Project website: https://simeng96.github.io/PyTorch_Extension/

1. Summary

We implemented the High Order Convolution Module based on PyTorch and optimized its performance using PyTorch's C++ and CUDA extension. We experimented our implementations with NVIDIA Tesla K80. Our implementation with PyTorch's C++ extension is about 2x faster for forward and more than 10x faster for backward compared to basic Python implementation on GPU. And our implementation with PyTorch's CUDA extension is more than 20x faster for forward and 100x faster for backward compared to basic Python implementation on GPU.

2. Background

We implemented a 2D High Order Convolution Module (the 2D version of the 1D high order module specified in paper [1]) based on Pytorch. A High Order Convolution Module is usually used for image or video related tasks in Deep Neural Networks and it can sometimes perform a more effective way to extract information from images or videos than a Convolution Module.

To explain how the 2D High Order Convolution Module works, we use B for batch size (usually be a power of 2, ranges from 8 to 512), C for channel (3 in this project), H for height (224 in this project), W for weights (224 in this project), k_size for (weight tensor's size). The steps of the calculation is as follows:

(1) Take an input tensor X of shape (B, C, H, W), and go through a Conv2D layer to get a weight tensor W of shape (k_size * k_size, B, 1, H, W);

(2) Use the weight tensor W as a Convolution Layer to sweep through the input tensor X, the weights (k_size * k_size values) for the receptive field around X[b][c][h][w] is W[:][b][0][h][w]. Then return the output tensor out with shape (B, C, H, W), which is the same as the shape of the input tensor X.

To maintain the shape on dimensions 2 and 3, we uses zero padding in both (1) and (2), which also makes the module more challenging to implement in C++ and CUDA. Also note that the weight tensor's dimension 2 has size 1. This is a trick in deep learning used to reduce the calculation by sharing the same set of weights across all the channels. The dimension 2 is left to 1 for 2 purposes: (1) fits better to tensor operation APIs, (2) make it more extendable to non-channel-sharing implementation.

As a Deep Learning framework, PyTorch has implemented lots of Modules which runs efficiently in nn.Module, such as Conv2D. However, when running the High Order Convolution Module as a whole, the efficiency is extremely slow. A typical Deep Neural Network with High Order Convolution Module takes approximately half of the time running the High Order Convolution Module, and the other parts of the network takes another half of the time.

The above problem could probably related to some inefficiency of PyTorch's own mechanism, which gives us much motivation to build the algorithm on our own. The detailed analysis and our approach to improve the bottleneck is described in the next section.

3. Approaches

3.1 Bottleneck Analysis using Python extension

To figure out the bottleneck of the above low-efficiency problem, we implemented the Python version of the High Order Convolution Module using Pytorch's nn.Module class. By utilizing this class, we only need to implement the forward pass of the module, and the backward pass is implemented automatically by the framework. For this reason, we use this version as ground truth to check the correctness of C++ and CUDA implementation.

For step (1) mentioned in the previous section, we use PyTorch's built-in Conv2D module. What we implemented by ourselves is step (2). The algorithm is pretty straight forward:

- Zero Pad input X to get a padded X_p of shape (B, C, H+4, W+4)
- For each position i of weight tensor dimension 0:
    - Get sliced and padded X_p_s with shape (B, C, H, W)
    - Ge the weight tensor at position i of shape (B, 1, H, W)
    - Multiply X_p_s and W[i] to get a partial result tensor Z_p with shape (B, C, H, W)
    - Add the above partial result tensor Z_p to the final result tensor

The above algorithm of step (2) can be called as pad-slice-multiply algorithm. We believe that this is the most efficient one we can get for the following reasons:

(1) Tensor operation is much more efficient than using for loops to go through each element of the tensor under PyTorch framework;
(2) Slicing, which is frequently repeated in the above algorithm, under PyTorch framework does not create a piece of data on memory, although it creates a new tensor instance wrapped on the same piece of data in a different view.

After implemented the module, we ran the module with a single batch of input with shape (B=64, C=3, H=224, W=224). The forward time is **1812.80 ms** and backward time is **18508.86 ms** on CPU. On GPU (we are using Nvidia's K80 in all of our experiments), the forward time is **23.15 ms** and backward time is **235.66 ms**. We can see that the backward part is more than 10 times slower than the forward part. We know that the calculation complexity both step (1) and step (2) are similar, and that PyTorch's Conv2D runs forward and backward pass in a similar amount of time. So the bottleneck of the module is in the backward pass of step (2), which is implemented automatically by PyTorch.

