ResNet with efficient GPU kernels

Feng Li, Christopher Goebel, Yuankun Fu

May 2019

Abstract

Deep neural networks (DNN) have become a hot topic recently. As network becomes deeper and deeper, it can be different difficult to train DNN (e.g. vanishing.exploding gradients). ResNet, first proposed in 2015, uses residual blocks in the network to make it possible to stack more layers in the network [4]. ResNet can be easily defined in many existing frameworks, however we are looking into the possible performance improvement of using different CUDA optimization techniques and provide a minimal but efficient implementation of ResNet. In this work, we have implemented the network of ResNet and provide efficient computation kernels using CUDA. We also compared our CUDA implementation with cuDNN [2], our results show that the best cuDNN implementation can achieve at most 12X faster than other algorithm in our experiment.

Contents

1 Introduction

In this paper, we first introduce the design of our ResNet network, followed by efficient gpu kernels implementation, then we compare our convolution implementation with cuDNN.

2 Framework

In this section, we firstly introduce the structure of our simplified ResNet, then we give more detailed insights of our framework, which includes the following information

1. Tensor design (how to manage host and gpu memory for our framework).
2. Simple layers (some simple layers other than convolution layer and pooling layer).
3. Module layers and residual blocks.
4. Other utilities and overall workflow during training process.

2.1 Network Introduction

Residual learning, which is one of of the key concept in ResNet, is the reason why ResNet can handle large number of layers. In the CIFAR10 experiment, the authors used a ResNet of three stages, each of which has 2 residual blocks. The original net is shown in the left part of Figure 1. The full-length ResNet has more complex structure, and takes much longer time to train.

Since we want to focus more efficient GPU kernels and apply those optimization techniques we have learned from the class, we decide to use a simplified ResNet structure instead. The structure is shown in the right part in Figure 1. Even though the structure is minimized, we still have all of the required components for constructing a ResNet.

2.2 Detailed Design

In this part, we will explain more detailed design decisions we made during the development of this ResNet GPU implementation. The most basic building block of our work is tensor, which is a representation of multi-dimensional array. Using tensors in GPU, we can build different neuron network layers which conduct the computation totally in GPU. Then those layers are composed into larger module layers, and using simple layer and module layers, we build the simplified ResNet.

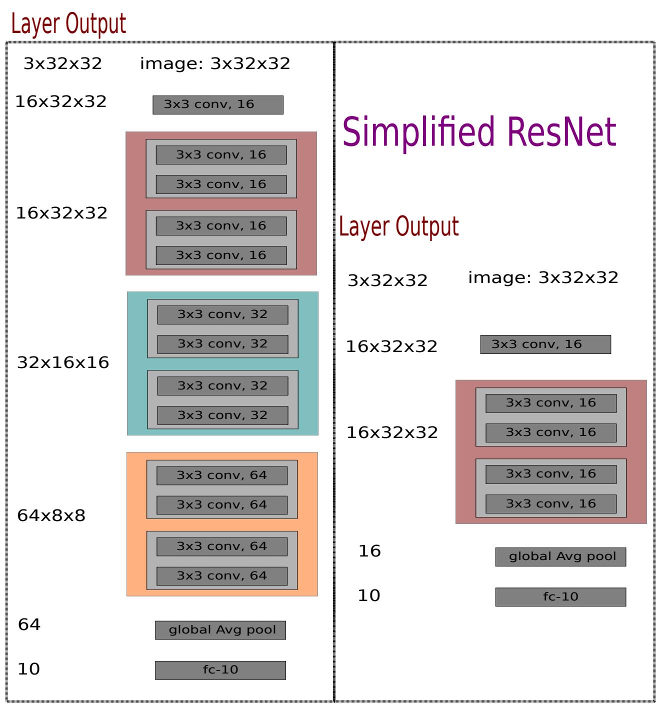


Figure 1: Simplified ResNet

2.2.1 Tensor

Each of the layer takes a input an tensor, generates a output tensor, and the network will pass the output of this layer to upper layer (usually by reference).

Since in our network, data are fed in batches, input has shape $N \times C \times H \times W$, where N is the number of input images in this batch, C is the number of channels in each image, and H and W are the spatial information of the image(height, width) respectively.

From the bottom(input) to the top(loss layer, e.g. softmax) of the network, the N stays the same, while other three dimensions might vary. We decide to store the actual information in plain C array, and use a *dim_t* type to describe the dimension of the tensor. Note that this C array can be either in host or device, so we have a *memory_type_t* to describe it.

```
typedef enum {
  CPU_MEM = 0,
  GPU_MEM = 1,
  BAD_MEM = 2,
  EMPTY_MEM = 3,  // uninitialized, e.g. net input
} memory_type_t;

typedef struct tensor {
  dim_t dim; //  uint dims[4]
  memory_type_t mem_type;
  T *data;
} tensor_t;  // tensor
```

Listing 1: Tensor Definition

Apart from the input/output tensor for each layers, we also need tensor structure for layer cache and learnable parameters of each layer, which are going to be introduced in next subsection 2.2.2

2.2.2 Layer

Till now, the following layers are implemented and tested in GPU:

1. convolution layer
2. relu-conv layer
3. residual blocks
4. pooling layer

There are two types of layers in our framework: **simple layer** and **module layer**. Simple layers are those who only do one single operation, such as convolution, relu, pooling layers. Module layers are built on simple layers or other module layers.

Relu layer can be one example of simple layer, as we can see in listing 2. For relu layer, all input tensor and output tensor and their gradient tensors are pre-allocated in GPU, the forward/backward operations only do computation without any memory allocation/deallocations.

```
status_t relu_forward_device(tensor_t const d_x, lcache_t* cache,
                             tensor_t d_y) {
  do_device_relu_forward<<<32, 1024>>>(d_x, d_y);

  if (cache) {
    lcache_push(cache, d_x);
  }
  return S_OK;
}
status_t relu_backward_device(tensor_t const d_dx, lcache_t* cache,
                              tensor_t d_dy) {
  // lcache_dump_stat(cache);
  tensor_t d_x = lcache_pop(cache);
  do_device_relu_backward<<<32, 1024>>>(d_dx, d_x, d_dy);

  return S_OK;
}
```

Listing 2: Relu Layer in GPU

The **cache** in the listing 2 stores temporary information between forward/backward process during the training, which will be introduced in more details next.

2.2.3 Cache and Module Layers

Another example to demonstrate the usage of cache can be a linear (fully-connected) layer, which takes an input X, applied with weight W, produces the output Y.

$$Y = X \times W \tag{1}$$

During back-propagation, we can calculate the gradient of loss w.r.t the W:

$$dW = X^T \cdot dY \tag{2}$$

So, when we derive the gradient of loss w.r.t W, apart from the gradient information from higher layer (dY), we also need some data information from our forward phase (in this case X). How to store X in the cache? Our first solution is to allocate a new tensor in device during forward, and during backward it will be freed. This naive approach is easy to implement and maintain, since the layer itself has the ownership of the cached tensors. The disadvantage is that we use more memory space to store redundant data, and allocation/de-allocation in each forward/backward pass can be very expensive. A better approach is to allocate cache space in advance, and de-allocate in the end, and during forward/backward in each iteration, just reuse the same GPU memory.

This "pre-allocate" strategy can be straightforward for simple layer, but difficult to implement for module layers. For example, a residual block consists of two conv-relu modules, and each of the conv-relu module consists of a convolution layer and relu layer. If we want to allocate space for the whole residual block, we need prepare space for each of the unit layers, and arrange them in a patterned order, so that they can be used by different unit layer correctly. Listing 3 shows how we compose residual blocks, the *context* tracks the cache information for the whole residual block, and it's also passed into the sub-module layers. More detailed implementation of module layers and memory management can be found in *src-device/layer_sandwich_device.cu* in the source tree.

