Technical Report of SWCaffe: A Deep Learning Framework for the Sunway TaihuLight

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 where major memory and computing operations are conducted on CPEs of SW26010. We categorize operations in a DNN as computing-centric and memory-centric kernels and provide parallel design for them considering the architectural characteristics. We compare the performance in one CG (742.4 GFlops) with intel 12-core E52680 V3 CPU (1280 GFlops). Our framework beats intel-Caffe with a speedup of 1.50x on VGG-16 network with single-precision floating-point data.

Keywords: Deep Learning, Many-core architecture

1 Introduction

Our paper makes the following contribution: A high performance Deep Learning framework that takes advantage of the computing power of emerging many-core processor SW26010.

2 Challenges

Large network structures require substantial computational and memory throughput. It is a big opportunity to utilize the latest SW26010 many-core processor for network training tasks. However, directly porting original Caffe to the SW26010 many-core architecture and accelerating the core computing with BLAS routines on CEPs is extremely inefficient. The challenges to design a highly-efficient deep learning framework on the Sunway TaihuLight come from the following points.

- Low bandwidth provided by the SW26010.
- Special cache memory hierarchy requires new memory access pattern.
- No automatic Vectorization
- Poor BLAS library. The BLAS designer, who is a quite famous professor in China, said he will never provide any support to me, because I pointed out bugs and shortcomings in their library publicly.

We classify the computing kernels into two categories according to their Arithmetic Intensity (AI). Convolutional, Inner product layers have high AI, while ReLU, Pooling and im2col layers have low AI.

3 Computing Efficiency Optimization

Convolutional layers are the most time-consuming parts in CNNs. When applying the convolutional layer solutions of the swDNN library 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.

3.1 Mixed Convolutional Layer Solution

We design a mixed strategy for convolutional layer implementations, which enable SWCaffe to support different convolutional layer parameter configurations. As shown in Figure 1, for small parameter configurations, we use the conv_layer of original Caffe instead.

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

3.2 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 :

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

The convolutional layer in the backward propagation can be illustrated as

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

Adding pad operation for convolutional layer of swDNN is nontrivial on the SW26010. Figure 2 illustrates the process of padded convolution and shows a naive implementation on CPE. We propose a general padded convolutional layer on 64 CPEs of SW26010 in Algorithm 1. In this case, the input and output images are both added with pad before convolution.

We present the in_diff updating during backward propagation in Figure 3. The *pad_in* here is $K-1$. The *pad_out* here is the pad size for input.

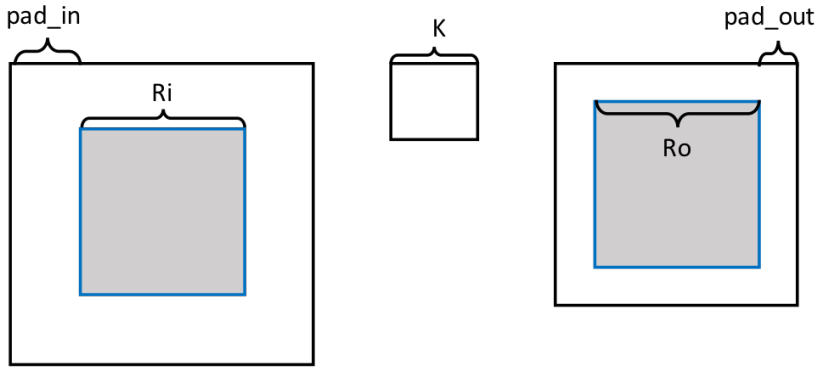
3.3 Data Layout Transformation

As illustrated in Figure 5, 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 layer. It is noteworthy that the positions of trans_layer in forward propagation and backward propagation maybe different.

