CSCE636 Term Project —— Developing a CPU-Based CNN Inference Engine

Di Tian
Department of Electrical & Computer Engineering
Texas A&M University
College Station, TX 77843
tiandi@tamu.edu

Abstract

The highly parallelizable GPUs make training deeper, wider and more sophisticated Convolutional Neural Network(CNN) easily. However, in production environment, CPU is often preferred to process stream data. In this project, we first proposed a CNN inference framework design. We then developed features for supporting CPU-based inference. Finally, we optimized our inference engine from both graph-level and op-level. The engine is tested in a single-core single-thread environment with ResNet-50 model. The results show that our engine is 45.24% faster than TorchScript with no accuracy loss.

1 Introduction

The prevalence of powerful computational devices such as GPUs has dramatically pushed the complexity of Convolutional Neural Networt Architecture to a whole new level. Some techniques, [1][2][3], have been introduced to make the training of super deep, wide and sophisticated CNN easier.

Although highly parallelizable computing devices such as GPUs can also speed-up CNN inference tasks, the data is often feed in a stream fashion, i.e one image at a time, in real production environment. Processing single image on GPUs is not superior, sometimes even inferior to CPU because of the data movement overhead and the weakness on processing serialized task(a single computational graph is almost a serialized computing task).

The main works in this project are:

- A CPU-Based CNN Inference Engine with ONNX model supported. The engine is written in C++. We use OneDNN[4] library as the computing back-end. We applied both graph-level and op-level optimization techniques and gains 45.24% speed-up with no accuracy los over TorchScript in a single-core single-thread environment. The optimization techniques including: operator fusion, data reformatting and various tricks to exploiting data locality.
- We proposed a CNN Inference Framework design. The framework can be easily extended with different model formats and computing platforms.
- A python script for training ResNet[1] with PyTorch.

The rest of the report is organized as follows. Section 2 describes the proposed CNN Inference Framework design. Section 3 describes the optimization techniques we applied in our inference engine. In Section 4 we present experimental methodology and results. We conclude our work and discuss the future directions in Section 5.

Preprint. Under review.