2.2.4 Network and ResNet

After we construct all of the layers, we can concatenate them together one by one to form the whole ResNet network.

```
status_t resblock_forward_device(cublasHandle_t handle, tensor_t
    const d_x, tensor_t d_w1, tensor_t d_w2, lcache_t* cache,
    conv_param_t const params, tensor_t d_y, struct
    layer_context_device* context) {
  tensor_t d_tmp = context[0].d_tmp;
  // lower conv_relu layer inside the residual block
  conv_relu_forward_device(handle, d_x, d_w1, cache,
        params, d_tmp, &context[2]);
  // upper cov_identity-relu layer inside the residual block
  conv_iden_relu_forward_device(handle, d_tmp, d_x,
        d_w2, cache, params, d_y, &context[3]);
  return S_OK;
}
status_t resblock_backward_device(cublasHandle_t handle, tensor_t
    d_dx, tensor_t d_dw1, tensor_t d_dw2, lcache_t* cache,
    conv_param_t const params, tensor_t const d_dy, struct
    layer_context_device* context) {
  // get cached variable
  tensor_t d_dtmp = context[0].d_dtmp;
  tensor_t d_dx_iden = context[1].d_dtmp;
  // upper conv-identity-relu layer inside the residual block
  conv_iden_relu_backward_device(handle, d_dtmp, d_dx_iden, d_dw2,
        cache, params, d_dy, &context[3]);
  // lower conv_relu layer inside the residual block
  conv_relu_backward_device(handle, d_dx, d_dw1, cache, params,
        d_dtmp, &context[2]);
  elementwise_add_inplace_device<<<32, 1024>>>(d_dx, d_dx_iden);
  return S_OK;
}
```

Listing 3: Residual Block

Figure 2 demonstrates most of the memory information we need maintain, to construct a network like ResNet. In each layer, there are input/output tensors and their gradients, learnable parameters such as weight information for linear layer and filter information for convolution layer, and cache information we mentioned above. All of the memory in the Figure 2 are constructed when the network is constructed, and no other dynamic host/device memory needs to be allocated during later forward/backward passes.

2.2.5 Utilities and General Workflow

There are several utilities we have created for this framework, two of the most important ones are data loader and weight initialization.

Since we want to minimize the dependencies on other packages, we wrote our own data loader. Data loader read the pixel information from orignal CIFAR-10 dataset, normalize the data, and divide the data batches. The loader will load a random set of instances(images) from the training set, and the training will run multiple epochs to reach higher accuracy.

Weight initialization is also very important for training a deeper network. We used Kaiming uniform Initialization for fully connected layer and and Kaiming normal initialization for all convolution layers. More information about those initialization methods are available in [3, 5].

In each iteration, the following steps happens in order:

1. random training batch is fed to the bottom layer of the network;
2. Then forward functions are called one after another;
3. Loss layer calculated the gradient of the loss;
4. Gradient are propagated back in the reverse direction, gradient of all learnable parameters are updated;

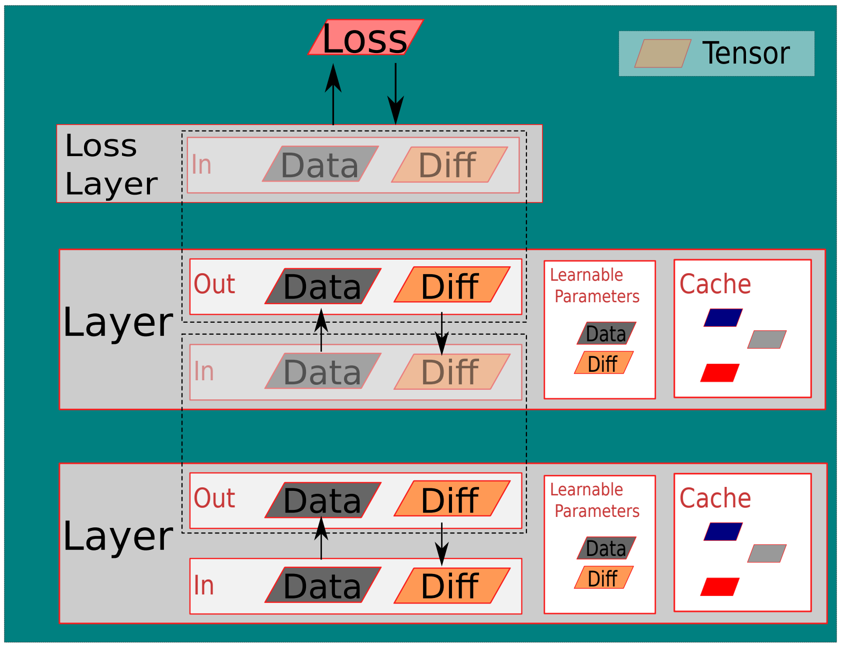


Figure 2: Management of memory in the network

5. Optimizer(e.g. SGD) updates current learnable parameters using the updated gradients.

3 GPU Kernel Implementation & Improvement

During the presentation we demonstrated device based convolution layers, and device based pooling. However since the presentation we have expanded our framework's device capabilities significantly. We have now completed device based forward and backward RELU, as well as forward and backward identity layers. Completing these layers allowed us to complete the formation of the entire residual block on the GPU. This gets us very close to completion of the entire ResNet entirely on the GPU. The remaining elements we need to port to device code are the loss layer, and SGD.

3.1 Harness Functions

Since the framework was originally built as a pure C host based implementation, we have made it possible to run CUDA with data that resides on the host as input, or with an input that resides on the device. To provide this functionality, we created harness functions which allow a user to call device kernels from host functions which are using host data host based data to the device, or to call kernels directly. This approach allows us to parameterize the kernel launches as well.

3.1.1 Host Harness

```
status_t <device_func_name>_host_harness(...);
```

Listing 4: naming convention for host harness

In listing 4, we see an example of the naming convention used in host harness functions. A host harness is called with host allocated memory. The function of a host harness is to allocate device memory, launch the CUDA kernel, and then copy the results back to the host. These functions are not meant to be efficient, but are provided for convenience and for testing purposes. Additionally, the ability to call the host harness functions allows us to easily form networks which selectively use host or device resources for any component we want; Although it is unlikely to be used this way.

```
status_t conv_forward_device_host_harness(cublasHandle_t handle,
                                          tensor_t h_x,
                                          tensor_t h_w,
                                          lcache_t* hcache,
                                          conv_param_t hparams,
                                          tensor_t h_y)
{
  tensor_t d_x = tensor_make_copy_h2d(h_x);
  tensor_t d_w = tensor_make_copy_h2d(h_w);
  tensor_t d_y = tensor_make_copy_h2d(h_y);

  // call device code
  conv_forward_device(handle, d_x, d_w, hcache, hparams, d_y);

  tensor_copy_d2h(h_x, d_x);
  tensor_copy_d2h(h_w, d_w);
  tensor_copy_d2h(h_y, d_y);

  // d_x is cached so don't destroy
  // d_w is cached so don't destroy
  tensor_destroy_device(&d_y);

  return S_OK;
}
```

Listing 5: full conv_forward_device_host_harness example

Listing 5 shows an example in which the function `convolution_forward_device(6)` is called via a host harness. The harness will allocate device memory, copy host memory to the device, call the device function, copy memory back to the host, and then deallocate the device memory. Note that in Listing 5, we make use of the `tensor_make_copy_h2d(1)`, and the `tensor_copy_d2h(2)` functions to enable us to copy memory to and from the device.

