Technique Report for Convolutional Layers Optimization in swCaffe

Jiarui Fang

Tsinghua University

Abstract. Based on our previous work swDNN, we propose the SWCaffe [1] which maps one of the most popular deep learning framework Caffe to the Sunway TaihuLight. SWCaffe is a fully optimized framework which refactor major computation parts to SW26010 many-core processor. In this report, we provide our implementation details for convolutional layers implementation in swCaffe.

Keywords: Deep Learning, Many-core architecture

1 Methodology

In this report, we present our efforts on convolutional layers implementation, which is considered as the most time-consuming part in CNNs. When applying the convolutional layer solutions of the convolution routines in swDNN library [2] to SWCaffe, we met some problems.

- swDNN is only suitable for parameter configurations with large input and output channels sizes.
- swDNN is not designed for convolutional layer with pad.
- SW26010 provides a bad support for single-precision floating point operations, which is default precision option used in DNN.

1.1 Mixed Convolutional Layer Solution

We design a mixed strategy for convolutional layer implementations, which enable SWCaffe to support different convolutional layer parameter configurations. To support different convolutional layer parameter configurations in real CNN applications, we propose a mixed strategy combining the explicit GEMM plan used in original caffe and the implicit GEMM plan. As shown in Figure 1, for small parameter configurations, we use the conv_layer of original Caffe instead. In swCaffe, for implementation with swDNN routines, we can dynamically choose the best plan between original caffe one and the swDNN one.

[1] https://github.com/feifeibear/SWCaffe
[2] https://github.com/THUHPGC/swDNN

forward

```
if(B > =128 && B%128==0
Ni > 64 && Ni%32==0 &&
No > 64 && No%32 ==0) {
  swdnn_conv_forward();
} else {
  caffe_conv_forward();
}
```

backward

```
if(B > =128 &&
Ni > 64 && Ni%32==0 &&
No > 64 && No%32 ==0
Ni > 64 && Ni%32==0) {
  swdnn_conv_backward();
} else {
  if(Ni > =128 && Ni %128==0
    No > 64 && No%32 ==0
    Ni > 64 && Ni%32==0){
      swdnn_conv_backward_in_diff();
    } else {
      caffe_conv_backward_in_diff();
    }
    if(Ni>=128 && && Ni %128==0
      No > 64 && No% 32 ==0
      B > 64 && B % 32==0) {
      swdnn_conv_backward_weight_diff();
    } else {
      caffe_conv_backward_weight_diff();
    }
}
```

Fig. 1: Convolutional Layer Implemented in SWCaffe

1.2 im2col

In our mixed strategy, BLAS-based convolutional layers are implemented with gemm and im2col operations. Although gemm can be accelerated with BLAS routines on CPEs. im2col operations are still left on MPE. Figure 2 illustrated the time proportion of im2col and gemm in the layers using BLAS plan with batch size as 1. We can see that im2col and col2im take over 75% time.

Figure 3 shows our im2col and col2im plan on one CPE. During im2col process, each CPE reads one row of a input image into LDM buffer with DMA get operation. After adding with pad, CPE write $K \times K$ line of data into memory.

1.3 Padded Convolutional Layer

In most of CNNs, the convolutional layers are always implemented with pad. To avoid explicit memory copy to add pad, we control the memory access partern in convolutional layers. The convolutional layer in the forward propagation can be illustrated as :