3.4 Single-Precision Floating-Point Support

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

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. 2: Convolutional Layer with pad

Algorithm 1 Padded Convolutional Layer

for $Co_{start} = 0 : b_{Co} : C_o - 1{+}2 * pad\_out$ **do**
  for $cR_o = 0 : R_o - 1{+}2 * pad\_out$ **do**
    $cR_o\_real = cR_o - pad\_out$
    if $cRo\_real < 0 || cRo\_real >= Ro$ **then**
      continue
    end if
    for $cK_r = 0 : K_r - 1$ **do**
      $cR_i\_real = cR_o + cK_r - pad\_in$
      if $cR_i\_real < 0 || cR_i\_real >= Ri$ **then**
        continue
      end if
      for $cC_i = Co_{start} : Co_{start} + b_{Co} + K_c - 1$ **do**
        $cC_i\_real = cC_i - pad\_in$
        if $cC_i\_real < 0 || cC_i\_real >= C_i$ **then**
          continue
        end if
        DMA get $D_i \leftarrow N_i \times B$ channels of input images(cC_i_real, cR_i_real)
        for $cKc = 0 : K_c - 1$ **do**
          $cC_o = cC_i - cK_c$
          $cC_o\_real = cC_o - pad\_out$
          if $cC_o\_real < 0 || cC_o\_real >= C_o$ **then**
            continue
          end if
          if $cC_o >= Co_{start}$ and $cC_o < Co_{start} + b_{Co}$ **then**
            DMA get $W \leftarrow N_i \times N_o$ channels of filter kernels (cK_c, cK_r)
            $D_o(cC_o - Co_{start}) += W \times D_i$
          end if
        end for
      end for
    end for
    for $cC_o = Co_{start} : Co_{start} + b_{Co}$ **do**
      $cC_o\_real = cC_o - pad\_out$
      if $cC_o\_real < 0 || cC_o\_real >= C_o$ **then**
        continue
      end if
      DMA put $N_i \times B$ channels of output images $(cC_o\_real, cR_o\_real) \leftarrow D_o$
    end for
  end for
end for

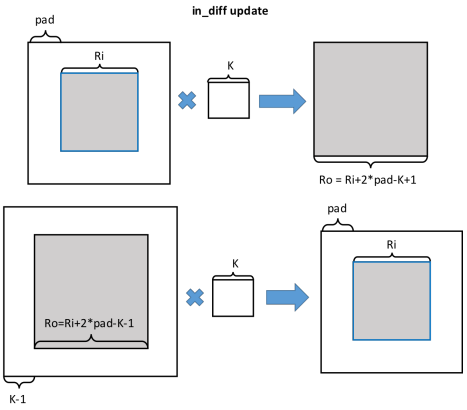


Fig. 3: Updating In_Diff in backward propagation

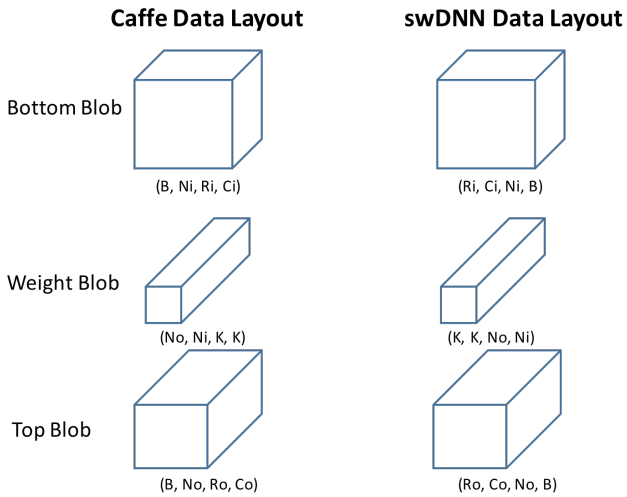


Fig. 4: Data Layout of swDNN and Caffe

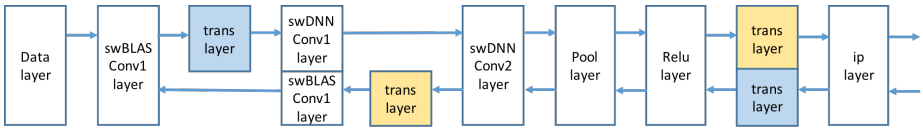


Fig. 5: Add trans_layer to the network structure.

