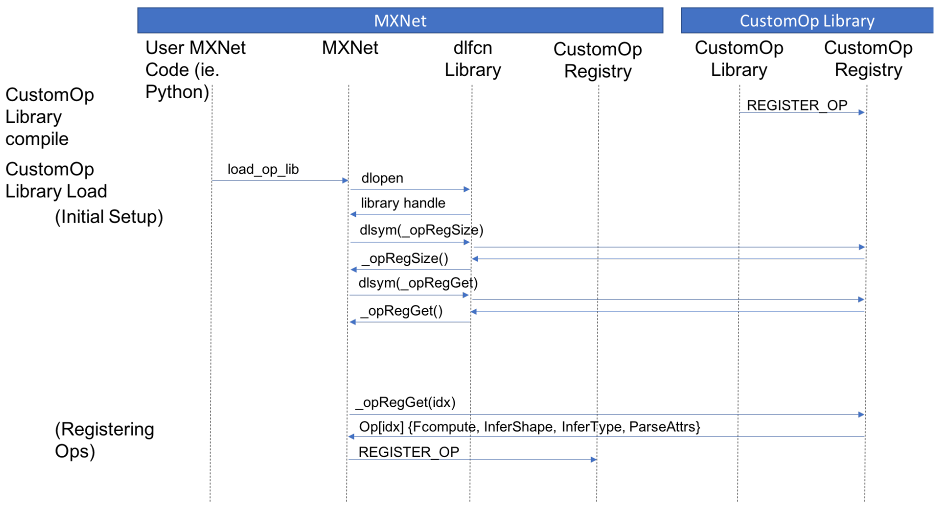
When building a customOp Library, users will write 4 functions for each operator: `Forward`*,* `InferShape`*,* `InferType`*, and* `ParseAttrs`*. These are similar to the standard functions required for current Backend C/C++/CUDA operators in MXNet. Similarly, they will register their op (ie. the 4 functions) in the library. As shown above, this "local-registration" will be parsed by MXNet when loading the customOp library at runtime.*



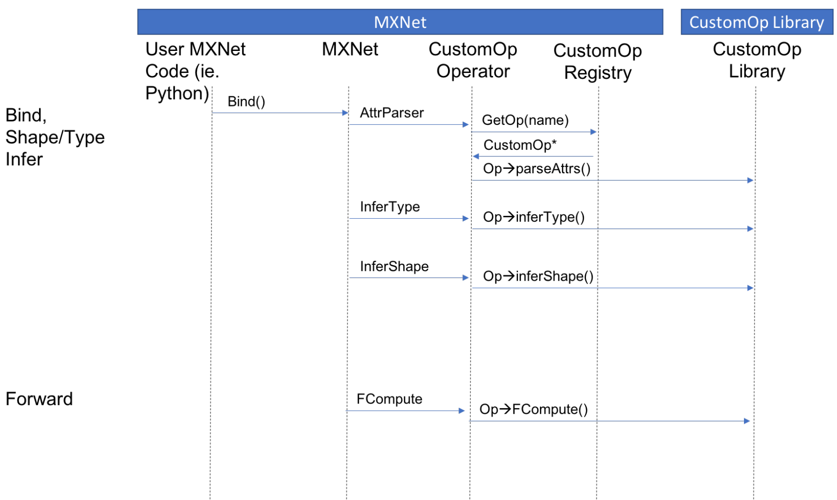
Runtime Behavior

Heres the overall runtime behavior for CustomOps. Its it is broken down into 2 parts: initial library load, and operator execution.

First, the user writes their custom op functions: Forward, InferShape, InferType, and ParseAttrs. Then they statically register the functions in their library with `REGISTER_OP`. Next they compile and produce a shared library (so/dll). Then they start MXNet, and load their library. During the initial setup, the user calls *mx.library.load* in their code to load their shared library. During the loading process, MXNet parses all of the operators that have been registered by getting the number of ops registered with the `_opRegSize` function. Then it iteratively gets each op by calling the `_opRegGet` and analyzes it before re-registering it inside MXNet's NNVM registry.