3.1.2 Device Harness

```
status_t <device_func_name>_device_harness(...);
```

Listing 6: naming convention for device harness

In listing 6, we see the structure for a device harness call. All a *_device_harness function does is wrap around the actual kernel launch. Shown in listing 7, the primary purpose of *_device_harness functions is to provide a simplified interface for the launch and keep the launch parameters out of the main code. The launch parameters from the kernel come from static variables in the definition files for the associated kernel. See the Launch Parameters section below for more information about the launch parameters.

```
void build_mask_device_harness(tensor_t d_a, tensor_t d_mask) {
  assert(d_a.mem_type == GPU_MEM);
  assert(d_mask.mem_type == GPU_MEM);

  build_mask_device<<<_build_mask_blocks, _build_mask_threads>>>(
    d_a, d_mask);
}
```

Listing 7: full conv_forward_device_host_harness example

3.2 Launch Parameters

Customizing launch parameters in our framework is possible for the CUDA kernels. To accomplish this, in the headers associated with device launches, we provide functions which allow a user to adjust the grid size (number of blocks) or the block size (number of threads per block). Each CUDA kernel can be configured with these functions. For the build_mask_device_harness in listing 7, we can see that the kernel launch "build_mask_device" is parameterized by the two variables "_build_mask_blocks", and "_build_mask_threads." These variables are static to the translation unit, and can be set through the functions mentioned above. Two overloads are provided for each *_device_harness function. For the build mast function, the functions headers are shown in listing 8. Note that their are overloads for both dim3 and int parameters. This allows users to call functions how they want. For 1D grids of 1D blocks, the convenience of the int overloads is very nice.

```
void set_build_mask_blocks(dim3 grid_dim);
void set_build_mask_threads(dim3 grid_dim);

void set_build_mask_blocks(int blocks);
void set_build_mask_threads(int threads);
```

Listing 8: setter functions for kernel launch

Note that in the above listing 8, the names of the setter functions follow the naming convention shown in listing 9.

```
void set_<operation_name>_blocks(1);
//and
void set_<operation_name>_threads(1);
```

Listing 9: setter functions for kernel launch

Also note that not all kernels in our system follow these standards yet. The ability to search kernel launch parameter sets is driving the need to expand the controls over these areas, and as such, the framework is actively being converted to the formats described in this section.

3.2.1 Parameter Search

Our device library provides a parameter search bench-marking function, which can be accessed through the "bench" folder in the project root. The parameter search currently only works on the convolution forward and backward operations. The header is shown below in listing 10 Running this function will take a long time depending on the number of parameters to be searched. The parameters to search can be modified by adjusting the containers shown in listing 11.

```
// the parameter search test header
TEST_F(TestLayerConvSpeed, bench_custom_forward_backward);
```

Listing 10: parameter search function header

```
// the parameter search test header
std::vector<int> block_arr = { 32, 64 };
std::vector<int> thread_arr = { 128, 256 };
```

Listing 11: important parameters for parameter search

3.3 General GPU Strategy

All of the CUDA code used in the forward, back, for both convolution and the formation of the residual blocks uses the same strategy. The general idea is to follow a multi-step process to factor 1D index into it's relevant components to enable access into some type of structure. We can then keep the actual data access call exactly the same as the CPU code, but we can gain the benefits of parallelism through the use of a CUDA "grid-strid-loop" Listing 12 shows a typical 2D looping pattern, which is used to index into a 1D structure. It is a very common pattern.

```
// a standard 2D loop to 1D access
for (int i = 0; i < num_row; ++i) {
    for (int j = 0; j < num_col; ++j) {
      cout << data[i * num_col + j] << " ";
    }
}
```

Listing 12: a typical 2D loop for accessing 2D data in a 1D structure

The way this pattern can be connected to the concept of a grid stride loop is to recognize that all grid stride loops consider elements in a 1D grid, and the participating threads reference into that grid through a 1D global index. Converting the above 2D loop into a 1D loop is easy, as shown below in listing 13.

```
// same access pattern, but we generate our indexes through
    factoring
for (int iter = 0; iter < num_row * num_col; ++iter) {
    int i = iter / num_col;
    int j = iter % num_col;

    cout << data[i * num_col + j] << " ";
}
```

Listing 13: a typical 2D loop for accessing 2D data in a 1D structure

Converting the above listing 13 into a grid-stride-loop is similarly trivial as shown in listing 14. The key to

```
__global__ f(float * data, int num_col, int num_row) {
    for (int iter = blockIdx.x * blockDim.x + threadIdx.x;
         iter < n;
         iter += blockDim.x * gridDim.x)
      {
          int i = iter / num_row;
          int j = iter / % num_col;
          cout << data[i * num_col + j] << " ";
      }
}
```

Listing 14: a raw grid stride loop showing same access as above, but on the GPU

The grid stride loop can be further simplified through the use of c++ 11 style range based for loops, and the use of iterators. The iterator pattern is powerful and beautiful and really makes the grid stride loop feel very similar to writing normal code. Additionally, grid stride loops are often very fast, and innately provide coelescetd access patterns. This is not always true though depending on the access pattern, and is frequently not true when you use the general technique being discussed here. However, this method is a great first step for a baseline in all kernels.

```
__global__ f(float * data, int num_col, int num_row) {
      for (int iter : grid_stride_range(0, num_col * num_row))
      {
          int i = iter / num_row;
          int j = iter / % num_col;
          cout << data[i * num_col + j] << " ";
      }
}
```

Listing 15: using c++ style iterators to perform the striding operation for you

3.4 Convolution Forward

The convolution forward function uses five GPU kernels. They perform the following responsibilities.

- Tensor transpose 3012 (custom)

- Batched add padding (custom)
- Image to column operation (custom v1 and v2)
- GEMM (used cublas)
- 2D Matrix transpose (used cublas)

The 3 custom operations all follow the grid stride approach described above. I will briefly discuss the custom operations below.

3.4.1 Transpose 3012

The transpose operation was a very interesting one, not because it was a particularly difficult operation to perform, but because after experimenting with it a bit, I found out that you could collapse the operation from a 4D approach, as is typical in tensor transpositions, to a 2D operation. This was accomplished on the host side before it was done on the GPU. The GPU code is shown in listing 16.

```
static __global__ void _do_tensor_make_transpose_3012_device(
    tensor_t d_transpose, tensor_t d_src) {
  uint n = d_capacity(d_src);
  uint group_size = d_src.dim.dims[0] * d_src.dim.dims[1] * d_src.
    dim.dims[2];
  uint stride = d_src.dim.dims[3];

  for (auto i : grid_stride_range(0u, n)) {
    uint src_idx = i / group_size + (i % group_size) * stride;
    d_transpose.data[i] = d_src.data[src_idx];
  }
}
```

Listing 16: transpose 3012 is actually a rotation of dimension

Note the inner loop in listing 16. The src_idx is calculated based on what appears to be a 1D access, but it is using the 2D access pattern to do so. Finally one more interesting point about the grid stride loop here is that the writes are coalesced, while the reads are not.

3.4.2 Batched Add Padding

The batched add padding operation was a difficult one. This was due to complex mappings in both the source and the target data locations. For brevity, I will omit the source code of padding operation from this paper. One important thing to note is that the padding operation is actually not necessary. The im2col (image to column) operation can actually be reformulated to handle the padded elements directly. Reformulating the image to column operation has two significant consequences. First, the memory needed to allocate the additional padded tensor is saved, which can be very important on the GPU, especially when large networks are concerned. Second, it moves the un-coalesced reads and writes from this operation to the image to column operation.

The importance of that within this context is that the complexity of the image to column operation is already extremely high. Adding the additional complexity that handling the padding incurs on the image to column operation is something that we chose to avoid in this implementation.

3.4.3 Image To Column

Image to column (and it's column to image counterpart) are two of the most complex operations that we accomplished in this project. The source code of the GPU implementation is below in listing 17. While this operation appears relatively simple, it took a lot to get to the point where we could verify its correctness.

```
static __global__ void
    _do_im2col_inner_device_naive_thread_per_filter(
    tensor_t cols, tensor_t x_padded, uint N, uint C, uint H, uint
   W, uint HH,
    uint WW, uint fltr_height, uint fltr_width, uint padding, uint
    stride)
{
  uint cols_d1   = cols.dim.dims[1];
  uint img_sz    = C * x_padded.dim.dims[2] * x_padded.dim.dims[3];
  uint chan_sz   = x_padded.dim.dims[2] * x_padded.dim.dims[3];
  uint row_sz    = x_padded.dim.dims[2];

  uint filters_per_channel  = HH * WW;
  uint filters_per_image    = C * filters_per_channel;
  uint total_filters        = N * filters_per_image;

  for (auto iter : grid_stride_range(0u, total_filters)) {

    uint n = iter / filters_per_image;
    uint c = (iter / filters_per_channel) % C;
    uint j = (iter / WW) % HH;
    uint k = (iter % WW);

    for (uint f_row = 0; f_row < fltr_height; ++f_row) {
      for (uint f_col = 0; f_col < fltr_width; ++f_col) {
        uint row = c * fltr_width * fltr_height + f_row *
    fltr_width + f_col;
        uint col = j * WW * N + k * N + n;
        uint target_idx = row * cols_d1 + col;
        uint src_idx = (n * img_sz) + (c * chan_sz) + (stride * j +
     f_row) * row_sz + stride * k + f_col;

        cols.data[target_idx] = x_padded.data[src_idx];
      }
    }
  }
}
```

Listing 17: Image to column version 1 (thread per filter) operation using grid stride loop

Additionally, this operation assigns one thread to each "filter" location in the src "x_padded" array. This has some interesting effects. Generally the idea is that if the filters are small, each thread will do very little work. If the filters are large, the amount of work done by each thread will be very large. This will cause a limited degree of parallelism give than we have a small number of filters.