$$top = bias + conv(add\_pad(bottom), weight); \tag{1}$$

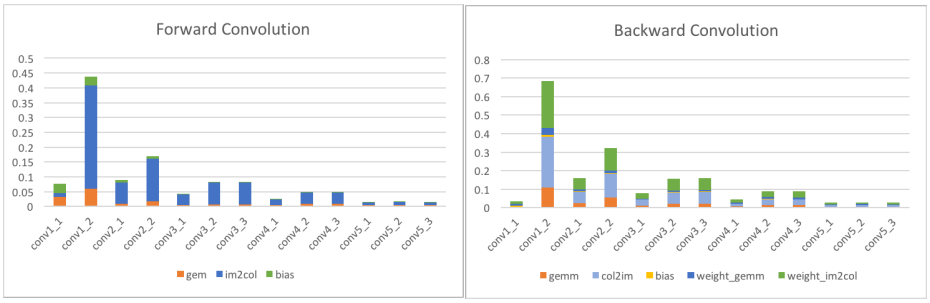


Fig. 2: Time propagation of convolutional layers with BLAS plan

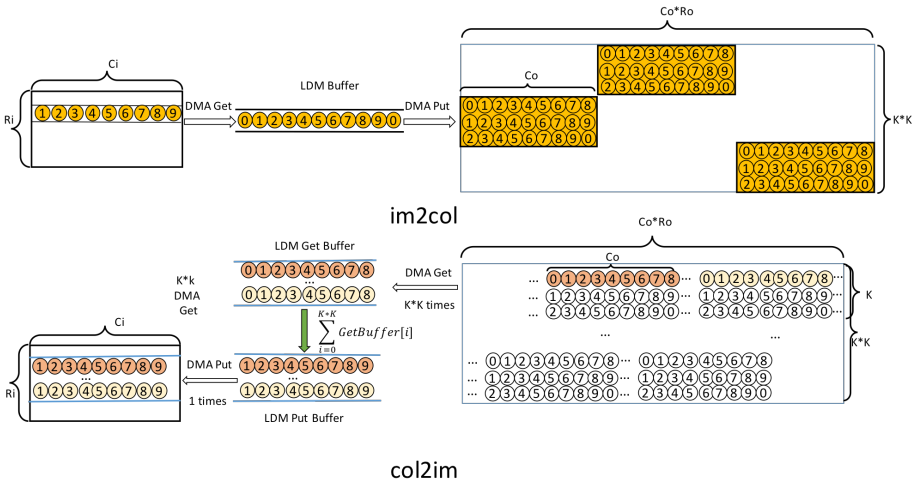


Fig. 3: Im2col on one CPE.

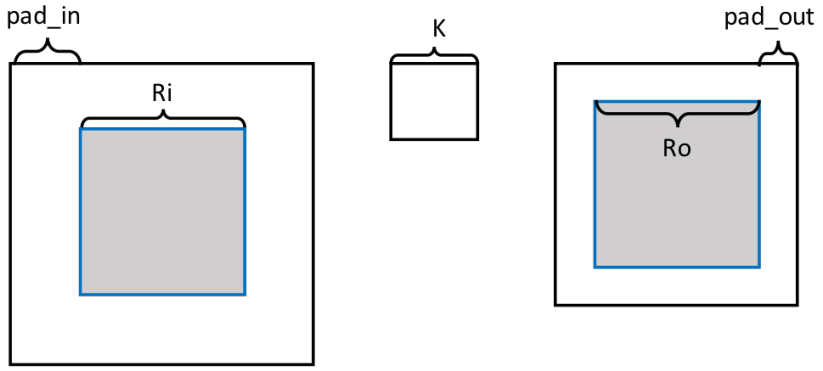
The convolutional layer in the backward propagation can be illustrated as

$$\begin{aligned} &delete\_pad(in\_diff, pad) = conv(add\_pad(out\_diff, K-1), rot180(weight)); \\ &weight\_diff = conv(add\_pad(in\_data, pad), out\_diff); \end{aligned} \tag{2}$$

We present the in_diff updating during backward propagation in Figure 5. The *pad_in* here is $K-1$. The *pad_out* here is the pad size for input.

Adding pad operation for convolutional layer of swDNN is nontrivial on the SW26010. Figure 4 illustrates the process of padded convolution on CPU with for loops. We propose a general padded convolutional layer on 64 CPEs of SW26010 in Algorithm 1. The for loops are performed with the virtual coordinates without considering padding. But the real memory access is conducted after mapping from virtual coordinates to real coordinates.

delete_pad(out, pad_out) = conv(*add_pad*(in, pad_in), weight)




```
for(cB = 0; cB < B; cB++)
 for(cNo = 0; cNo < No; ++cNo)
  for(cNi = 0; cNi < Ni; ++cNi)
   for(cKr = 0 ; cKr < K; cKr++)
    for(cKc = 0 ; cKc < K; cKc++)
     for(cRo = 0; cRo < Ro+2*pad_out; cRo++)
      for(cCo = 0; cCo < Co+2*pad_out; cCo++) {
        cRi=cRo+cKr; cCo=cCo+cKc;
        cRi_real = cRi-pad_in; cCi_real = cCi-pad_in;
        cCo_real = cCo-pad_out; cRo_real = cRo-pad_out;
        if(cRi_real, cCi_real, cRo_real, cCo_real are valid)
        Out(B,cNo,cRo_real,cCo_real) +=
In(B,cNi,cRi_real,cCi_real)*Weight(cNi,cNo,cRi_real,cCi_real);
}
```