4 Bandwidth Efficiency Optimization

4.1 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 6 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 7 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.

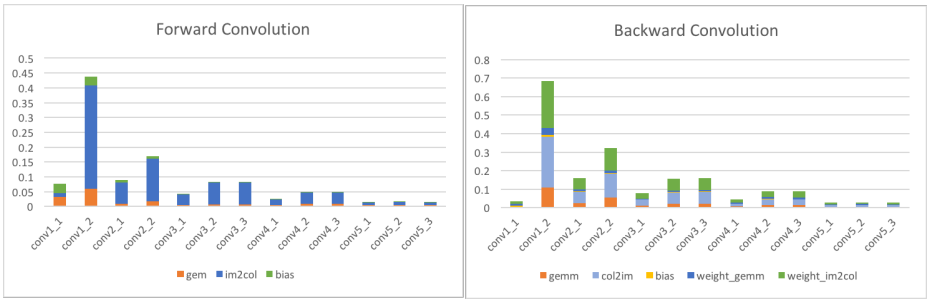


Fig. 6: Time propagation of convolutional layers with BLAS plan

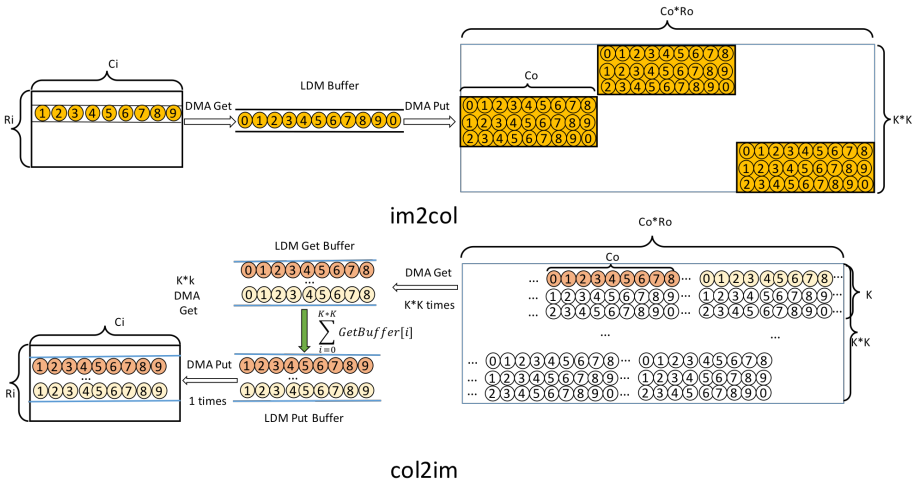


Fig. 7: Im2col on one CPE.

4.2 swPool

4.3 swRelu

5 Results

We compared SWCaffe on 1/4 CPU, i.e. one CG on SW26010 (fp peak performance is 742.4 GFlops), with the Intel E5 2680V3 (fp peark performance is 1280 GFlops). We present the performance of intel accelerated with OpenBLAS and SWCaffe version on single-precision floating-point data.

5.1 VGG-16 Results

The macroarchitecture of VGG16 can be seen in Figure 8.