In order to improve upon this ”thread per filter” approach, we chose to implement an even more complicated approach which engages one ”thread per element” of the target output array. The first consequence of this is that we now get a massively improvement on the amount of parallelism we engage. Listing 18 shows the improved function.

```
static __global__ void _do_im2col_inner_device_naive_thread_per_filter(
    tensor_t cols, tensor_t x_padded, uint N, uint C, uint H, uint W, uint HH,
    uint WW, uint fltr_height, uint fltr_width, uint padding, uint stride)
{
  uint cols_d1  = cols.dim.dims[1];
  uint img_sz   = C * x_padded.dim.dims[2] * x_padded.dim.dims[3];
  uint chan_sz  = x_padded.dim.dims[2] * x_padded.dim.dims[3];
  uint row_sz   = x_padded.dim.dims[2];

  uint filters_per_channel  = HH * WW;
  uint filters_per_image    = C * filters_per_channel;
  uint total_filters        = N * filters_per_image;

  for (auto iter : grid_stride_range(0u, total_filters)) {

    uint n = iter / filters_per_image;
    uint c = (iter / filters_per_channel) % C;
    uint j = (iter / WW) % HH;
    uint k = (iter % WW);

    for (uint f_row = 0; f_row < fltr_height; ++f_row) {
      for (uint f_col = 0; f_col < fltr_width; ++f_col) {
        uint row = c * fltr_width * fltr_height + f_row * fltr_width + f_col;
        uint col = j * WW * N + k * N + n;
        uint target_idx = row * cols_d1 + col;
        uint src_idx = (n * img_sz) + (c * chan_sz) + (stride * j + f_row) * row_sz + stride * k + f_col;

        cols.data[target_idx] = x_padded.data[src_idx];
      }
    }
  }
}
```

Listing 18: Image to column version 2 (thread per element) operation using grid stride loop

3.5 Convolution Backward

The convolution backward operation is nearly the direct opposite of the forward operation. The general idea is that we are going to unwrap the flattened tensors, calculate some derivatives, and update some tensors.

- Tensor transpose 1230 (custom)
- Batched remove padding (custom)
- Column to image operation (custom v1 and v2)
- GEMM (used cublas)
- 2D Matrix transpose (used cublas)

Showing the transpose and padding removal operations here is a bit redundant, since they are so similar to the forward. The transpose is a little interesting because one consequence of the access pattern is that the coalescence shifts from the write side to the read side.

However, the col2im operation is significantly different. Because the im2col can be thought of as duplicating portions of the input space, the col2im is responsible for recombining the duplicated space. As such, we have to add a atomicAdd function. Listing 19 shows the "thread per element" version of col2im. The "thread per filter" version is available as well, but is left out for brevity.

```
static __global__ void _do_col2im_inner_device_thread_per_element(
    tensor_t d_cols, tensor_t x_padded, uint N, uint C, uint H,
    uint W, uint HH,
    uint WW, uint field_height, uint field_width, uint padding,
    uint stride)
{
  uint dx_col_d1   = d_cols.dim.dims[1];
  uint x_p_d1      = x_padded.dim.dims[1];
  uint x_p_d2      = x_padded.dim.dims[2];
  uint x_p_d3      = x_padded.dim.dims[3];

  uint C_fh_fw_HH_WW = C * field_height * field_width * HH * WW;

  for (auto iter : grid_stride_range(0u, N * C * H * W *
    field_height * field_width)) {

    uint i = iter / C_fh_fw_HH_WW;  // ii is the target image
    uint c = (iter / (field_height * field_width * HH * WW)) % C;
    // jj is the channel in the image
    uint fi = iter / (HH * WW * field_width) % field_height;
    uint fj = (iter / (HH * WW)) % field_width;
    uint h = (iter / WW) % HH;
    uint w = iter % WW;

    uint row = c * field_width * field_height + fi * field_width +
    fj;

    uint col = h * WW * N + w * N + i;
    uint src_idx = row * dx_col_d1 + col;
    uint target_idx =
        i * x_p_d1 * x_p_d2 * x_p_d3
        + c * x_p_d2 * x_p_d3
        + (stride * h + fi) * x_p_d3
        + stride * w + fj;

    atomicAdd(&(x_padded.data[target_idx]), d_cols.data[src_idx]);
  }
}
```
Listing 19: Column to image version 2 (thread per element) operation using grid stride loop

3.6 Global Pooling Layer

In ResNet, we need to implement a global average pooling at the last layer, which downsamples each channel into one value([N,C,H,W] -¿ [N,C,1,1]). Listing 20 and 21 shows how we implement the global pooling on GPU.

```
static __global__ void _do_forward(T *x, uint num_image, uint
    num_channel, uint channel_capacity, T *y) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < num_image * num_channel) {
    T mean = 0;
    T *channel_start = x + idx * channel_capacity;
    for (uint i = 0; i < channel_capacity; i++) {
      mean += channel_start[i];
    }
    mean /= channel_capacity;
    y[idx] = mean;
  }
}
```
Listing 20: forward_global_pooling kernel

```
static __global__ void _do_backward(T *dx, uint num_image, uint
    num_channel, uint channel_capacity, T *dy) {
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < num_image * num_channel) {
    T scale_by = 1.0 / (channel_capacity);
    T *channel_start = dx + idx * channel_capacity;
    for (uint i = 0; i < channel_capacity; i++) {
      channel_start[i] = scale_by * dy[idx];
    }
  }
}
```
Listing 21: backward_global_pooling kernel

4 cuDNN Convolution Layer Implementation & Improvement

In this section, we introduce how we utilize NVIDIA CUDA Deep Neural Network library (cuDNN) to implement the forward and backward convolution layers.

4.1 Big Picture of cuDNN Workflow

In general, it takes 4 steps to implement a convolution layer (or other layers) using cuDNN. Figure 3 shows that the first step is to initialize the cuDNN library and create the descriptors for the input, filter, output and convolution layer. The second step is to set the descriptors. The third step is to allocate work space on GPU memory. The fourth step is to call the forward and backward convolution layers. At last, we call synchronization and copy device memory back to host.

