Bring your own Accelerator

- Link to dev List discussion
- Feature Shepherd
- Problem
- Proposed Approach

User experience for backend library creators:
User experience for ML/DL scientists:

**Loading Accelerator Libraries**
- User-specified
- Bundled
- Environment Variable
- Accelerator APIs
- Future Proofing APIs
- Other API concerns
- Backward compatibility
- Performance Considerations
- Test Plan
- Implementation plan
- Alternative Approaches
- Technical Challenges
- Milestones
- References

**Link to dev List discussion**
https://lists.apache.org/thread.html/464712f0136fb51916ca9f1b702b99847e108d9bd0b6a2b73fc91f1@%3Cdev.mxnet.apache.org%3E

**Feature Shepherd**
Need volunteer to help shepherd

**Problem**

MXNet is a high performance machine learning framework and leverages many high performance tools and libraries in the backend such as MKLDNN, cuDNN, TensorRT, and nGraph among others. Some recent backend additions to MXNet are: TensorRT (subgraph), and Elastic Inference. Adding each of these backends required modifying the MXNet source code, deep knowledge of how MXNet works, and months of time to work with the community to add custom processor-specific changes.

However, adding support for these backends does not change MXNet, and should not require community approval to run MXNet on a new processor. This proposal adds APIs to enable MXNet to run anywhere, on any custom chip or backend library without requiring the backend code to be committed to MXNet and forcing the developers to open-source their custom architecture-specific code/routines unnecessarily.

**Proposed Approach**

“Bring your own Accelerator” is a set of Accelerator APIs that allow MXNet to interface with any custom ML accelerator chip or ML library. It will bring a new differentiator to MXNet that other ML frameworks lack.

The main problem with adding new backends to MXNet is adding the new functionality to the MXNet code base, recompiling MXNet, and upstreaming the changes (requiring community support/approval). The library approach we present will enable new backends to be compiled separately from the MXNet code base and will not require linking against all of MXNet's 3rd party dependencies (ie. TVM, NNVM, etc.). A single header file `mxnet_acc.h` will be used to define the APIs between MXNet and accelerator libraries.

The accelerator library will be loaded via `dlopen` dynamically in the MXNet backend and the APIs will be located in the library using `dlsym` (standard `posix` functions from `dlfcn.h`). Similar functions exist in Windows (LoadLibrary and GetProcAddress). We will use C types/structs to eliminate the compiler version/compatibility issue. This eliminates the requirement for new backends to be compiled or linked against MXNet, or even using the same compiler.

In terms of operator coverage, we cannot expect that an accelerator supports every operator that MXNet has. Instead we will follow the same subgraphing/partitioning scheme that MXNet already supports where the CPU context will be used for any operators not supported by the accelerator.

In this project, we will create a set of abstractions through an API that allows accelerator vendors to create external libraries to interface their custom hardware to MXNet without modifying the MXNet code base. We’ll streamline how MXNet interacts with processors, and create a user-facing API to dynamically load accelerator libraries at runtime. This will allow accelerator vendors to distribute their library separately from MXNet, decoupling the release of MXNet versions from accelerator library versions.

**User experience for backend library creators:**

There are two ways that ML chips/libraries can be implemented:

- As a library with a set of APIs to execute individual operators (ie. cuDNN, MKLDNN). We'll call this the imperative execution mode.
As a library that pre-processes the whole graph first and then executes it via a LoadModel/Infer type of API (ie. TensorRT, nGraph, TVM/Neo, EIA). We’ll call this the symbolic execution mode.

In this proposal, we will focus on the symbolic mode.