To see how it works by implementing the backward pass by ourselves, we implemented a new version using PyTorch's C++ extension, which is section 3.2.

Also, we can find that the module has only less than 100X speedup on GPU, which means that this version is not fully utilizing the computation power of GPU. To run the module more efficiently in parallel on GPU, we implemented our CUDA version, which is explained in section 3.3.

3.2 C++ implementation

To solve the bottleneck problem mentioned in section 3.1, we split step (1) and step (2) mentioned above. Step (1), which is implemented using PyTorch Conv2D API, is of high efficiency. Improving the bottleneck induced by step (2) is our purpose in this section.

For the forward pass, we used exactly the same algorithm (pad-slice-multiply) as 3.1, except that instead of calling PyTorch in Python, we used PyTorch's Aten and Torch libraries to perform the tensor operation. For the backward pass, we are updating d_x and d_weights by a similar way as defined in the forward pass, which also performs pad-slice-multiply from the gradient of the output to the gradient of weights and inputs.

In both our forward and backward pass implementations, we also utilized the PyTorch's Aten and Torch libraries to the maximum. So we believe out C++ implementation is of the highest efficiency for the same reason as the Python version described in section 3.1.

To bind the C++ code which implements step (2) and Python wrapper which implements other parts of the module, we used functions in setuptools and torch.utils.cpp_extension to bind them together and compile the C++ code before running.

Under the same condition and using the same shape of input as section 3.1, the running time of C++ implementation for the forward pass is **1797.24 ms**, and for the backward pass is **5207.66 ms**. On GPU, the forward pass takes **22.40 ms** and the backward pass takes **25.87 ms**.

Note that the forward pass is pretty much the same as the Python implementation, for we are using exactly the same algorithm explicitly. However, the backward pass has a huge improvement compared to the Python version. The reason of this improvement could be that PyTorch framework is using a dummy (we did not study why it is dummy) and low efficiency way to calculate the gradient automatically.

3.3 CUDA implementation

Although the C++ version has a huge improvement on both CPU and GPU compared to the Python version, we did not define how to run the work in parallel on GPU. This part is implemented in PyTorch's Aten and Torch libraries, which provides a nice abstraction for our C++ implementation level.

However, we believe PyTorch's Aten and Torch libraries are not doing well in job decomposition and allocation, for the speedup of the GPU compared to the CPU is only around 100X. To explicitly decide the decomposition, allocation and data transfer algorithms, we wrote our own CUDA version on GPU.

Compared to the C++ implementation, CUDA implementation is quite different. Instead of using the tensor operation API provided by PyTorch Python primitives or C++ libraries, we performed element-wise operations on each thread in a data-parallel way.

In the forward pass, we used each thread to update on position at dimensions 2 and 3 for the output tensor. In the backward pass, we used each thread to update each position at dimensions 3 and 4 for d_weights and each position at dimensions 2 and 3 for d_x. We tried pretty hard to make sure that we are updating only the position related to that thread instead of others and used chain-rule to find and calculate using all corresponding elements related to the elements to be updated. This part is very hard to be race-free and we tried a lot to guarantee correctness.

In this implementation, we ensured continuity of each tensor before the tensor is passed back and forth between the CPU and GPU. After making sure the tensor's underlying data is continuous on GPU, we can improve the locality of each thread block by ensuring that each block is mainly accessing a consecutive piece of the tensor in the lowest two dimensions, which means that a consecutive part of the underlying data is referenced by each block. Note that there is an exception for the weights tensor. We are traversing through its highest dimension, which means a poor locality. The reason for tolerating this part to have poor locality is that we generated the weight tensor in this specific dimension, which is defined by the primitive of the PyTorch's standard Conv2D API. Making this already generated weight tensor's highest dimension shifted to the lowest dimension and making it contiguous again would result in more overhead in data transfer. And the code would be harder to maintain with a different shape of weights at only CUDA part.

As for decomposition strategy, we are using each block to contain W threads and allocate H blocks. Note that H and W are both 224 in this task, which is a reasonable value for the number of blocks and the number of threads per block as well as the total number of threads launched on GPU. Also, this decomposition makes it easier to guarantee locality mentioned above. This decomposition method also works well with typical input sizes such as (256, 256) and (512, 512), which are the input sizes commonly used in Deep Neural Networks. Things might be different if the input size is much smaller or larger. We are thinking of making an adaptive decomposition algorithm in the future if that kind of situation happens.