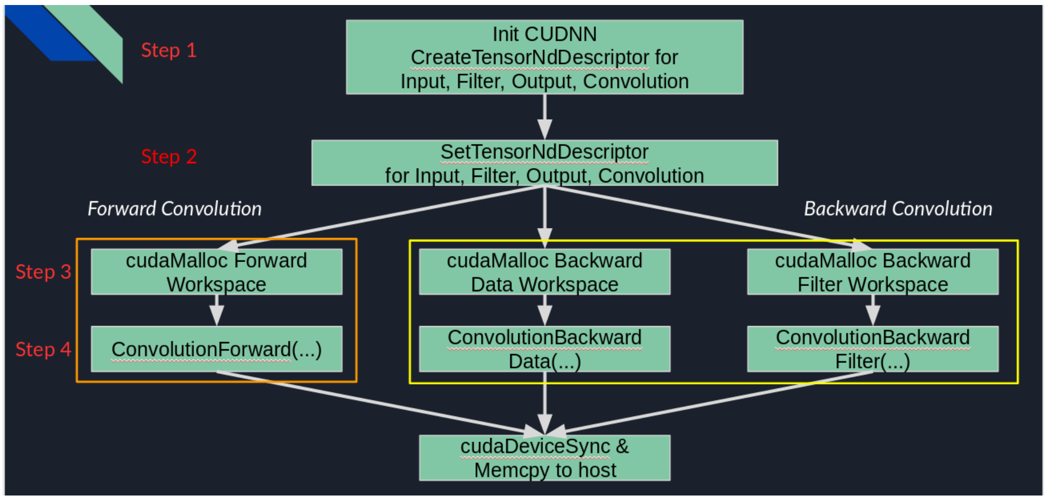

Step 1
Init CUDNN
CreateTensorNdDescriptor for
Input, Filter, Output, Convolution
Step 2
SetTensorNdDescriptor
for Input, Filter, Output, Convolution
Forward Convolution
Backward Convolution
Step 3
cudaMalloc Forward Workspace
cudaMalloc Backward Data Workspace
cudaMalloc Backward Filter Workspace
Step 4
ConvolutionForward(...)
ConvolutionBackward Data(...)
ConvolutionBackward Filter(...)
cudaDeviceSync &
Memcpy to host

Figure 3: cuDNN workflow big picture

4.1.1 Interface to awnn Framework

Figure 4 is conceptual convolution layer, which shows that x and dx (gradient of x) are input data, w (weight) and dw (gradient of w) are filters, and y and dy (gradient of y) are output. This also comply with the parameters in all later cuDNN functions. Below is our convolution layer interface. We will describe

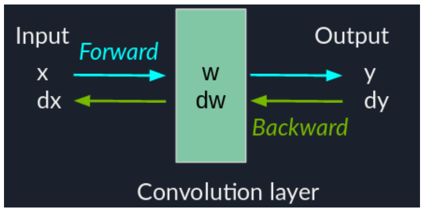


Figure 4: conceptual convolution layer

them in the following.

```
status_t convolution_forward_cudnn(
    tensor_t const x, tensor_t const w,
    lcache_t* cache, conv_param_t const params, tensor_t y,
    cudnnHandle_t handle_, cudnnTensorDescriptor_t cudnnIdesc,
    cudnnFilterDescriptor_t cudnnFdesc,
    cudnnTensorDescriptor_t cudnnOdesc,
    cudnnConvolutionDescriptor_t cudnnConvDesc);

status_t convolution_backward_cudnn(
    tensor_t dx, tensor_t dw, lcache_t* cache,
    conv_param_t const params, tensor_t const dout,
    cudnnHandle_t handle_, cudnnTensorDescriptor_t cudnnIdesc,
    cudnnFilterDescriptor_t cudnnFdesc,
    cudnnTensorDescriptor_t cudnnOdesc,
    cudnnConvolutionDescriptor_t cudnnConvDesc);
```

Listing 22: Our cuDNN interfaces for forward and backward convolution

4.2 Create Descriptors

Before we create the four descriptors, we need to initialize the cuDNN library.

```
cudnnStatus_t cudnnCreate(cudnnHandle_t *handle)
```

Listing 23: Initialize cuDNN library

This function creates a handle to an opaque structure holding the cuDNN library context. It allocates hardware resources on the host and device and must be called prior to making any other cuDNN library calls.

Then we use these three function to create a descriptor object by allocating the memory needed to hold its opaque structure.

```
cudnnStatus_t cudnnCreateTensorDescriptor(
                cudnnTensorDescriptor_t *tensorDesc)

cudnnStatus_t cudnnCreateFilterDescriptor(
                cudnnFilterDescriptor_t *filterDesc)

cudnnStatus_t cudnnCreateConvolutionDescriptor(
                cudnnConvolutionDescriptor_t *convDesc)
```

Listing 24: Create Tensor Filter and Convolution Descriptors

Here are my implementation code. Note after completing all computation, we also need to clean the memory to avoid memory leak.

```
cudnnHandle_t handle_;
cudnnTensorDescriptor_t cudnnIdesc;
cudnnFilterDescriptor_t cudnnFdesc;
cudnnTensorDescriptor_t cudnnOdesc;
cudnnConvolutionDescriptor_t cudnnConvDesc;
// Create Descriptors
checkCudnnErr( cudnnCreate(&handle_));
checkCudnnErr( cudnnCreateTensorDescriptor( &cudnnIdesc ));
checkCudnnErr( cudnnCreateFilterDescriptor( &cudnnFdesc ));
checkCudnnErr( cudnnCreateTensorDescriptor( &cudnnOdesc ));
checkCudnnErr( cudnnCreateConvolutionDescriptor( &cudnnConvDesc ));
```

```
// do something ...
clean:
if (cudnnIdesc) cudnnDestroyTensorDescriptor(cudnnIdesc);
if (cudnnFdesc) cudnnDestroyFilterDescriptor(cudnnFdesc);
if (cudnnOdesc) cudnnDestroyTensorDescriptor(cudnnOdesc);
if (cudnnConvDesc)cudnnDestroyConvolutionDescriptor(cudnnConvDesc);
if (handle_) cudnnDestroy(handle_);
```

Listing 25: Initialization Implementation

4.3 Set Descriptors

4.3.1 Set 4d Tensors Descriptors Using 4d Interfaces

Image Batches are described as 4D Tensor [n, c, h, w] with stride support as shown in Figure5. n is number of images, c is channels of each image, and h and w are the number of row and columns of each image.

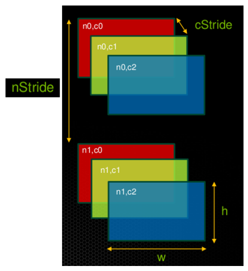


Figure 5: 4d Tensor

Listing 26 is the interface to set 4d tensor. We set `format = CUDNN_TENSOR_NCHW`. There are other options, such as `CUDNN_TENSOR_NHWC`, which is limited support in later convolution operation. `CUDNN_TENSOR_NCHW_VECT_C` is another option, which each element of the tensor is a vector of multiple feature maps. We set `dataType = CUDNN_DATA_FLOAT`. As cuDNN documentation stated, you can also set `CUDNN_DATA_DOUBLE`, but we tried, and it gives 0 output in forward convolution.

```
cudnnStatus_t cudnnSetTensor4dDescriptor(
    cudnnTensorDescriptor_t tensorDesc,
    cudnnTensorFormat_t     format,
    cudnnDataType_t         dataType,
    int                     n,
    int                     c,
    int                     h,
    int                     w)
```

Listing 26: cudnn Set 4d Tensor Descriptor

Similarly, Listing 27 shows the interface to set the 4d filter. k is the number of filters, c is the channel of each filter, and h and w are the number of filter row and columns.

```
cudnnStatus_t cudnnSetFilter4dDescriptor(
    cudnnFilterDescriptor_t     filterDesc,
    cudnnDataType_t             dataType,
    cudnnTensorFormat_t         format,
    int                         k,
    int                         c,
    int                         h,
    int                         w)
```

Listing 27: cudnn Set 4d Filter Descriptor

Listing 28 shows how to set the 2d convolution descriptor. u is vertical filter stride, v is horizontal filter stride. Note for the **mode** setting. You can choose either `mode = CUDNN_CONVOLUTION` or `CUDNN_CROSS_CORRELATION`. The original math definition of convolution is to flip the filter by 180 degree if setting `CUDNN_CONVOLUTION`. But in current AI theory, without 180 degree flip up, `CUDNN_CROSS_CORRELATION` is what they so called "convolution". And this is a bug which we debugged for a long time. Since it can pass the verification code from cuDNN sample code, but it did not match the expected value lists given by numpy. For `dilation_h = 1.0, dilation_w = 0.0`. We will talk the last two parameters later. The reason why tjhis function is called 2d convolution is that we are cross-correlation computing 2d filter with 2d data input.

```
cudnnStatus_t cudnnSetConvolution2dDescriptor(
    cudnnConvolutionDescriptor_t    convDesc,
    int                             pad_h,
    int                             pad_w,
    int                             u,
    int                             v,
    int                             dilation_h,
    int                             dilation_w,
    cudnnConvolutionMode_t          mode,
    cudnnDataType_t                 computeType)
```

