Inception-V3 Inference Booster

Hongjie Fang[*]
Department of Computer Science and Engineering
Shanghai Jiao Tong University
galaxies@sjtu.edu.cn

Peishen Yan[†]
Department of Computer Science and Engineering
Shanghai Jiao Tong University
1050335889@sjtu.edu.cn

Haoran Zhao[‡]
Department of Computer Science and Engineering
Shanghai Jiao Tong University
zhao.hr@sjtu.edu.cn

Abstract

In this work, we build the Inception-V3 network in CUDA from scratch. After implementing basic inference process of the model which produces the correct results, we analyze time-consuming modules and optimize them respectively. For convolution optimizations, we apply implicit im2col algorithms, which shows better efficacy comparing to CUDNN implementations in experiments. The inference time of our final Inception-V3 inference booster reaches **61.096ms** on average, which is only 60% of the CUDNN inference time. Our implementations is publicly available at `https://github.com/galaxies99/inception-cuda`.

1 Pipeline Overview

We use a bottom-up approach to build and test the modules that will be repeated used in Inception-V3[1] network (*e.g.* convolution layer, pooling layer, fully-connected layers, *etc.*), as well as some basic operations (*e.g.* concatenation, gathering, ReLU activation functions, *etc.*). As there are several modules with similar structures in the network (*e.g.*, inception{A,B,C,D,E} modules, input & output layer *etc.*), we build these modules using encapsulated functional modules mentioned before and test each modules simultaneously. Finally, we stitch the modules together according to the full network structure to obtain the Inception-V3 network. The implementation pipeline is illustrated in Fig. 1.

For each module, we use several methods to implement its function in order to check the correctness of our implementations and compare the performance among several implementations. We first implement the C++-based CPU method and the CUDA-based method. After checking correctness of these implementations, we use CUDNN to build the modules for performance comparisons. After finishing implementations of all modules and the full Inception-V3 network, we also use PyTorch framework to implement the network in Python for debugging.

[*]Student ID: 518030910150
[†]Student ID: 518030910094
[‡]Student ID: 518021910481

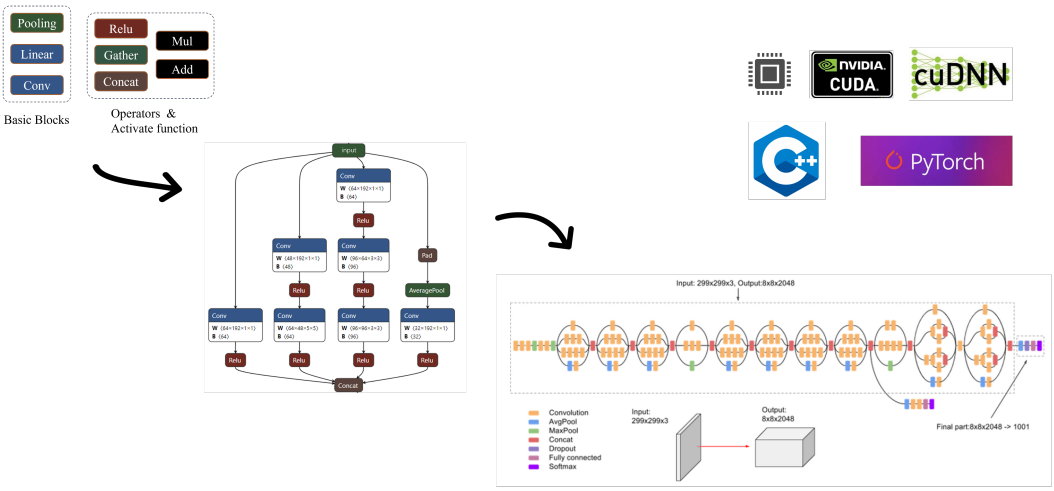

Figure 1: Inception-V3 inference implementation pipeline

Another task we have to do is to load the model weights into our C++ code. Since there is no suitable library and modules for C++ to directly load the onnx model weights, we first use python package `onnx` to load the model weights and convert them into JSON format. Then we are able to write a C++ parser to directly parse the JSON file and load the weights in it.