Later when a CustomOp operator is bound/executed the functions from the shared library are executed. During the bind step, the attributes for the operator are analyzed by the customOp's `parseAttrs` function in the shared library. For type and shape inference, the respective functions are also called through the `inferType` and `inferShape` APIs. Lastly, when executing the forward pass, the `Forward` function is called for the operator from the shared library.



New MXNet APIs

These are new APIs that are added to MXNet

C APIs

- MXLoadLib - API to load libraries
  - Checks version number
  - Calls initialize on the library
  - Check that each operator defines required functions
  - Register each operator found

Python APIs

- `load` - API to load libraries
  - Takes a path to the library
  - checks if the path exists and if points to file
  - calls C API `MXLoadLib` to perform actual loading

APIs for implementing Custom Operators

These are the new APIs that users will implement for their custom operators

- `parseAttrs` - takes a set of key/value pairs for attributes and gives users an opportunity to validate the attributes passed to their custom operator.
  - ```
    int parseAttrs(std::map<std::string, std::string> attrs,
                   int* num_in,
                   int* num_out);
    ```
  - Inputs: the map of attributes passed to the operator from the user
  - Outputs: num_in, num_out - the number of input/output arrays required for this operator
  - returns 1 if success, or zero if failure
- `inferType` - performs type inference for this operator
  - ```
    int inferType(std::map<std::string, std::string> attrs,
                  std::vector<int> &intypes,
                  std::vector<int> &outtypes);
    ```
  - Inputs: the map of attributes
  - Inputs/Outputs: intypes, outtypes - the list of input/output types that should be inferred. Values of of -1 should be defined by this operator as a specific type
  - returns 1 if success, or zero if failure
- `inferShape` - performs shape inference for this operator
  - ```
    int inferShape(std::map<std::string, std::string> attrs,
                   std::vector<std::vector<unsigned int>> &inshapes,
                   std::vector<std::vector<unsigned int>> &outshapes);
    ```
  - Inputs: the map of attributes
  - Inputs: inshapes - the shapes of the input arrays
  - Outputs: outshapes - the shapes of output arrays
- `forward` - performs forward pass of this operator
  - ```
    int forward(std::map<std::string, std::string> attrs,
                   std::vector<MXTensor> inputs,
                   std::vector<MXTensor> outputs,
                   OpResource res);
    ```
  - Inputs: the map of attributes
  - Input data: inputs, input tensors
  - Output data: outputs, output tensors

API for registering operator and its functions:

- `REGISTER_OP` - registers the operator in the library
  - ```
    REGISTER_OP(sam)
    .setForward(myFCompute)
    .setParseAttrs(parseAttrs)
    .setInferType(inferType)
    .setInferShape(inferShape);
    ```
  - `REGISTER_OP` - macro that defines an custom operator object with given name
  - `setForward` - sets the FCompute function
  - `setParseAttrs` - sets the parse attributes function
  - `setInferType` - sets the infer types function
  - `setInferShape` - sets the infer shapes function

Example Custom Operators

Examples of creating custom operators, building them into a library, and loading them at runtime to test them can be found here:

https://github.com/apache/incubator-mxnet/tree/master/example/extensions/lib_custom_op

The GEMM example contains two operators. The state-less operator shows a regular operator here:

https://github.com/apache/incubator-mxnet/blob/master/example/extensions/lib_custom_op/gemm_lib.cc#L169-L174

The example GEMM stateful operator is here:

https://github.com/apache/incubator-mxnet/blob/master/example/extensions/lib_custom_op/gemm_lib.cc#L220-L225

The example build command to build the GEMM operators into a library is here:

https://github.com/apache/incubator-mxnet/blob/master/example/extensions/lib_custom_op/Makefile#L21
    ```

The example python code to load the library and test the operator for both symbol and ndarray APIs is here:

https://github.com/apache/incubator-mxnet/blob/master/example/extensions/lib_custom_op/test_gemm.py