Listing 28: cudnn Set 2d Tensor Descriptor

4.3.2 Set Descriptors using Nd interfaces

Similarly, we can also use Nd tensor/filter/convolution interface to set descriptors as shown in Listing 29. Since in our final implemenation, we follow the NCHW formate, thus we set `nbDims = 4`, `int dimA[] =n, c, h, w`. But the critical part in the Nd tensor descriptor is to generate the correct `strideA[]`.

```
cudnnStatus_t cudnnSetTensorNdDescriptor(
    cudnnTensorDescriptor_t tensorDesc,
    cudnnDataType_t         dataType,
    int                     nbDims,
    const int               dimA[],
    const int               strideA[])
```

Listing 29: cudnn Set Nd Tensor Descriptor

```
static void generateStrides(const int* dimA, int* strideA, int
    nbDims, cudnnTensorFormat_t filterFormat) {
```

```
  if (filterFormat == CUDNN_TENSOR_NCHW) {
    strideA[nbDims-1] = 1 ;
    for(int d = nbDims-2 ; d >= 0 ; d--) {
      strideA[d] = strideA[d+1] * dimA[d+1] ;
    }
  }
}
```

Listing 30:]generate the correct strideA[]

For NCHW data, after call `generateStrides`, we get the following strideA information.

```
StrideA[3] = 1
StrideA[2] = dimA[3] = W
StrideA[1] = dim[2] * strideA[2] = dimA[2] * dimA[3] = H * W
StrideA[0] = dim[1] * strideA[1] = dimA[1] * dimA[2] * dimA[3] = N
    * H * W
```

Listing 31:]generate the correct strideA[]

```
cudnnStatus_t cudnnSetFilterNdDescriptor(
    cudnnFilterDescriptor_t filterDesc,
    cudnnDataType_t         dataType,
    cudnnTensorFormat_t     format,
    int                     nbDims,
    const int               filterDimA[])
```

Listing 32: cudnn Set Nd Filter Descriptor

Let's talk about the dilation here. As cuDNN documentation, $outputDim = 1+(inputDim+2*pad-(((filterDim-1)*dilation)+1))/convolutionStride$. Thus it will scale the output dimention. Also note to set `CUDNN_CROSS_CORRELATION` here.

```
cudnnStatus_t cudnnSetConvolutionNdDescriptor(
    cudnnConvolutionDescriptor_t    convDesc,
    int                             arrayLength,
    const int                       padA[],
    const int                       filterStrideA[],
    const int                       dilationA[],
    cudnnConvolutionMode_t          mode,
    cudnnDataType_t                 dataType)
```

Listing 33: cudnn Set Nd Convolution Descriptor

4.4 Convolution

4.4.1 Allocate Work Space Size

Allocate work space size is straightforward, only needs to note that they are temporary buffers to compute the convolution layer. If no longer needed, free them.

```
// Allocate forward workspace
checkCudnnErr ( cudnnGetConvolutionForwardWorkspaceSize(
        handle_, cudnnIdesc, cudnnFdesc, cudnnConvDesc,
        cudnnOdesc, algo, &workSpaceSize) );
if (workSpaceSize > 0) {
    cudaMalloc(&workSpace, workSpaceSize);
}
// Allocate backward data workspace
checkCudnnErr ( cudnnGetConvolutionBackwardDataWorkspaceSize(
        handle_, cudnnFdesc, cudnnOdesc, cudnnConvDesc,
        cudnnIdesc, algo_data, &workSpaceSize) );
if (workSpaceSize > 0) {
    cudaMalloc(&workSpace, workSpaceSize);
}
// Allocate backward filter workspace
checkCudnnErr ( cudnnGetConvolutionBackwardFilterWorkspaceSize(
        handle_, cudnnIdesc, cudnnOdesc, cudnnConvDesc,
        cudnnFdesc, algo_weight, &workSpaceSize) );
if (workSpaceSize > 0) {
    cudaMalloc(&workSpace, workSpaceSize);
}

if (workSpace) cudaFree(workSpace);
```

Listing 34: get work space size and cudaMalloc them

4.4.2 Forward Convolution

```
cudnnStatus_t cudnnConvolutionForward(
    cudnnHandle_t                         handle,
    const void                           *alpha,
    const cudnnTensorDescriptor_t         xDesc,
    const void                           *x,
    const cudnnFilterDescriptor_t         wDesc,
    const void                           *w,
    const cudnnConvolutionDescriptor_t    convDesc,
    cudnnConvolutionFwdAlgo_t             algo,
    void                                 *workSpace,
    size_t                                workSpaceSizeInBytes,
    const void                           *beta,
    const cudnnTensorDescriptor_t         yDesc,
    void                                 *y)
```

Listing 35: cudnn forward convolution interface

This function executes convolutions or cross-correlations over x using filters specified with w, returning results in y. Scaling factors alpha and beta can be used to scale the input tensor and the output tensor respectively.

For the `algo`, we set to `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_RECOMP_GEMM`, which expresses the convolution as a matrix product without actually explicitly form the matrix that holds the input tensor data, but still needs some memory workspace to precompute some indices in order to facilitate the implicit construction of the matrix that holds the input tensor data. Other options, e.g., `CUDNN_CONVOLUTION_FWD_ALGO_GEMM`, which significant workspace may be needed. In our final experiment, we also set algo with the other values:

- `CUDNN_CONVOLUTION_FWD_ALGO_FFT` (uses the Fast-Fourier Transform approach to compute the convolution),
- `CUDNN_CONVOLUTION_FWD_ALGO_FFT_TILING` (use fft, but splits the inputs into tiles. A significant memory workspace is needed to store intermediate results but less than `CUDNN_CONVOLUTION_FWD_ALGO_FFT` for large size images),
- `CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD` (uses the Winograd Transform),
- `CUDNN_CONVOLUTION_FWD_ALGO_WINOGRAD_NONFUSED` Since this algorithm needs significant workspace , we don't use this one.

4.4.3 Backward Data

```
cudnnStatus_t cudnnConvolutionBackwardData(
    cudnnHandle_t                        handle,
    const void                          *alpha,
    const cudnnFilterDescriptor_t        wDesc,
    const void                          *w,
    const cudnnTensorDescriptor_t        dyDesc,
    const void                          *dy,
    const cudnnConvolutionDescriptor_t   convDesc,
    cudnnConvolutionBwdDataAlgo_t        algo,
    void                                *workSpace,
    size_t                               workSpaceSizeInBytes,
    const void                          *beta,
    const cudnnTensorDescriptor_t        dxDesc,
    void                                *dx)
```

Listing 36: cudnn backward data interface

This function computes the convolution data gradient of the tensor dy, where y is the output of the forward convolution in `cudnnConvolutionForward()`. It uses the specified algo, and returns the results in the output tensor dx. Scaling factors `alpha` and `beta` can be used to scale the computed result or accumulate with the current dx. For algo, we set `CUDNN_CONVOLUTION_BWD_DATA_ALGO_1`, since this algorithm expresses the convolution as a matrix product without actually explicitly form the matrix that holds the input tensor data. The results are deterministic. We also use other option, e.g., `CUDNN_CONVOLUTION_BWD_DATA_ALGO_FFT`, which this algorithm uses a Fast-Fourier Transform approach to compute the convolution. A significant memory workspace is needed to store intermediate results. The results are deterministic. Unfortunately, the algorithms WINOGRAD in backward has compile error, although we have already used the latest cuDNN and CUDA library and driver.

4.4.4 Backward Filter

```
cudnnStatus_t cudnnConvolutionBackwardFilter(
    cudnnHandle_t                        handle,
    const void                          *alpha,
    const cudnnTensorDescriptor_t        xDesc,
    const void                          *x,
    const cudnnTensorDescriptor_t       dyDesc,
    const void                          *dy,
    const cudnnConvolutionDescriptor_t  convDesc,
    cudnnConvolutionBwdFilterAlgo_t     algo,
    void                                *workSpace,
    size_t                              workSpaceSizeInBytes,
    const void                          *beta,
    const cudnnFilterDescriptor_t       dwDesc,
    void                                *dw)
```

Listing 37: cudnn backward filter interface

This function computes the convolution weight (filter) gradient of the tensor dy, where y is the output of the forward convolution in `cudnnConvolutionForward()`. It uses the specified algo, and returns the results in the output tensor dw. Scaling factors alpha and betacan be used to scale the computed result or accumulate with the current dw. Similarly, we use `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_1` which expresses the convolution as a matrix product without actually explicitly form the matrix that holds the input tensor data. The results are deterministic and Other option we used `CUDNN_CONVOLUTION_BWD_FILTER_ALGO_FFT` which This algorithm uses the Fast-Fourier Transform approach to compute the convolution. Significant workspace is needed to store intermediate results. The results are deterministic.