The basic version of our implementations is able to produce the correct results, but the inference time reaches about 3 minutes, which is quite unacceptable. After optimizations, when there is no other process occupying GPU, our Inception-V3 inference booster can reaches $\sim$ **60ms** per inference with O2 compile option and $\sim$ **130ms** per inference without O2 compile option, as shown in Fig. 2.

```
group1@acalab-W760-G30:~/src$ /usr/local/cuda-10.2/bin/nvcc -arch=sm_60 -fmad=false -lcuda -lcublas -lcudnn -lineinfo utils.cu conv.cu fc.cu pooling.c
u activation.cu opers.cu layers.cu inception.cu inception_main.cc -o ../test/inception_main -Wno-deprecated-gpu-targets -O2
inception_main.cc: In function 'int main()':
inception_main.cc:82:43: warning: deprecated conversion from string constant to 'char*' [-Wwrite-strings]
     readInput("../data/inceptionInput.txt");
                                           ^
inception_main.cc:83:45: warning: deprecated conversion from string constant to 'char*' [-Wwrite-strings]
     readOutput("../data/inceptionOutput.txt");
                                             ^
In file included from inception_main.cc:7:0:
loader.hpp: In function 'Inception load_weights_from_json(const char*, bool)':
loader.hpp:116:54: warning: ignoring return value of 'int fscanf(FILE*, const char*, ...)', declared with attribute warn_unused_result [-Wunused-resul
t]
                            fscanf(fp, "%c", &ch_num);
                                                    ^
loader.hpp:131:54: warning: ignoring return value of 'int fscanf(FILE*, const char*, ...)', declared with attribute warn_unused_result [-Wunused-resul
t]
                            fscanf(fp, "%c", &ch_num);
                                                    ^
inception_main.cc: In function 'void readInput(char*)':
inception_main.cc:22:47: warning: ignoring return value of 'int fscanf(FILE*, const char*, ...)', declared with attribute warn_unused_result [-Wunused
-result]
            fscanf(fp, "%lf", &inputArr[i][j]);
                                           ^
inception_main.cc: In function 'void readOutput(char*)':
inception_main.cc:32:50: warning: ignoring return value of 'int fscanf(FILE*, const char*, ...)', declared with attribute warn_unused_result [-Wunused
-result]
            fscanf(fp, "%lf", &benchOutArr[i][j]);
                                              ^
group1@acalab-W760-G30:~/src$ ../test/inception_main
Average Time is: 61.095844 ms
```

Figure 2: The final performance of our implementations

2 Module Optimizations

2.1 Fully-Connected Layer Optimizations

Fully-connected layer lies in the output layer of the Inception-V3 network. It is essentially a matrix multiplication, or the multiplication of a matrix by a vector when the batch size is equal to 1. So our optimizations basically follow the optimizations of matrix multiplications.

In matrix-vector multiplication, each element of the matrix is computed once, but the elements of the vector are used multiple times, so we can use shared memory to speed up the computation if we apply tilling method in the calculations. For example, as shown in Fig. 3, the tile width is 2, so there are 2 threads in each block, each sharing a space of length 2, and each thread is responsible for loading 1 input element in each round. Thus, for every 2 outputs computed, the entire input is loaded on average only 1 time per element, which may save a lot of time.

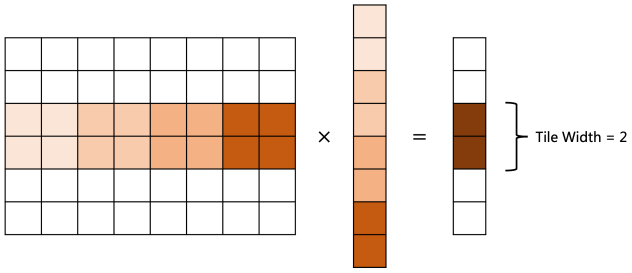

Figure 3: Tilling example

2.2 Convolution Layer Optimization

The focus of our optimization is the convolution layers. The convolution layers are found throughout our network, with totally 94 apperances. Therefore, optimizing the convolution layers will essentially optimize our Inception-v3 network. For a basic convolution operation shown in Fig. 4,

- **Parameter:** a $C_{out} \times C_{in} \times K_h \times K_w$ weight matrix $\mathbf{W}$, an optional bias vector $\mathbf{b}$, and other parameters like input channels C_{in}, output channels C_{out}, kernel size K_h, K_w, stride s_h, s_w and padding size p_h, p_w;
- **Input:** an $N \times C_{in} \times H_{in} \times W_{in}$ tensor $\mathbf{s}^{(\text{in})}$;
- **Output:** an $N \times C_{out} \times H_{out} \times W_{out}$ tensor $\mathbf{s}^{(\text{out})}$, where H_{out} and W_{out} are the output size given by

$$H_{out} = (H_{in} - K_h + 2p_h)/s_h + 1, \quad W_{out} = (W_{in} - K_w + 2p_w)/s_w + 1.$$