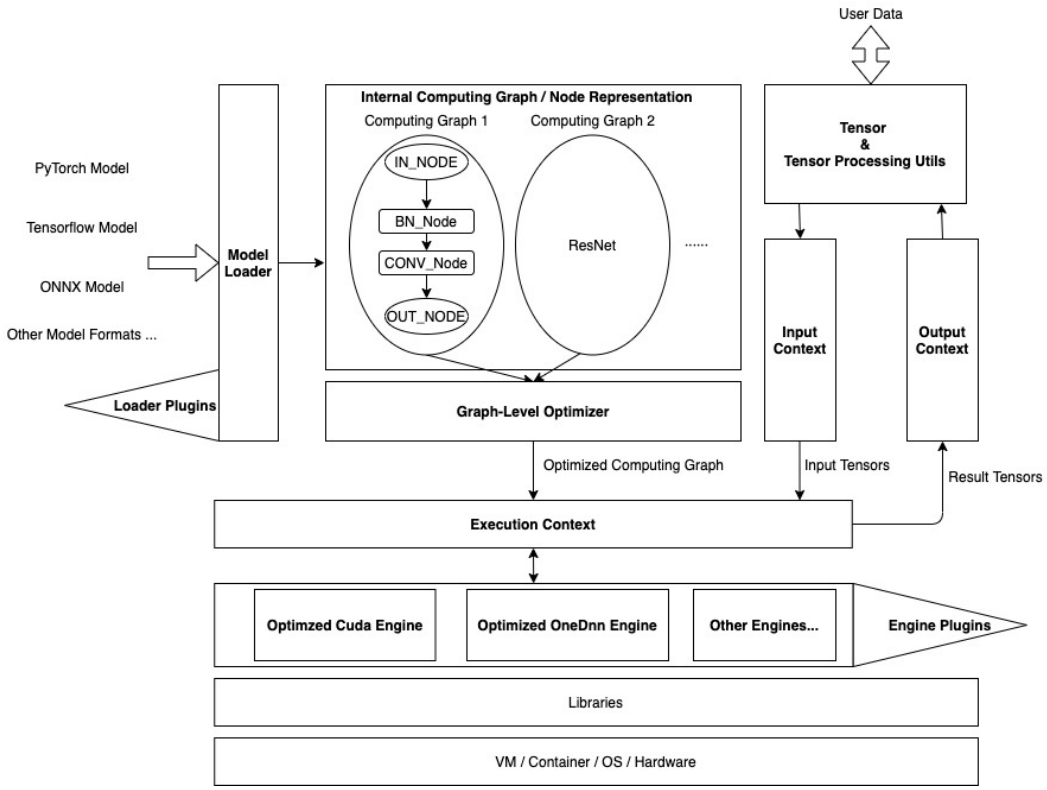

Figure 1: The architecture design

2 CNN inference framework

2.1 The needs

We first summarize the basic needs a CNN inference framework is expected to satisfy:

- Support or can be easily extended with different model formats
- Support or can be easily extended with different platforms / devices / libraries
- Out-of-the-box optimization
- Provide rich data pre-processing utilities
- Easy-to-use input/output encapsulations and APIs

We only summarize the most basic needs to a CNN inference framework. There'are a lot more needs in real production environment, such as logging, monitoring, metrics collection, support for concurrent execution and etc, but they are out of the scope of this project.

2.2 Architecture design

We proposed a easy-to-extend CNN inference framework architecture design which satisfies the needs we listed above.

The architecture design follows a modular deign principle. The main components are introduced as follows.

Model Loader To support multiple model formats, a model loader layer is included to convert different model formats to the internal computing graph representation. A plugin module could also be provided to support users to load their own model formats.

Computing Graph / Node Representation The logical computing process can be abstracted as an acyclic computing graph consists of computing nodes(operators). This layer can be seen as a logical computing plan that is ready for running and optimizing. Later, we will introduce the Optimized Engine layer which generates the physical computing plan according to the logical computing plan for different libraries, platforms and devices.

Graph-level Optimizer The optimization process is decoupled from graph creation process so that new optimization options are easy to integrated into the system. Also, the optimization process can be implemented as independent sub-process so that users can choose and combine the optimization techniques as they want.

Tensor & Tensor Processing Utils Data is usually represented as tensors in the context of Deep learning. A Tensor & Tensor Processing Utils module can be implemented for the use of representing data and providing data pre-processing and post-processing functionalities. We can use existing tensor library to implement this module.

Input & Output Context In production environment, there're a lot of information besides the input and output data to maintain during the inference. For example, developers might need to inject specific information to the log, metrics. This can be done with a Input Context Object. For another example, developers might want to extract the running traces or performance metrics from the engine. This can be done with a Output Context Object.

Execution Context The inference processes usually generate some intermediate results. An execution context object is used to maintain these results.

Optimized Engine Layer To optimize for each specific library, platform and device, an extensible Optimized Engine Layer is included. Engines are implemented and optimized for each computing node(or operator) using different libraries, platforms and devices. Developers only need to create a corresponding engine object for their intended libraries, platforms and devices and feed the computing graph and input data to the engine object to get the result. Also, a plugin module is added so that users can implement their own engines.

3 CNN inference optimizations

3.1 Op-level optimization

For CNN models, most inference time is spent on convolutions. Here, we introduce the techniques we employed to optimize convolution.

3.1.1 Optimized Convolution Algorithm

In this section, we denote the output image dimensions as OC, OH and OW, the kernel dimensions as OC, IC, KH, KW and the input image dimensions as IC, IH, IW. Note that we exclude the batch

size dimension for simplicity. The plain 6-loop convolution algorithm shown in algorithm 1 does not utilize the vector instructions provided by modern CPU.