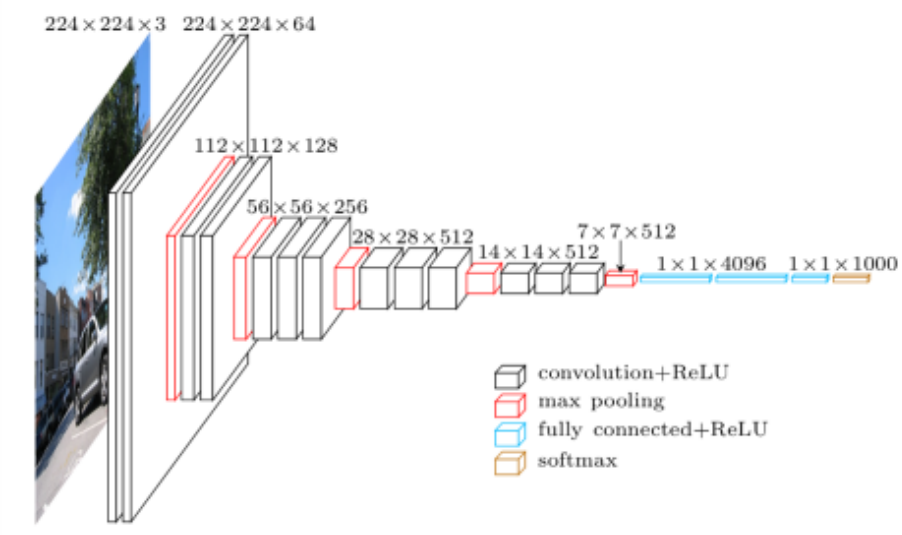


Fig. 8: macroarchitecture of VGG16

The time of each layer in forward propagation is presented in Figure 9. Compared with the intel, the conv_1 layer is so slow with SWCaffe during forward propagation. It is because that the first layer has a small channel size and swBLAS library is not good at leading GEMM with thin matrices when adopt BLAS-CONV plan. Look forward for better supports from swBLAS library.

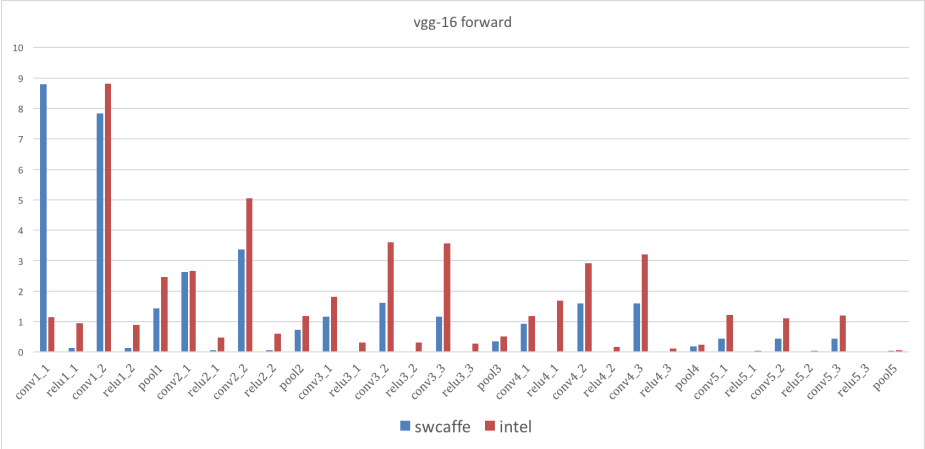


Fig. 9: Time of each layer in forward propagation

The time of each layer in backward propagation is presented in Figure 10.

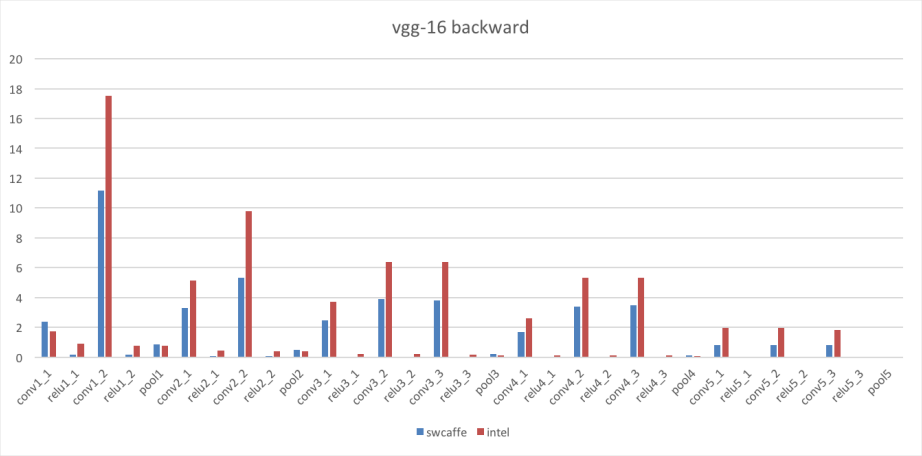


Fig. 10: Time of each layer in forward propagation

The speedup of each layer in forward propagation is presented in Figure 11.