Under the same conditions and using the same shape of input as section 3.1, the running time of CUDA implementation for the forward pass is **1.04 ms**, and for the backward pass is **1.29 ms** on GPU, which is about 20X faster than the forward pass in Python extension and 200X faster than the former's backward pass.

3.4 Code Running

All of the code mentioned above can be found in Git Repo:
https://github.com/Simeng96/PyTorch_Extension/tree/master/code

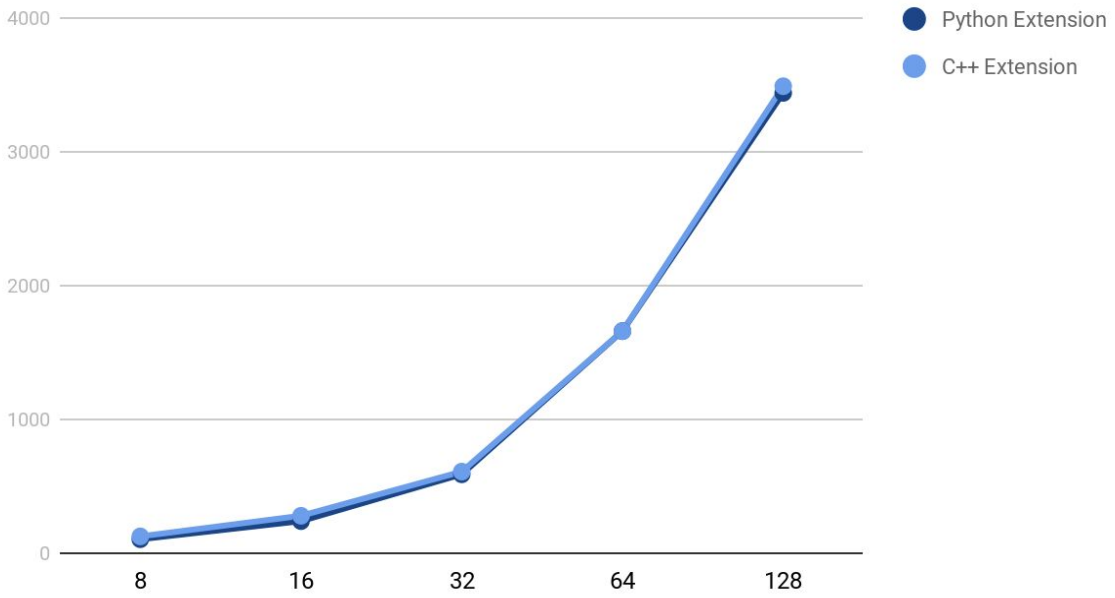
(1) python/ folder contains the original high order convolution model built on PyTorch extension (nn.Module). Use the following commands to run the code:
   - python benchmark.py py cpu
   - python benchmark.py py gpu

(2) cpp/ folder contains a python wrapper and uses c++ to implement the critical part of the high order convolution model. Use the following commands to compile and run the code:
    - cd cpp/
    - python setup.py install
    - cd ..
    - python benchmark.py cpp cpu
    - python benchmark.py cpp gpu
(3) cuda/ folder contains a python wrapper and a c++ wrapper and uses CUDA to implement the critical part of the high order convolution model. Use the following commands to compile and run the code:
    - cd cuda/
    - python setup.py install
    - cd ..
    - python benchmark.py cuda cpu
    - python benchmark.py cuda gpu
(4) tools/ folder are reference data preprocessing tools and data loaders for users who want to use real data to run the module
(5) To check the correctness of C++ and CUDA extension, we use Python extension as ground truth (note that we implement Python extension using PyTorch's nn.module, which only requires the implementation of forward pass and the backprop is implemented automatically):
    - python check.py