Algorithm 1: The plain convolution algorithm

```
for oc = 0...OC-1 do
    for ic = 0...IC-1 do
        for oh = 0...OH-1 do
            for ow = 0...OW-1 do
                ih=stride*oh
                iw=stride*ow
                for kh = 0...KH-1 do
                    for kw = 0...KW-1 do
                        O[oc][oh][ow] += I[ic][ih+kh][iw+kw] * W[oc][ic][kh][kw]
                    end
                end
            end
        end
    end
end
```

To utilize them, we need to vectorize the independent part of our computation. One possible solution is to blocked the input data and kernel weights along the channel dimension. The block convolution algorithm is shown in algorithm 2. The VLEN denotes the vector register length.

Algorithm 2: Blocked convolution algorithm

```
OC_b = OC / VLEN
IC_b = IC / VLEN
for oc_b = 0...OC_b do
    for ic_b = 0...IC_b do
        for oh = 0...OH-1 do
            for ow = 0...OW-1 do
                ih=stride*oh
                iw=stride*ow
                for oc = 0...VLEN do
                    for ic = 0...VLEN do
                        for kh = 0...KH-1 do
                            for kw = 0...KW-1 do
                                O[oc_b][oh][ow][oc] += I[ic_b][ih+kh][iw+kw][ic] *
                                  W[oc_b][ic_b][kh][kw][oc][ic]
                            end
                        end
                    end
                end
            end
        end
    end
end
```

3.1.2 Data layout transformation

For the optimized convolution we presented above, one important optimization is to transform the layout of the data so that cache locality can be better exploited.

To better present the layout we employed, we first present the plain data layout in 2D convolution, the NCHW data layout. Where N stands for batch size, C stands for channel size, H stands for height and W stands for width. The plain data layout can be visualized as Figure 2[4].

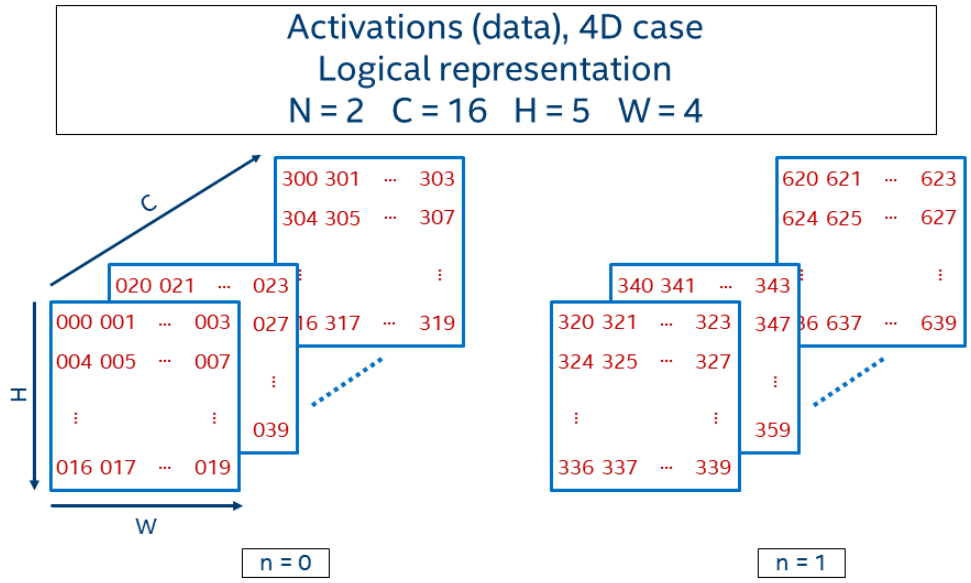

Figure 2: The plain NCHW layout

Source data layout transformation In the most inner loop of the optimized convolution algorithm, the blocked data is manipulated along the channel dimension. Hence, the data layout to exploit this behavior is the NCHW[x]c data layout. Where c means data is clustered along channel dimension, and the block size is x. In this data layout, the kernel parameter on each position is clustered along the channel dimension with small blocks. For example, for NCHW[x]2 layout, assume N = 1, C = 4, H = W = 2, let K denotes the array contains the kernel weights. Then the first 8 parameters are stored in memory sequentially as K[0][0][0][0], K[0][1][0][0], K[0][0][0][1], K[0][1][0][1], K[0][0][1][0], K[0][1][1][0], K[0][0][1][1], K[0][1][1][1]... Figure 3[4] shows another kernel under this layout.