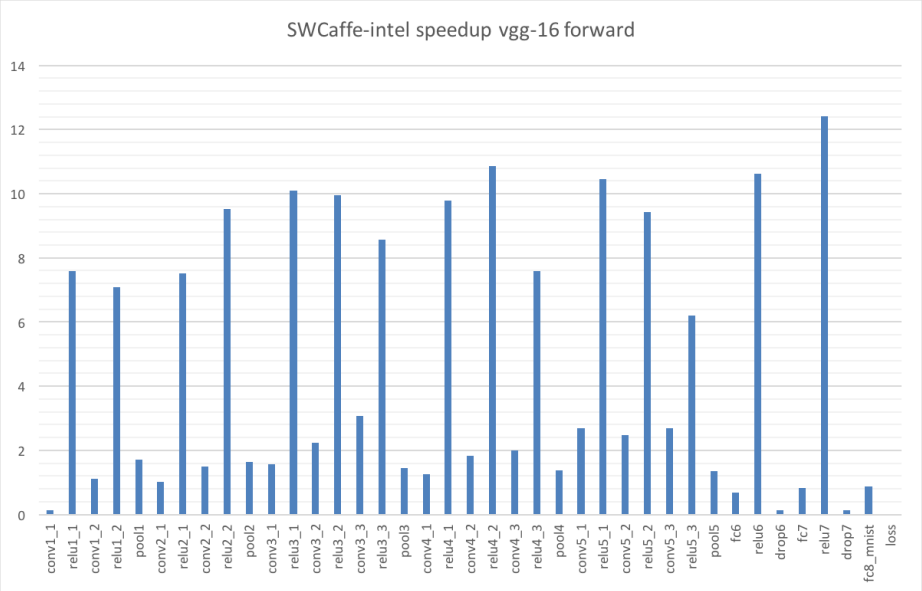


Fig. 11: Speedup of each layer in forward propagation

The speedup of each layer in backward propagation is presented in Figure 12.

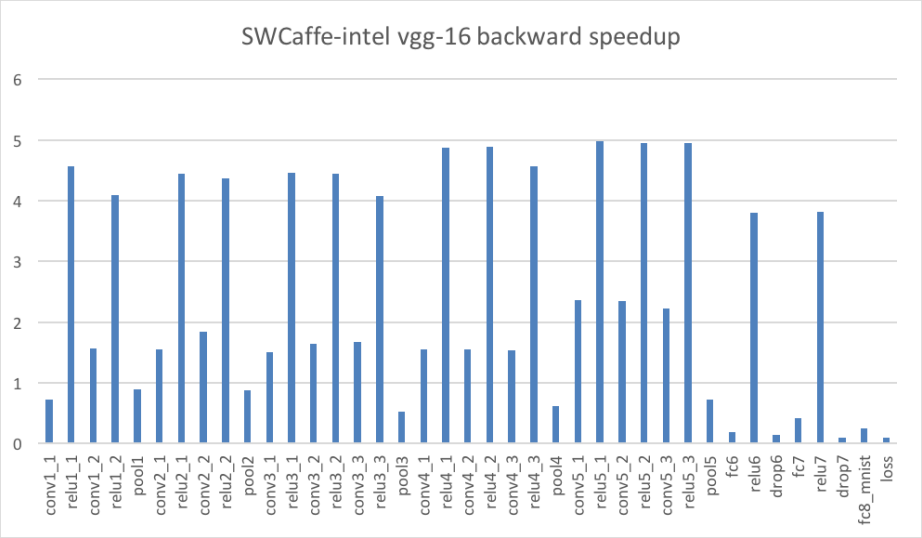


Fig. 12: Speedup of each layer in forward propagation

5.2 ALEXNET Results

SW26010 is poor at dealing with pow and sqrt. Currently, we can not figure out a good solution for LRN layers, so we do not take them into consideration here. In addition, we do not use im2col in the CONV-BLAS plan of Alexnet, due to the stride for conv is not 1. Fortunately, for the latest new network, stride is always to be 1. However, we will still fix this problem in the near future. The macroarchitecture of AlexNet can be seen in Figure 13.