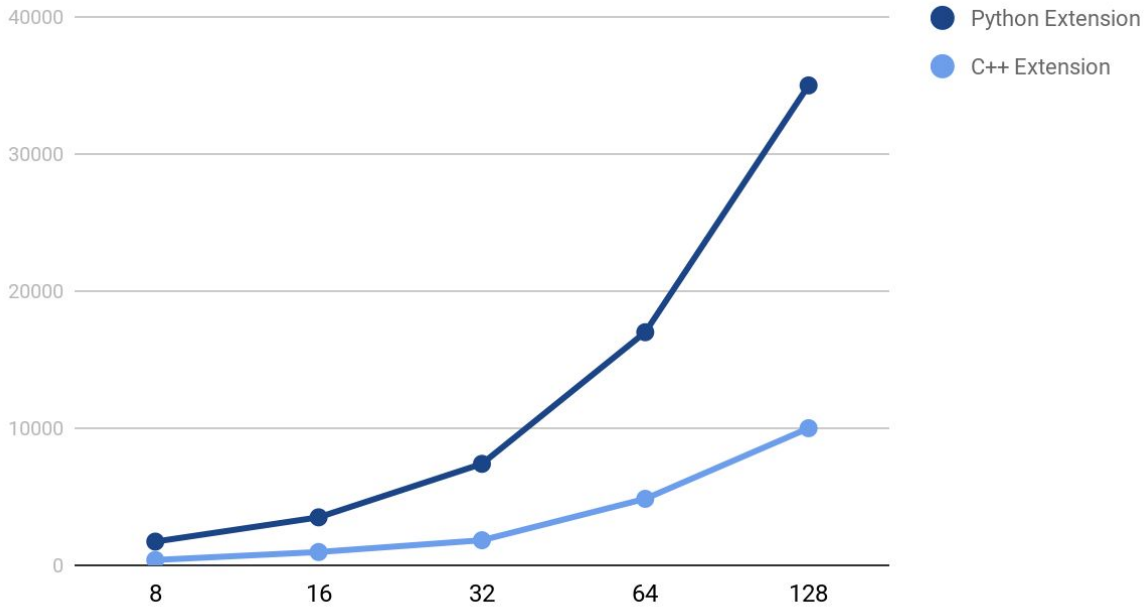
4. Results

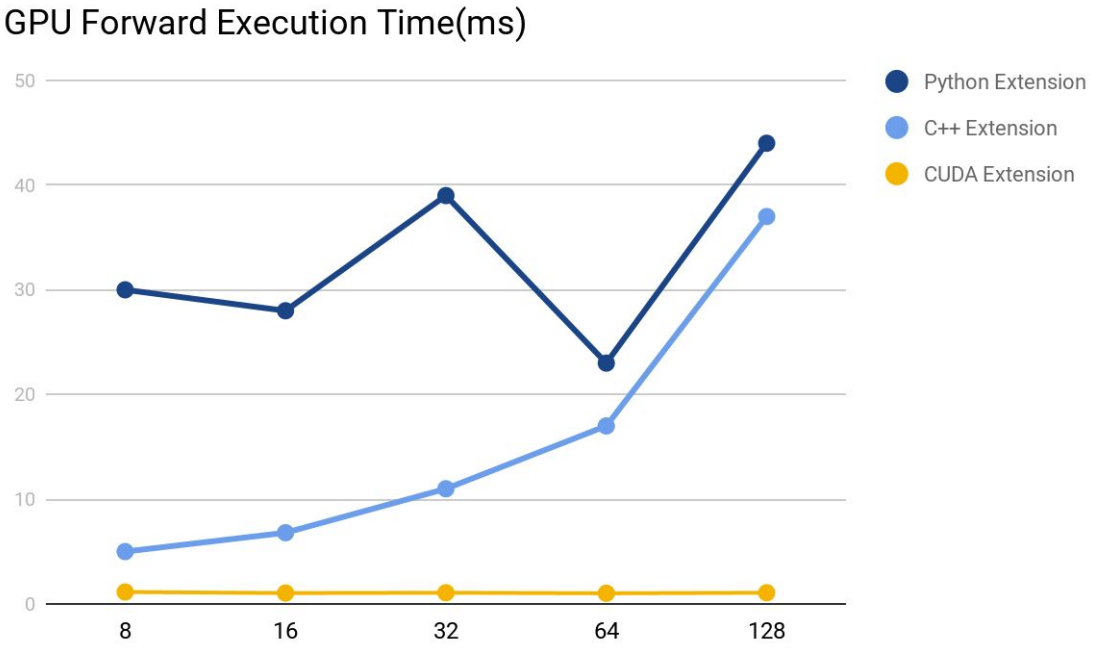
We measured our different implementations with execution time. We run each implementation five times and use the average time as the final result. We experimented both on CPU and GPU with different input sizes. The following graphs show the results of our different implementations of different batch size. The first two graphs compare the efficiency of the PyTorch's C++ extension based implementation with the Python implemented version on CPU. The last two graphs compare the performance of the PyTorch's C++ extension based implementation and CUDA extension based implementation with the basic Python implemented version on GPU. The batch size is range from 8 to 128. Each batch include 588KB input data and 4900KB weights data.