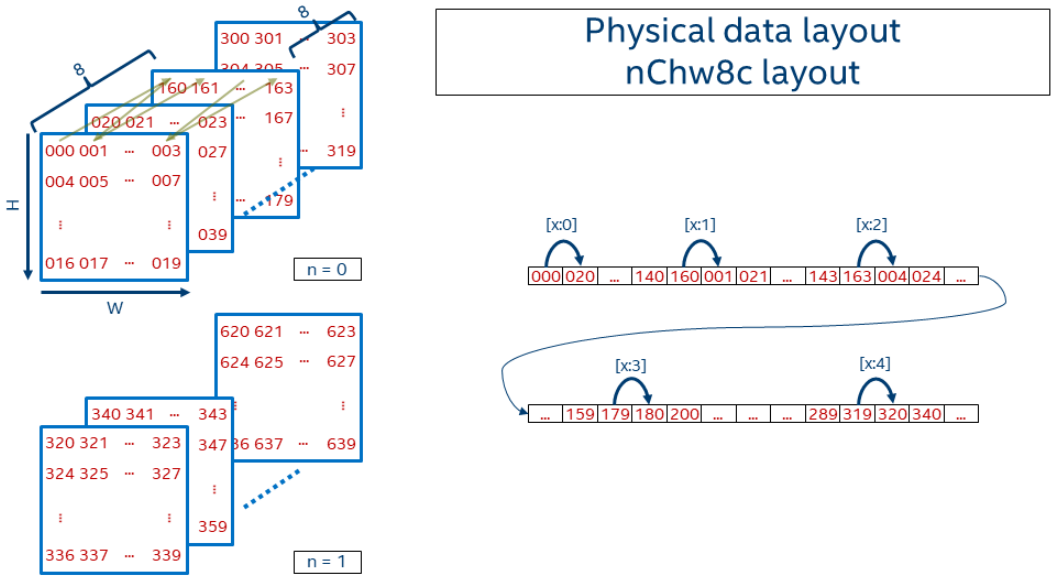

Figure 3: NCHW8c layout

Weight data layout transformation To conform the optimized convolution algorithm, the kernel weights also need to be transformed. The transformation is exactly the same we have done to the source data, i.e the NCHW[x]c data layout. The only difference is that we only need to transform the weight data once. For libraries that support automatic data layout transformation, this can be done by first running the computing graph once and cache the transformed weight data for later use.

3.2 Graph-level optimization

We have discussed about operator-level optimizations. We then look at graph-level optimization techniques.

3.2.1 Operator fusion

Some operators can be fused into one operators to reduce the amount of computation and data movement overhead.

Conv-BN fusion The most commonly used two operators in CNN models: convolution operator and batch normalization operator, can be fused to one convolution.

We illustrate this in an intuitive way. First, convolution can be implemented as a matrix multiplication with im2col algorithm. Let denote this as: $Y = W_{CONV}X + \mathbf{b_{CONV}}$. Second, batch normalization can be seen as multiplying a diagonal matrix with the input matrix and add a vector with it. Let denote this as: $Y = Diag_{BN}X + \mathbf{b_{BN}}$. It's obvious that the two operatos can be fused so that the result convolution has matrix multiplication form as $Y = Diag_{BN}(W_{CONV}X + \mathbf{b_{CONV}}) + \mathbf{b_{BN}}$

Eliminate element-wise operators Element-wise operators such as relu, sigmoid and scale can be fused into the preceding operators as in-place post-operations. This reduces the data movement overhead.