User experience for ML/DL scientists:

We expect users (data scientists) to treat accelerators like any other context as they would normally in MXNet. The only things they need to be aware of are:

- the “mx.load_acc()” API to load an accelerator library dynamically at runtime. Users specify the path to the library to load, and an optional accelerator name to use and override the name provided by the library via the getAccName API. def load_acc(path, acc_name=None)
- accelerator contexts are added to the mx module after loading, so that users can easily call “mx.acc()”

Below is an example code snippet for using the Accelerator APIs:

```python
import mxnet as mx

# load accelerator library, returns a context with device id 0
ctx = mx.load_acc("/path/to/libmyacc.so")

# after loading library, accel context can also be created by
ctx = mx.acc()
ctx = mx.acc(0)

# can also list the available accelerators just like
# mx.test_utils.list_gpus(), returns [0, 1, ...]
ctx_list = []
acc_list = mx.test_utils.list_acc(mx.acc())
for i in acc_list:
    ctx_list.append(mx.acc(i))

# bind model
sym, arg_params, aux_params = mx.model.load_checkpoint(NAME, EPOCH)
mod = mx.mod.Module(symbol=sym, context=ctx)
mod.bind(data_shapes=[('data', (1,3,224,224))], label_shapes=mod._label_shapes)
mod.set_params(arg_params, aux_params, allow_missing=True)

# forward pass
mx_img = mx.nd.array(IMG, ctx=ctx)
Batch = namedtuple('Batch', ['data'])
data = Batch([mx_img])
model.forward(data)
```

Loading Accelerator Libraries

We will provide users the simplest and most familiar ways to use accelerator libraries.

User-specified

Users can custom load accelerator libraries through the load_acc API specifying the path. This will enable users to write some code quickly and try things out without too much setup or configuration.

Bundled

MXNet can bundle libraries in with its installation (pip, jar, etc) and can find those libraries during the init process (ie. import mxnet). This will create a better user experience that “just works” for specific use-cases like EIA.

Environment Variable

Users can point to a directory of accelerator libraries by setting the MXNET_ACC_LIBRARIES variable. This will make it easier for users to generalize their MXNet code by removing environment-specific paths. This variable will be checked during MXNet’s initialization process.

Accelerator APIs

The main APIs that will be defined in mxnet_acc.h are categorized and described below. These APIs use only C (no C++) to avoid potential problems with using different compilers/STL.

- **Accelerator Identification**
  - `GetAccName` - returns 3 letter name for the accelerator (ie. returns “eia” for mx.eia() context)
  - `void getAccName(char *name);`
  - `GetNumAcc` - returns number of accelerators in the system
  - `int getNumAcc();`
**Future Proofing APIs**

We are future proofing accelerator library APIs by providing generic interfaces to interact with the accelerator library. The `configure` function takes a set of keyword args (inputs) and returns a set of keyword args (outputs). This API can be called multiple times with different behavior each time depending on the inputs, and so it can represent any set of additional APIs that an accelerator might need.

- **Generic accelerator configuration**
  - pass any keyword/arg mapping into an accelerator, and potentially get some outputs. return status/error
  - int configure(const char *in_keys[], char *in_vals[], int num_in, char *out_keys[], char *out_vals[], int *num_out);
  - called by the user via configure function on an mxnet accelerator context in any language binding (python shown in the example below):
    - ctx = mx.acc()
    - status = ctx.configure(init=str(True), other="thing")
    - status = ctx.configure(call="getStatus")
    - drink = ctx.configure(call="getMeACoffee")

**Other API concerns**

Some accelerators perform special handling of the weights/params to optimize execution by placing in special on-chip/high-speed memories. In the `LoadModel` API, we need to clearly identify which MXTensors are weights/params and which are input data (ie. image, text, etc. to the model).

**Backward compatibility**

No issues, this is a new functionality. Existing custom hardware backends for MKL/MKL-DNN/CUDNN/TensorRT will continue working.

**Performance Considerations**

We will performance analyze the overheads introduced by using a dynamically loaded library by creating a test accelerator library that simply reuses the existing CPU and GPU operator implementations. Then we'll compare these "accelerators" against the current CPU and GPU contexts.

**Test Plan**

We will create a test accelerator library that simply reuses the existing CPU and GPU operator implementations an run all existing unit tests.