The performance of AlexNet can be seen in Figure 14. The speedup of AlexNet can be seen in Figure 15. The poor performance of first few conv_layers is due to the lack of im2col support with stride scheme.

6 UNDO

- Integrating image-size-aware swDNN version into SWCaffe.
- check results for conv_pad_full.
- Supporting trans_layer
- RNN profile

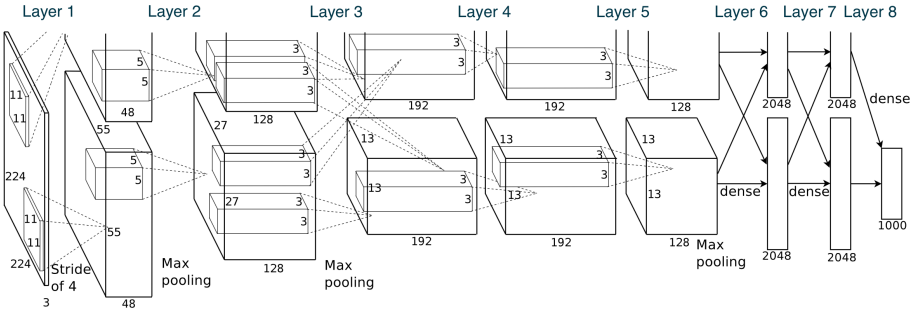


Fig. 13: macroarchitecture of AexNet

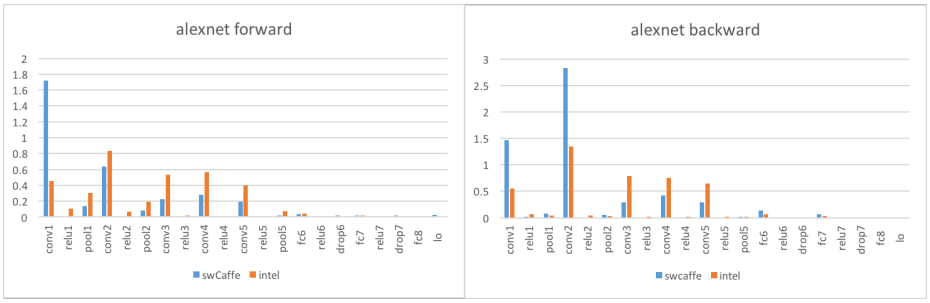


Fig. 14: Time of each layer in forward propagation of AlexNet

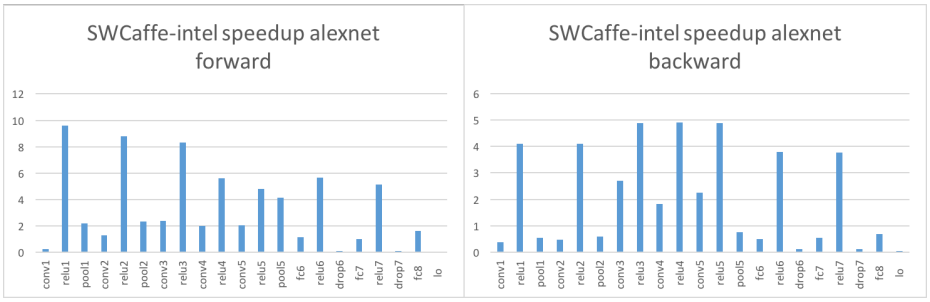


Fig. 15: Speedup of each layer in forward propagation of AlexNet

7 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