Shortcut fusion Another common operation is shortcut. In CNN models, residual block is frequently used in which the input data is directly sum up with the output data. We find that run an independent summation operation for the shortcut is unnecessary. Instead of allocating a new block of memory to contain the resulting data, we can simply reuse one of the two operands' memory and simply accumulate another operand's data on it. This avoids unnecessary memory allocations and provides better cache locality.

3.2.2 Organize data layout transformation

Data layout transformation is a local method for optimizing a specific operator. However, frequent data layout transformations among operators cause significant data reorder overhead. In this section, We present the global data layout transformation organization to reduce this overhead.

Operators can be categorized to three kinds[5]:

- Layout-oblivious operations. These operations do not care the data layout at all, they can run under any kind of layout. Element-wise operations such as relu, sigmoid, etc., fall in this category.
- Layout-tolerant operations. These operations can run under different layout, but the layout information need to be provided. For example, convolution, batch normalization and pooling fall in this category.
- Layout-dependent operations. These operations can only run under a specific layout. Transformation opera- tions like Flatten, Reshape, etc., fall in this category.

In [5], the author first inserted layout transformation as a special operator into the computing graph and then optimize on the entire computing graph.

We find that for our purpose, since we are using OneDNN[4] library which supports automatic data layout selection, we only need to reorder data layout before executing each operator and don't need to reorder it back after the execution finished. It turns out that this is actually efficient enough for CNN models because after operator fusions, most of the operators are convolutions or other operators

that tolerates different data layout such as batch normalization. Therefore, we usually only need to reorder the data two times. Once before the first convolution and the second before the flatten that precedes the fully connected layer.

4 Results

We test our CNN inference engine on a mac computer with a Intel(R) Core(TM) i7-4770HQ CPU @2.20GHz. The CNN model we used is ResNet-50, the dataset is Cifar-10. The experiments are run under single-core single-thread environment.

Optimization Techniques	Latency(ms)	Improvement over baseline(%)	Accuracy(%)
Baseline(only relu fused)	41.19	0	93.2
Conv & BN fused	37.70	8.47	93.2
Shortcut fused	38.92	5.51	93.2
Conv & BN fused, Shorcut fused	31.87	22.38	93.2
Data reformatting	33.35	19.03	93.2
Data reformatting & Weight caching	27.38	33.52	93.2
All applied	19.47	52.73	93.2
TorchScript	35.56	13.67	93.0

Table 1: Performance for different optimization techniques

We see that both operator fusion and data reformatting provide about 20% speedup. Surprisingly, caching reformatted kernel weight provide another 14% speedup. None of the techniques has negative influence on the prediction accuracy. Combined all of the optimization techniques, our inference engine outperforms TorchScript by 45.24%.

5 Conclusion and Future Directions

In this project, we first proposed a easy-to-extend CNN inference framework design. We then implement a CPU-Based CNN inference engine under this framework. We applied both graph-level optimization and op-level optimization techniques, including operator fusion, data layout transformation, reformatted weight caching and layout transformation organization. The results show that operator fusion and data layout transformation both yield about 20% speedup. Caching reformatted kernel weight provide another 14% speedup. By combining these techniques, we gained 52.73% speedup over the baseline engine and 45.24 % speedup over TorchScript.

The potential future directions including:

- Optimizing the inference engine under multi-core multi-thread environment
- Employing model quantization techniques to further improve inference performance
- Exploring mixed CPU-GPU inference, developing dynamic device scheduling algorithm to fully utilize the overall computational resources

References

[1] He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: CVPR. (2016)

[2] He, K., Zhang, X., Ren, S., Sun, J.: Identity Mappings in Deep Residual Networks. In: ECCV. (2016)

[3] Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: ICML. (2015)

[4] OneDNN: OneDNN documentation. https://oneapi-src.github.io/oneDNN/index.html, 2019. (Online; accessed 26-Nov-2020)

[5] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, Yida Wang: Optimizing CNN Model Inference on CPUs. In: USENIX. (2019)