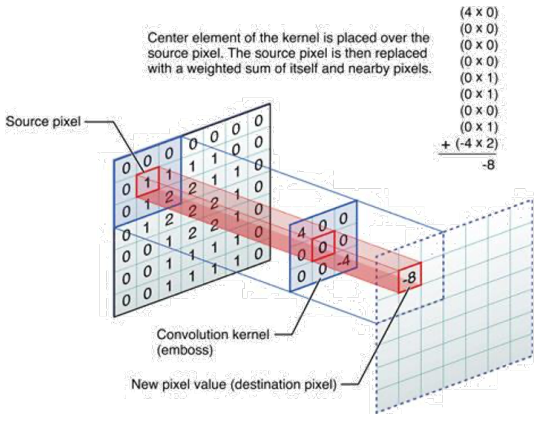

Figure 4: Convolution

The convolution formula can be written as

$$\mathbf{s}^{(\text{out})}_{n,c_o,h,w} = \mathbf{b}_{c_o} + \sum_{c_i=1}^{C_{in}} \sum_{k_h=0}^{K_h-1} \sum_{k_c=0}^{K_c-1} \mathbf{W}_{c_o,c_i,k_h,k_w} \mathbf{s}^{(\text{in})}_{n,c_i,s_h h+k_h-p_h,s_w w+k_w-p_w}$$

where n is the batch identifier and N is the batch size.

For naive CPU implementation, it takes $O(NC_{out}H_{out}W_{out}C_{in}K_hK_c)$ to complete a convolution calculation. For correctness consideration, our basic CUDA implementation simply calculates the number of multiplications in a forward process, and dispatches the multiplications to different blocks/threads. Hence, in the experiments later, we can see that the performance gain is limited.

After looking into convolution optimizations, we found that it can be further optimized using **im2col** method. The main idea is that, if we combine the three summation sign together and regard the summation as the multiplication between a row vector and a column vector, then the convolution operation can be viewed as the matrix multiplication in some sense, as illustrated in Fig. 5.

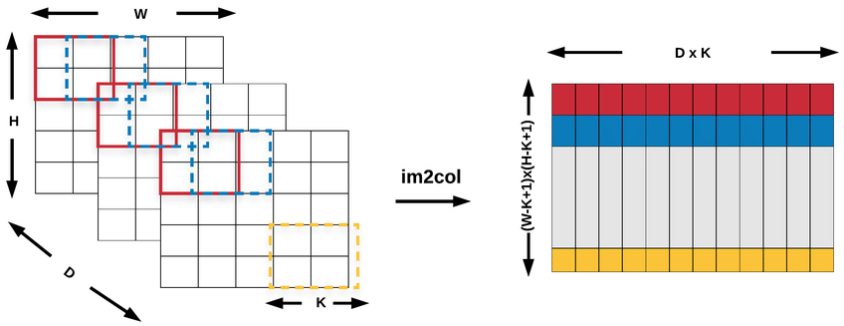

Figure 5: the illustration of im2col method [2]

Specifically, we can regard the weight matrix as an $C_{out} \times (C_{in} \times K_h \times K_w)$ 2D matrix $\widetilde{\mathbf{W}}$. For every (n, h, w) pair, we regard input $\mathbf{s}^{(\text{in})}_{n,c_i,s_h h+k_h-p_h,s_w w+k_w-p_w} (1 \le c_{in} \le C_{in}, 0 \le k_h < K_h, 0 \le k_w < K_w)$ as a column vector of size $C_{in} \times K_h \times K_w$. Hence the input is expanded into an $N \times (C_{in} \times K_h \times K_w) \times (H_{out} \times W_{out})$ 3D matrix $\widetilde{\mathbf{S}}^{(\text{in})}$, which may contain duplicate elements. Therefore, for a given batch n, if we assume the output is $\mathbf{s}^{(\text{out})}_n$, then

$$\mathbf{s}^{(\text{out})}_n = \widetilde{\mathbf{W}}\widetilde{\mathbf{S}}^{(\text{in})} + \mathbf{b}^{(\text{broadcast})}$$

where $\mathbf{b}^{(\text{broadcast})}$ is the bias broadcasted in the last two dimensions (height & width of input).

Therefore, we convert convolution to matrix multiplication, and we can use the same method in optimizing fully-connected layers to optimize the new matrix multiplication problem. However, if we explicit generate the input matrix $\widetilde{\mathbf{S}}^{(\text{in})}$, it may take a lot of GPU memory space since there are duplicates elements and the array is basically disposable. If we analyze the performance of explicit im2col method, we can see that GPU memory allocation and de-allocation is time-consuming. Therefore, we try to further optimize the im2col method by inplace calculation, *i.e.*, implicit im2col.