**Implementation plan**

- **Initialize** - MXNet calls this function when library is loaded, passes MXNet version to the library. This is the opportunity for the library to return an error if it cannot be used with a specific version of MXNet.
  - int initialize(int version);

- **NDArray format**
  - We need a format to standardize on, that does not require any other submodule dependencies between MXNet and the accelerator library similar to how EIA is implemented already.
  - We'll leverage the already-implemented and open-source DLPack/DLTensor: [https://github.com/dmlc/dlpack/blob/master/include/dlpack/dlpack.h](https://github.com/dmlc/dlpack/blob/master/include/dlpack/dlpack.h)

- **Symbolic Execution**
  - `SupportedOps` - pass in string json of graph, returns list of IDs of nodes/ops that can run on accelerator. This API must be called after MXNet does shape/dtype propagation in order to provide the data types/sizes for each operator. Some accelerators will only support certain operators with certain data size limits, or only for certain data types, so this info is needed to determine if an accelerator can support a particular op.
    - void supportedOps(const char *json, const char *data_names[], const DLTensor *data, const int num_data, int *ids);
  - `LoadModel` - Pass in an ID, a string json of graph, map of input data names mapped to tensors data. This json graph is probably not the same graph passed to `supportedOps` above, since MXNet will perform graph partitioning based on the supported ops of the accelerator. dev_id is the ID of the accelerator in the system. Will also identify which inputs are weights/params versus input data.
    - int loadModel(const char *model_id, const char *json, const char *data_names[], const DLTensor *data, const int num_data, int dev_id);
  - `UnloadModel` - Pass in an ID for a model loaded with `LoadModel`, tells accelerator library to free up any memory used for previously loaded model.
    - void unloadModel(const char *model_id);
  - `Infer` - pass in an ID for model loaded with `LoadModel`, map of input data names mapped to tensor data for data thats changed. Returns map of data names to output tensor data. This is a blocking call.
    - int infer(const char *model_id, const char *in_names[], const char *out_names[], const DLTensor *in_data, DLTensor *out_data, int num_in, int num_out);
1. Implement a PR with basic symbolic flow: supported ops, load/unload model, infer
   Link to WIP PR: https://github.com/apache/incubator-mxnet/pull/15489
2. Implement a followup PR with imperative accelerator flow (fcompute, storage, copy, etc)

Alternative Approaches
Currently, custom accelerators like TensorRT must be implemented by modifying the MXNet backend and learning how MXNet works at the lowest level. The team that implemented TensorRT support in MXNet ran through many hurdles and the learnings from that effort are being applied in this proposal.

Technical Challenges
We'll need to version the MXNet operators with accelerator libraries so that as operator implementations change we catch the mismatch against older accelerator libraries.

Milestones
TBD

References
- Subgraph Project description within MXNet
  https://cwiki.apache.org/confluence/display/MXNET/Unified+integration+with+external+backend+libraries
- Related project using subgraph API for MKLDNN
  https://cwiki.apache.org/confluence/display/MXNET/MXNet+Graph+Optimization+and+Quantization+based+on+subgraph+and+MKL-DNN
- TensorRT runtime integration proposal
  https://cwiki.apache.org/confluence/display/MXNET/Runtime+Integration+with+TensorRT
- TensorRT Graph pass to produce TRT-specific subgraphs
  https://github.com/apache/incubator-mxnet/blob/5a9c3af4b101b85047b306575575ea9022a8474e/src/operator/contrib/tensorrt.cc
- TensorRT Operator registration inside MXNet
  https://github.com/apache/incubator-mxnet/blob/5a9c3af4b101b85047b306575575ea9022a8474e/src/operator/contrib/tensorrt.cc
  (Also see other tensorrt.* files in the contrib directory)
- ONNX to TensorRT conversion
  https://github.com/apache/incubator-mxnet/blob/e04565aa9534bdc35bdca57e67ef2307cb473792/src/operator/executor/onnx_to_tensorrt.cc
- TensorRT execution at runtime
  https://github.com/apache/incubator-mxnet/blob/2bece7641fbd4b72425fe6b1ded900ca19d3a8/src/c_api/c_api_executor.cc#L446
  https://github.com/apache/incubator-mxnet/blob/5a9c3af4b101b85047b306575575ea9022a8474e/src/operator/trt_graph_executor.cc
- TensorRT Unit Tests
  https://github.com/apache/incubator-mxnet/tree/5a9c3af4b101b85047b306575575ea9022a8474e/tests/python/tensorrt
- TensorRT Tutorial
  https://github.com/apache/incubator-mxnet/blob/cb6815c3963e1a97a1aa51fe5d45390d9fddfb/docs/tutorials/tensorrt/inference_with_trt.md