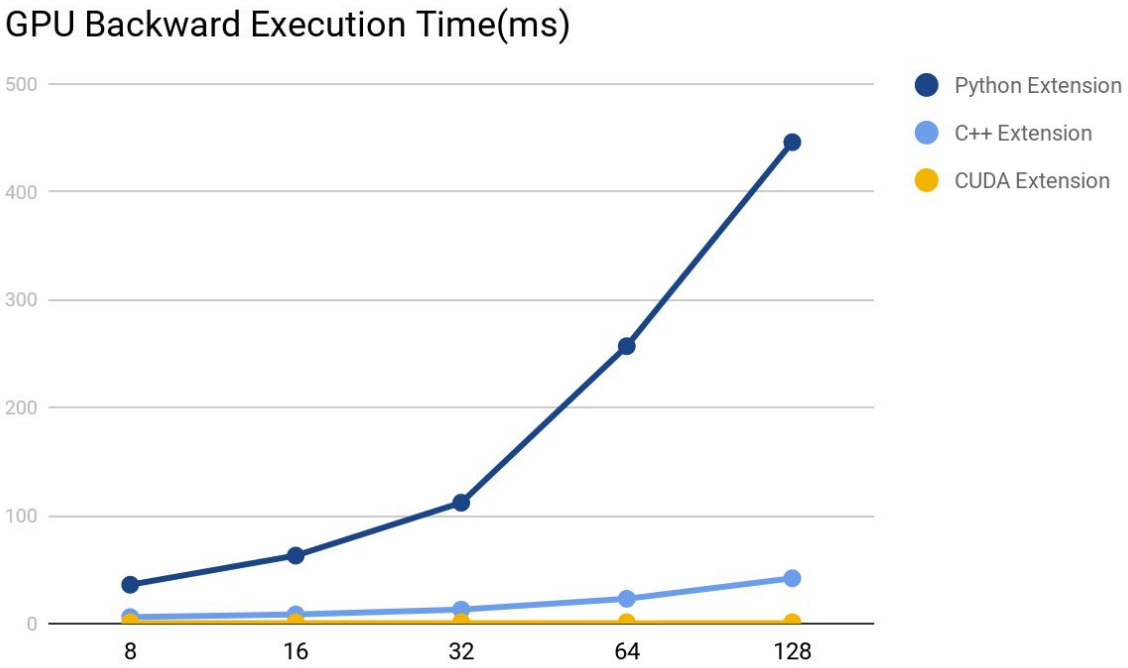
CPU Forward Execution Time(ms)



CPU Backward Execution Time(ms)







From the results, our GPU based CUDA implementation will always achieve better efficiency than C++ extension and Python extension for both forward and backward. And with the batch size increasing, the CUDA extension will always remain about 1.1ms execution time while for Python and C++ extension, the execution time almost double with the batch size increasing. So the speedup is higher when the input data increases. Besides, our CUDA and C++ extension implementations on GPU are more memory saving than basic Python extension

implementation. With 1x NVIDIA Tesla K80, the implementation based on Python extension can only take input data up to 256 batches while the C++ and CUDA extension based implementations can take up to 512 batches of input data.
To analyze how the running time changes with respect to the batch size, we increased the batch size from 8 to 512, however, the running time remains to be around 1.1 ms for forward pass 1.3 ms for backward pass. We can conclude from this observation that the speedup of CUDA implementation is not dominated by the amounts of computation, but something else. Furthermore, the data size to be transferred to GPU (of input and weights tensors) under batch size 512 is about 686MB, which takes around 1ms to transfer from CPU to GPU. This should be the dominant factor that limits the efficiency of the algorithm. The only thing we are not sure is that why does the running time remains the same no matter how we change the batch size. A reasonable guess would be that there is a certain amount of latency related to data transfer from the CPU to the GPU. This will be another topic worth studying in the future.

5. References

[1] https://openreview.net/pdf?id=SkVhlh09tX
[2] https://github.com/pytorch/extension-cpp/tree/master/

6. Work Distribution

Task	Assign
Learn PyTorch's source code and framework	Zhejin, Simeng
Implementation of Python Baseline	Zhejin
Implementation with PyTorch's C++ extension	Simeng
Implementation with PyTorch's CUDA extension	Zhejin
Optimization of GPU version	Zhejin, Simeng
Experiments	Zhejin, Simeng
Reports and poster session	Zhejin, Simeng

50% Zhejin 50% Simeng