Fig. 4: Convolutional Layer with pad

Algorithm 1 Padded Convolutional Layer

```
1: for Co_start = 0 : b_Co : C_o − 1+2 ∗ pad_out do
2:   for cR_o = 0 : R_o − 1+2 ∗ pad_out do
3:     cR_o_real = cR_o − pad_out
4:     if cRo_real < 0||cRo_real >= Ro then
5:       continue
6:     end if
7:     for cK_r = 0 : K_r − 1 do
8:       cR_i_real = cR_o + cK_r −pad_in
9:       if cR_i_real < 0||cR_i_real >= Ri then
10:        continue
11:      end if
12:      for cC_i = Co_start : Co_start + b_Co + K_c − 1 do
13:        cC_i_real = cC_i −pad_in
14:        if cC_i_real < 0||cC_i_real >= C_i then
15:          continue
16:        end if
17:        DMA get D_i ← N_i × B channels of input images(cC_i_real, cR_i_real)
18:        for cKc = 0 : K_c − 1 do
19:          cC_o = cC_i − cK_c
20:          cC_o_real = cC_o − pad_out
21:          if cC_o_real < 0||cC_o_real >= C_o then
22:            continue
23:          end if
24:          if cC_o >= Co_start and cC_o < Co_start + b_Co then
25:            DMA get W ← N_i × N_o channels of filter kernels (cK_c, cK_r)
26:            D_o(cC_o − Co_start)+ = W × D_i
27:          end if
28:        end for
29:      end for
30:    end for
31:    for cC_o = Co_start : Co_start + b_Co do
32:      cC_o_real = cC_o − pad_out
33:      if cC_o_real < 0||cC_o_real >= C_o then
34:        continue
35:      end if
36:      DMA put N_i × B channels of output images (cC_o_real, cR_o_real) ← D_o
37:    end for
38:  end for
39: end for
```

Algorithm 1 presents our padded convolutional layers on CPE clusters with blocking techniques to increase data locality in LDM.

1.4 Data Layout Transformation

As illustrated in Figure 7, the data layout which is suitable for swDNN for convolutional layer and the one used in original Caffe code are different. We design fast 4D data structure transformations on CPEs for layout transformation. However, transformations of the data structure each time when executing convolutional layer introduce some overhead. To reduce the transformation overhead, we add trans_layer in SWCaffe. Such optimization technique is based on the fact that the convolutional layers that can be accelerated with swDNN are gathered together. In such case, we can transform the data layout once and use such data structure during swDNN convolutaional layers and the ReLU, Pooling layers behind. And then, we transform data layout back to Caffe form after last swDNN

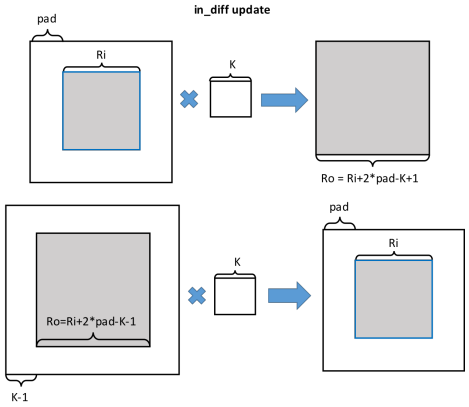


Fig. 5: Updating In_Diff in backward propagation

layer. It is noteworthy that the positions of trans_layer in forward propagation and backward propagation maybe different.

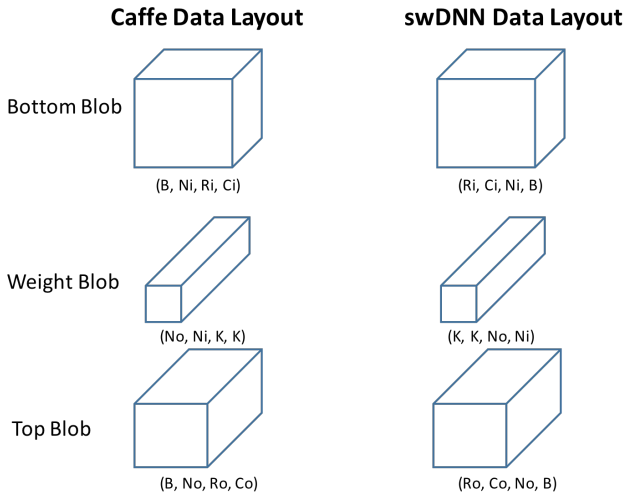


Fig. 6: Data Layout of swDNN and Caffe

1.5 Single-Precision Floating-Point Support

For float-swDNN, the instruction pipeline of GEMM kernel are more difficult be scheduled. It is because lack of instruction 'vldr' and 'vldc', which are only

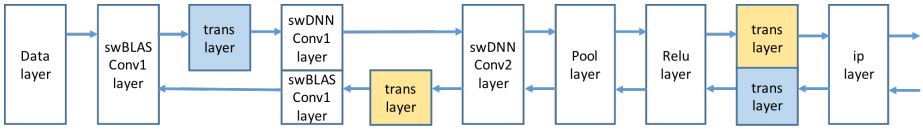


Fig. 7: Add trans_layer to the network structure.

supported for double vector operations. On the one hand. we replace instruction 'vldr' with vlds + putr and 'vldc' with vlds + putc. On the other hand, we pack float elements to double elements on the fly before executing of double-gemm kernels. We allocate the buffer in LDM as double precision. We extend the float LDM buffer to double inline after DMA operations. Such scheme introduces little overhead.

2 Acknowledgement

Thanks to the supports from Zheyu Zhang (Tsinghua Univ.), Liandeng Li (Tsinghua Univ.) and Xin You (Beihang Univ.). This work can not be finished so fast without help of yours.

References

1. https://github.com/feifeibear/SWCaffe