4.5 Verification

We use google test module to verify every function correctness and enhance our project test ability. For example, we use an input tensor with dimension $2 \times 3 \times 4 \times 4$ and filter $3 \times 3 \times 3 \times 3$. We use numpy as the expected results to compare with

4.5.1 Verify Forward Results

```
// device copy back to host
tensor_copy_d2h(y, d_y);
// make a same output size y_ref
tensor_t y_ref = tensor_make_alike(y);
// fill y_ref with expected value from numpy code
double value_list[] = {
    0.02553947, 0.03144079, 0.01900658, ... , 0.99532895};
tensor_fill_list(y_ref, value_list, array_size(value_list));
// compute relative error, expect less than 1^10-7
T rel_err = tensor_rel_error(y_ref, y);
EXPECT_LT(rel_err, 1e-7);
PINF("Cudnn_forward Consistent with expected results");
```

Listing 38: cudnn forward verification

The relative error of forward convolution is less than 1×10^{-7}.

4.5.2 Verify Backward Data & Weight Results

We use the numerical check to verify the accuracy of the gradient of x and weight during backward propagation. The relative error of the gradient of input (dx) is 5.94×10^{-5}, while the gradient of weight (dw) is 1.36×10^{-5}.

```cpp
// evaluate gradient of x
eval_numerical_gradient(
    [&](tensor_t const in, tensor_t out) {
      convolution_forward(in, w_copy, nullptr, conv_params, out);
    },
    x, dy, dx_ref);
EXPECT_LT(tensor_rel_error(dx_ref, dx), 1e-4);
PINF("cudnn gradient check of x... is ok");
// evaluate gradient of w
eval_numerical_gradient(
    [&](tensor_t const in, tensor_t out) {
      convolution_forward(x_copy, in, nullptr, conv_params, out);
    },
    w, dy, dw_ref);
EXPECT_LT(tensor_rel_error(dw_ref, dw), 1e-4);
PINF("cudnn gradient check of w... is ok");
```

Listing 39: numerical check written by λ closure in C++

4.6 Performance improvement for bench test

To improvement the performance, we need to move the creation and destroy of input, output, filter and convolution descriptor outside the testing loop. Thus we can avoid the frequently allocate memory operation as shown in Figure 6. After

5 Experiment & Performance Evaluation

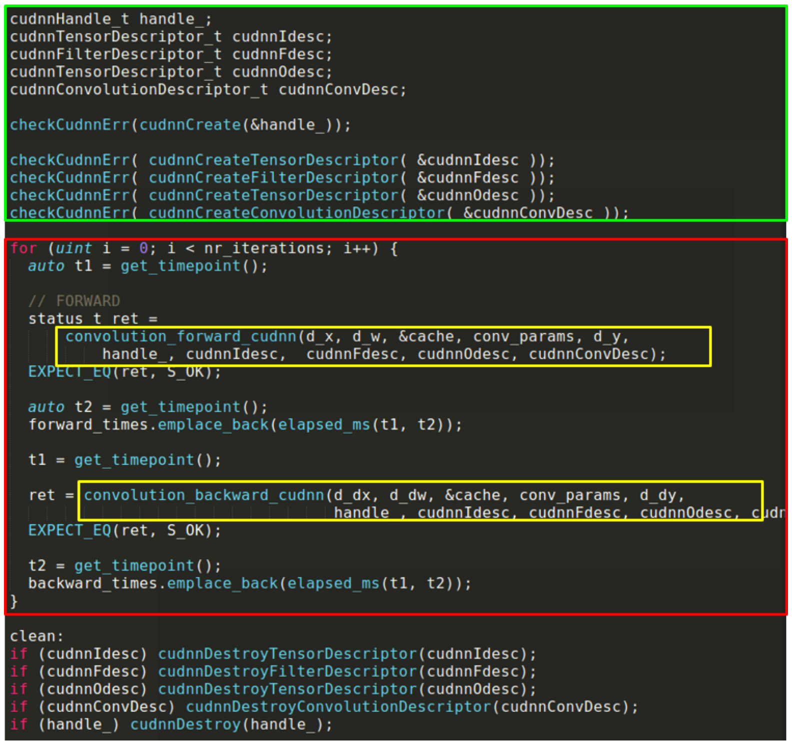
5.1 Experiment Setup

We design three groups of experiments: small, medium and large batches of images to test the forward and backward convolution. The specific parameters settings are as shown in Table 1. We compare our custom implementation with cuDNN with different algorithms. For our custom implementation, custom v1 uses thread per filter while custom v2 uses thread per element. For cuDNN algorithms, we choose Implicit_GEMM, FFT, FFT tiling and Winograd to test cuDNN forward convolution. Then for cuDNN backward convolution algorithm, we use the Implicit_GEMM and FFT. We run the following configurations on all implementation with 100 iterations. Then we calculate the average time for each call. The data type is single floating point.

5.2 Performance Evaluation

Figure 7 shows the final results of using different algorithms and implementation. The top axis is the average time per iteration. The left axis is the batch setting, i.e., number of images and number of filters.

Top three groups are using only 1 image(N=1 and F=1,4,16) We can find that our custom v2 is always faster than v1. This is because in the v1 version, each thread has to be responsible for an entire filter, while in v2 each thread



```
cudnnHandle_t handle_;
cudnnTensorDescriptor_t cudnnIdesc;
cudnnFilterDescriptor_t cudnnFdesc;
cudnnTensorDescriptor_t cudnnOdesc;
cudnnConvolutionDescriptor_t cudnnConvDesc;

checkCudnnErr(cudnnCreate(&handle_));

checkCudnnErr( cudnnCreateTensorDescriptor( &cudnnIdesc ));
checkCudnnErr( cudnnCreateFilterDescriptor( &cudnnFdesc ));
checkCudnnErr( cudnnCreateTensorDescriptor( &cudnnOdesc ));
checkCudnnErr( cudnnCreateConvolutionDescriptor( &cudnnConvDesc ));

for (uint i = 0; i < nr_iterations; i++) {
  auto t1 = get_timepoint();

  // FORWARD
  status_t ret =
      convolution_forward_cudnn(d_x, d_w, &cache, conv_params, d_y,
          handle_, cudnnIdesc,  cudnnFdesc, cudnnOdesc, cudnnConvDesc);
  EXPECT_EQ(ret, S_OK);

  auto t2 = get_timepoint();
  forward_times.emplace_back(elapsed_ms(t1, t2));

  t1 = get_timepoint();

  ret = convolution_backward_cudnn(d_dx, d_dw, &cache, conv_params, d_dy,
                                   handle_, cudnnIdesc, cudnnFdesc, cudnnOdesc, cudn
  EXPECT_EQ(ret, S_OK);

  t2 = get_timepoint();
  backward_times.emplace_back(elapsed_ms(t1, t2));
}

clean:
if (cudnnIdesc) cudnnDestroyTensorDescriptor(cudnnIdesc);
if (cudnnFdesc) cudnnDestroyFilterDescriptor(cudnnFdesc);
if (cudnnOdesc) cudnnDestroyTensorDescriptor(cudnnOdesc);
if (cudnnConvDesc) cudnnDestroyConvolutionDescriptor(cudnnConvDesc);
if (handle_) cudnnDestroy(handle_);
```

Figure 6: improvement of bench test for cuDNN

only calculates 1 element. If we increased the size of the filters in our tests, we would see an even bigger performance boost, but our filters are only 3×3 so that means each thread is handling only 9 elements. Further testing is planned using the parameter search implementation, and additionally with doubles. In our current results however, we don't see a huge performance increase with thread per element due to the limited size of the filters. If using FFT for forward and backward, they take longer time than the GEMM version. This is because the filter size of our experiment are small (only 3×3 in this case). FFT performs better performance when filter size is larger.[1] Another interesting thing to note is that the backward FFT is longer than the forward FFT time. Thus in later experiment Winograd performs nearly comparable with GEMM, which is the the best performance in all cases.