The basic idea is simple, we just need to calculate the correspondence between $\widetilde{\mathbf{S}}^{(\text{in})}$ and the actual input $\mathbf{s}^{(\text{in})}$ in tilling process during runtime. The correspondence relation is actually given in the convolution formula, *i.e.*,

$$\text{im2col}(hW_{out}+w, c_{in}K_hK_w+k_hK_w+k_w) \to \text{output}(h, w) \to \text{input}(s_h h+k_h-p_h, s_w w+k_w-p_w)$$

Therefore, we apply implicit im2col method to optimize the convolution layer with no additional GPU memory cost.

3 Experiments

3.1 CUDNN Implementations

We use CUDNN to implement the inception-V3 network for performance comparisons. We implement the convolution layer, pooling layer, activation layer and the fully-connected layer in CUDNN.

For convolution layer, we refer to the source code of the famous Caffe framework[3], and use `cudnnGetConvolutionForwardAlgorithm_v7(...)` to find suitable algorithms, then use `cudnnGetConvolutionForwardWorkspaceSize(...)` to get the workspace size and allocate spaces. `cudnnConvolutionForward(...)` and `cudnnAddTensor(...)` are used to multiply weights to the inputs and add bias to result respectively.

For pooling layer, we use `cudnnPoolingForward(...)` to perform pooling.

For activation layer, we use `cudnnActivationForward(...)` to apply activation functions.

For fully-connected layer, since fully-connected layer can be regarded as a special convolution layer with 1×1 kernel, we use the same CUDNN implementation in the convolution layer except specifying the kernel size as 1×1.

Other parts (*e.g.*, concatenation, gathering, *etc.*) remain the same as our CUDA implementations.

The performance of CUDNN implementations is shown in Fig. 6, from which we can observe that, it takes about $\sim$ **100ms** to perform inference with O2 compile option.

```
group1@acalab-W760-G30:~/src$ /usr/local/cuda-10.2/bin/nvcc -arch=sm_60 -fmad=false -lcuda -lcublas -lcudnn -lineinfo utils.cu conv.cu fc.cu pooling.c
u activation.cu opers.cu layers.cu inception.cu inception_cudnn.cc -o ../test/inception_cudnn -Wno-deprecated-gpu-targets -O2
inception_cudnn.cc: In function 'int main()':
inception_cudnn.cc:72:43: warning: deprecated conversion from string constant to 'char*' [-Wwrite-strings]
     readInput("../data/inceptionInput.txt");
                                           ^
inception_cudnn.cc:73:45: warning: deprecated conversion from string constant to 'char*' [-Wwrite-strings]
     readOutput("../data/inceptionOutput.txt");
                                             ^
In file included from inception_cudnn.cc:7:0:
loader.hpp: In function 'Inception load_weights_from_json(const char*, bool)':
loader.hpp:116:54: warning: ignoring return value of 'int fscanf(FILE*, const char*, ...)', declared with attribute warn_unused_result [-Wunused-resul
t]
                            fscanf(fp, "%c", &ch_num);
                                                    ^
loader.hpp:131:54: warning: ignoring return value of 'int fscanf(FILE*, const char*, ...)', declared with attribute warn_unused_result [-Wunused-resul
t]
                            fscanf(fp, "%c", &ch_num);
                                                    ^
inception_cudnn.cc: In function 'void readInput(char*)':
inception_cudnn.cc:22:47: warning: ignoring return value of 'int fscanf(FILE*, const char*, ...)', declared with attribute warn_unused_result [-Wunuse
d-result]
            fscanf(fp, "%lf", &inputArr[i][j]);
                                              ^
inception_cudnn.cc: In function 'void readOutput(char*)':
inception_cudnn.cc:32:50: warning: ignoring return value of 'int fscanf(FILE*, const char*, ...)', declared with attribute warn_unused_result [-Wunuse
d-result]
            fscanf(fp, "%lf", &benchOutArr[i][j]);
                                                 ^
group1@acalab-W760-G30:~/src$ ../test/inception_cudnn
Average Time is: 102.594482 ms
```

Figure 6: The final performance of CUDNN implementations

3.2 Performance Test

After implementations, we use the following command to compile our codes.

```
/usr/local/cuda-10.2/bin/nvcc -arch=sm_60 -fmad=false -lcuda -lcublas -
    lcudnn -lineinfo utils.cu conv.cu fc.cu pooling.cu activation.cu opers.
    cu layers.cu inception.cu inception_main.cc -o ../test/inception_main -
    Wno-deprecated-gpu-targets -O2
```

Here we use the option `-fmad=false` to enhance the precision of the inference process, and use `-arch=sm_60` to enable using double in CUDA code, which is also for precision consideration. The `-lcudnn` is used only for comparing our performance with cudnn implementations.