Table 1: Experimental setup of convolution parameters.

nr_imgs	nr_input_channel	input h/w	nr_output_channel	kernel h/w	pad	stride
1	4	32	1	3	1	1
1	4	32	4	3	1	1
1	4	32	16	3	1	1
4	4	32	1	3	1	1
4	4	32	4	3	1	1
4	4	32	16	3	1	1
16	4	32	1	3	1	1
16	4	32	4	3	1	1
16	4	32	16	3	1	1

For the medium case(N=4 and F=1,4,16), we find that the time of both our custom implementation has increased. While all cuDNN version remains almost the same as when N=1 cases.

For the large case(N=16 and F=1,4,16), we find that except for GEMM and Winograd, all other setting have increase a lot. FFT is slower than gem by at most 9.68X, FFT with tiling is slower than GEMM by at most 11.8X. Winograd takes 1.6X longer time than GEMM. Our custom v2 is 7.96X slower than GEMM.

6 Future work

Currently, we have a number of important enhancements and applications planned for the framework. Since the framework is easy to extend and work in, we intend to create a number of different architectures with it. We plan on testing multiple feed forward residuals within the context of Resnet, and far more. Having an easy to use platform for experimentation in network design is a big step toward being able to carry out research into new neural network architectures.

Before this happens though, we need to spend more time refining the test platform surrounding a few core elements. As an example, currently in order to plug in different kernels into the core of some of the important inner functions (such as col2im and im2col), we actually have to manually change the interior of the functions. This is untenable in the long run. After learning how the cuDNN interface works, we noticed that they add a parameter in their setup functions which allows them to specify the kernel they want to use. We intend to modify our interface to have similar functionality.

Another major change that we may experiment with is extending the kernel parameterization functionality. The ability to control the parameters in the core of the kernels that we have now is nice, but it is not complete, even for the basic grid sizes and block sizes. Currently only about half the framework has the upgrades which enhance the parameterization capabilities via _harness functions. Additionally, our framework does not have the capability of allowing a user to easily adjust shared memory or stream usage. Although this has not been a problem within the context of our current implementation, as we add functionality and start to manage the network more deeply, the necessity for controlling stream on a more granular scale increases. This will especially become true as we add the a more complex buffering strategy for device memory.

Currently, we have shown an example of a full residual block on the device. This is a significant step towards achieving a full implementation of Resnet. However, one major hurdle remains that prevents us from massively extending the network. The limitation we face is due to the size of the device memory. After stacking a few residual blocks, we quickly run out of memory. Additionally, the entire system needs to account for the memory required by training batches, which we would like to make as large as possible. In order to actually implement the entire Resnet on the GPU we will need to eventually build a pipelining system into the system which has the ability to dynamically buffer appropriate data in the GPU while transferring unused data, or data that won't be used soon back to the CPU.

7 Conclusion

In this work, we have implemented the network of ResNet and provide efficient computation kernels using CUDA. We also compared our CUDA implementation with cuDNN and the best cuDNN implementation can achieve at most 12X faster than other algorithms in our experiments. We have a lot of places we can improve our own custom implementation however.

From we created a major framework which was a significant engineering task. For the GPU, we learned and implemented a generalized approach which seems to be an excellent baseline and first step for almost any CPU function which uses for loops. In some cases, this approach seems to work very well and is able to hit the theoretical maximum for an algorithm. In other cases, it provides a faster than worse case naive benchmark. We used this method in a variety of ways and tested upgraded versions. We learned that the generalized approach to device programming tends to suffer when shared memory is required and localization is difficult.

Further we used a number of important engineering tools that furthered our engineering abilities. Working with Gtest, we created a large volume of tests which prove the correctness of our code. We validated our code in a variety of methods, use the mathematical definitions available as well as comparing our results to the output of known frameworks. We found our framework capable of matching the accuracy those frameworks.

Additionally, we provided a full host implementation of the entire framework, mixing C and CUDA to complete a large and well formed application. In the software development life cycle, we have completed nearly a full cycle.

This work was rewarding. We feel that we have gained significant and valuable experience, and intend to continue working within the framework.

References

[1] Tahmid Abtahi, Amey Kulkarni, and Tinoosh Mohsenin. Accelerating convolutional neural network with fft on tiny cores. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4. IEEE, 2017.

[2] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[5] Pytorch. Kaiming initialization in pytorch, 2019. [Online; accessed 3-May-2019].

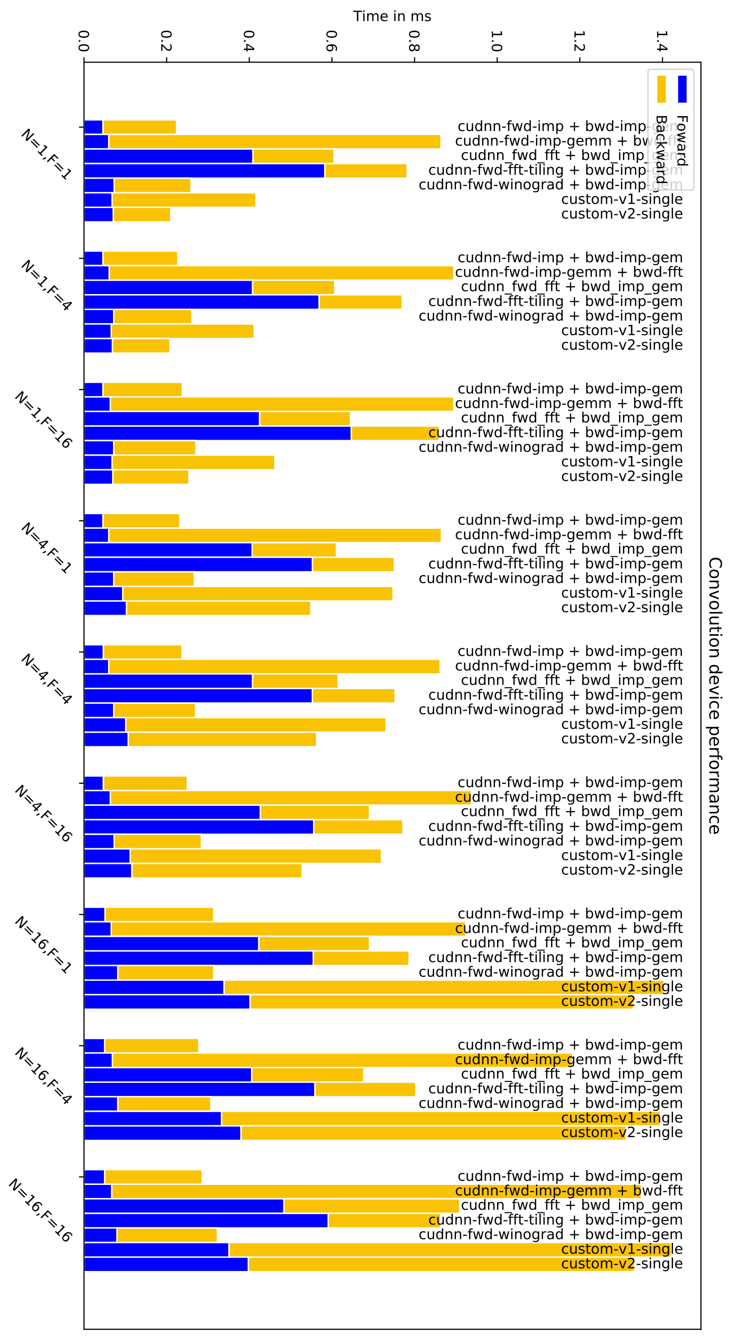


Figure 7: Performance of convolution layer using different algorithms and implementation