The results are shown in Tab. 1, which are evaluated when no other process is occupying the GPU. Notice that the CUDNN results are tested after warm-up forwards and handle creation.

Table 1: The evaluation results of several implementation methods

Method	Inference Time (ms)	Max GPU Memory Occupation
CPU	$\sim$ 180,000	-
CUDA (ours, basic)	$\sim$ 36,000	**530MB**
CUDNN	102.594	750MB
CUDA (ours, optimized)	**61.096**	**530MB**

3.3 Module Performance Test

We also conduct experiments to verify the effectiveness of our optimizations.

3.3.1 Fully-Connected Layers

For fully-connected layers, we choose the input channels 2048 and the output channels 1000, which corresponds to the channels of the fully-connected layer in the Inception-v3 network. The experiment results are listed in Tab. 2.

Table 2: The evaluation results of fully-connected layer implementations

Method	Inference Time (ms)
CPU	12.871
CUDA (ours, basic)	0.692
CUDA (tile width 2)	0.328
CUDA (tile width 4)	0.282
CUDA (tile width 8)	**0.208**

Therefore, we choose the optimized tilling matrix multiplication with tile size 8, which is **62x** faster than CPU implementation and over **3x** faster than our basic implementations.

3.3.2 Convolution Layers

For convolution layers, we evaluate the statistics under the following settings.

- Batch size 4;
- Input channels 16, output channels 16;
- Input size 128×128;
- Kernel size 3×3;
- stride 1×1;
- padding size 1×1.

For implicit im2col method, the tiling size is set to 16 after several experiments about it. The final results are shown in Tab. 3.

Table 3: The evaluation results of convolution layer implementations

Method	Inference Time (ms)
CPU	1392.27
CUDA (ours, basic)	223.227
CUDNN	1.096
CUDA (ours, implicit im2col)	**0.891**

Therefore, our optimized convolution implementation is approximately **1500x** faster than CPU, approximately **250x** faster than basic CUDA implementation. Moreover, it only takes 81% of the CUDNN execution time.

3.4 Discussions

Though CUDNN library has probably the best CUDA implementations, which have been tested in numerous experiments, in our experiment it performs worse than our implemented implicit im2col method. Therefore, in this section, we give some possible explanations about this weird results.

- The CUDNN implementations will take time to find the most suitable algorithms for convolution every time using `cudnnGetConvolutionForwardAlgorithm_v7(...)`, and the process may take a little bit time, since it will check the GPU architecture, and run a small sample on every algorithms (include im2col, Winograd algorithms, *etc.*).
- Our implementation simply use implicit im2col methods, which may be the suitable one to the given GPU after experiments.
- Our implementation use tile size 16 after comparisons, which may be more efficient than the tile size of the CUDNN implementations in certain architecture.

4 Conclusion

In summary, we build the Inception-V3 inference booster in CUDA from scratch. After analyses about each module, we apply implicit im2col method to optimize the convolution layers of the booster, and use tilling method to optimize the fully-connected layer. The optimizations give satisfactory results in the following experiments. In performance test of each module, our implemented implicit im2col method shows better efficacy than CUDNN implementations. We make a few discussions of the results. Finally, the inference time of our implementations reaches **61.096ms** per time on average, which is even faster than CUDNN implementations.

Acknowledgments

We would like to express our deepest gratitude to Prof. Xiaoyao Liang and Prof. Zhuoran Song for the detailed introduction about the parallel and distributed computing, especially about GPU and CUDA, from which we have learnt lots of knowledge that is useful in further researches.

We would like to thank teaching assistants for their dedications and supports.

Appendix: Group Project Division

The members of our group and the corresponding division of project are shown as follows.

- **Hongjie Fang (group leader, 518030910150)**: Convolution layer implementation and optimization, CUDNN implementation, InceptionD and InceptionE module implementations, onnx loader implementation, presentation, report writing.
- **Peishen Yan (518030910094)**: Pooling layer implementation, input & output layers and InceptionC module implementations, full network construction, presentation.
- **Haoran Zhao (518021910481)**: Fully-connected layer implementation and optimization, InceptionA and InceptionB module implementations, presentation.

References

[1] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

[2] M. Loukadakis, J. Cano, and M. O'Boyle, "Accelerating deep neural networks on low power heterogeneous architectures," 2018.

[3] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, 2014.